# Signal

## Sweep-based data acquisition and analysis system

**Signal Version 7.07**

# Signal Version 7.07

## The Signal help as a printed manual

*Cambridge Electronic Design Ltd.*

# Signal Version 7.07

## Revision History

## Acknowledgements

# Table of Contents

# File menu ............................................................................................**9-154**

# Edit menu ...........................................................................................**10-179**

# Single-channel analysis

# Cursor menu

# Sampling menu

# Script menu ................................................................16-295

# Window menu ...............................................................17-297

# Help menu .................................................................18-300

# Script language ...........................................................19-303

# Digital filtering ................................................................ 20-698

# Programmable Signal conditioners ................................. 21-715

# Amplifier telegraphs ...................................................... 22-721

# Auxiliary states devices ................................................. 23-733

# Technical support ........................................................... 24-770

# Signal version 7

The Signal software running under Windows together with one of the CED 1401 family of interfaces gives a PC the power to capture and analyse multi-channel waveform and time marker data.

Signal is designed to let you manipulate your data using the familiar Windows idioms. You can arrange the windows to display the data within them to best advantage and copy and paste the results to other applications. Alternatively, you can obtain printer hard copy directly from the application. When you close a data file, Signal saves the screen format and analysis window setup associated with it. When you open a file, Signal restores the configuration, so it is easy to resume work where you stopped in a previous session.

You can analyse sections of data by reading off values at and between cursors, or by applying the built-in frame by frame automated analyses such as waveform averaging and power spectrum. More ambitious users can further automate both data capture and analysis with scripts. The script language is described in The Signal script language manual.

### New features in version 7

Version 7 of Signal is completely compatible with earlier versions; it will read data files and sampling configurations created by versions 2, 3, 4, 5 or 6 without problems. Scripts that ran in versions 3, 4, 5 and 6 should work without modification (with a very few exceptions). The main new features in version 7 are:

- The auxiliary states device support has been extended to allow use of multiple devices in the same experiment.
- The auxiliary states device support has been extended to include the PowerMAG and DuoMAG ranges of transcranial magnetic stimulators.
- The auxiliary states device support has been extended to include the Digitimer DS8R biphasic constant current stimulator.
- Viewing and editing auxiliary states device settings during sampling is now possible, access is via the Sampling menu or a new control bar.
- Integer variables in the script language are now 64-bits in size, allowing massively greater integer values to be used.
- There is a new information window which can be used to generate customisable high-visibility displays.
- Script language support for the Continuous Wavelet Transform has been added.

There are many other improvements and more are planned. You can find a full list of new features, bug fixes and changes in the Revision history. Licensed users of Signal can download updated releases of Signal within the same version from our web site www.ced.co.uk as they become available.

### Hardware required

The absolute minimum requirement to run the program is a Pentium II with 256 MB of memory running Windows 7. All 64-bit versions of Windows are supported. The more memory you have and the faster the processor, the better Signal will run. A graphics accelerator will greatly improve drawing and scrolling speeds.

To sample data, you will need a CED Power1401 mk I, mk II or -3/3A or a Micro1401 mk I, mk II, -3 or -4, dynamic clamping requires a Power1401 mk II or -3/3A or a Micro4. Version 7 does not support the standard 1401 or the 1401*plus*, if you want to use a 1401*plus* you will have to use version 6, a version 6 installer is supplied on the Signal version 7 CD. Unless the exact type of 1401 is specified, when the terms Micro1401 and Power1401 are used in this manual they refer to all versions of these types of 1401.

### File icons

 The various file types in Signal have icons so that you can easily recognise them when you minimise their windows. The icons will automatically be used for the relevant files by programs such as Windows Explorer. All the icons have a set of waveforms to remind you of the application to which they belong. The icon to the left is the Signal application icon that you double-click to launch Signal from the Signal program group.

These icons are for Signal data files. The icon on the left represents Signal CFS data files or file views. The icon on the right represents an XY data file: a saved XY view. If you double-click on one of these in Windows Explorer it will launch the Signal application (if it is not already running) and open the data file in a data or XY view.

These icons are for text-type Signal documents and files. The icon on the left represents a text file, which can hold any textual data. The centre icon represents a Signal script file. Script files hold script programs that execute within Signal to automate analysis or to customise Signal behaviour in some way. The icon on the right represents a Signal sequencer file. These files store sequences of output pulses for use during sampling.

These icons are for other files created by Signal. The icon on the left represents a Signal sampling configuration, while the one on the right represents a Signal resources file.

### Direct access to the raw data

Some users may wish to write their own applications that manipulate Signal data files directly. A C library: The CFS library is available from CED. The library includes all functions necessary to read or write Signal data files from Windows programs. This library is available, along with complete documentation in PDF format, from the CED web site (https://ced.co.uk).

### Using this on-line help

Much of the information in this help file is also in the manuals Signal for Windows and The Signal script language. If you prefer printed documentation you may like to read the manuals and use this system for context-based help from the program (by pressing F1 or clicking Help buttons).

We do not explain standard procedures, for example clicking and dragging, using menu short-cut keys or using a file open dialog; we expect that you are already familiar with them. We use standard Windows idioms wherever possible so that you feel at home and have a consistent interface to work with.

The Overview page at the start of the Getting started chapter aims to give you a general understanding of Signal and what it does.

The Getting started section suggests some tasks you might undertake to get started with the system. We have supplied example data files for you to experiment with; there is no need to have your own data available at this time. This is followed by sections on General information, Sampling data, Pulse outputs, Sequencer outputs and Sampling with multiple states.

The next sections are a reference section to all the menu commands in Signal: File, Edit, View, Analysis, Cursor, Sample, Script, Window and Help.

Once you are familiar with the program, you may wish to investigate the script language so you can automate your data capture and analysis.

There are also sections on specific topics, such as: Sampling with clamp support, Single channel analysis, Digital filtering, Programmable signal conditioners, Telegraph support and Auxiliary states devices.

The *Signal Training Course Manual* covers selected topics in more detail. It is particularly useful for script authors as the approach is much more descriptive than the reference material presented here.

### Credits

Nearly all of the Signal code was written by CED programmers and is copyright Cambridge Electronic Design Ltd. However there are a few other sources of code who should be thanked and acknowledged (though of course CED is still responsible for any shortcomings):

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

- *XML resource files are handled using the XML library developed by Arseny Kapoulkine and available from https://pugixml.org/.*

# Installation

To be reading this help you must have already installed Signal. However, you may need to move it to another system (mindful of the Signal Software License).

Your installation disk is serialised to personalise it and the Signal software to you. Please do not allow others to install unlicensed copies of Signal.

Just put the CD-ROM in the drive and it will start the installation. You can also run the installation manually by opening the `Signal7` folder on the CD-ROM, then the `Disk1` folder and running `setup.exe`.

### During installation

You must select a suitable drive and folder for Signal and personalise your copy with your name and organisation. You can have earlier versions of Signal on the same system as long as they are in different folders. If you have a previous version on your system, make sure you install to a different folder. The installation program copies the Signal program plus demonstration, help, tutorial and example files. It also copies and installs all required 1401 support (device drivers and control panels). In rare cases you may need to install drivers manually; the installation program will tell you if this is the case and point you at detailed instructions. Your system may require a restart after installation to get all 1401 device drivers up to date.

### Installation location

By default, versions of Signal before 5.08 were installed by default into a C drive directory such as `C:\Signal7`. As installation outside `C:\Program Files` is frowned-upon these days, modern versions of Signal install by default into a directory inside `C:\Program Files` and we have made minor adjustments to ensure that Signal will work smoothly in the new location. When running from `C:\Program Files` Signal cannot save data, sampling configurations and other values inside the Signal installation directory as was often done previously, so the Signal installer creates two new directories for you to save your files in. These are:

*My Documents\Signal7*
This directory is always created and if you are installing Signal inside `C:\Program Files` this folder is also where the the installer puts the example data files, scripts and other useful files that were previously installed inside the Signal installation folder. This folder is user-specific (for each user that logs on a different folder acts as `My Documents`) and so you can use it for all of the files which you do not need to share with other users of the computer.

*Public Documents\Signal7Shared*
This directory is also always created and can be used for files that you want to share with other users of the computer.

### Custom install

To install without 1401 support, or to install extra information and documentation (which includes the latest copies of the software manuals) choose Custom installation.

### After installation

If you are new to Signal, please work through the Getting Started tutorial in the online help. Where you go next depends on your requirements. The Signal Training Course Manual is more descriptive than the other manuals, which are organised as reference material. However, it covers all versions of Signal and you will occasionally need to refer to the other manuals or this online help for version 7 specific details.

# Updating and removing Signal

You can update your copy of Signal to the latest version 7 release free of charge from our web site: www.ced.co.uk. You can only update a correctly installed copy of Signal version 7, though you can install the update into a different directory prom the previously installed software. There are full instructions for downloading the update on the web site.

Once you have downloaded the Signal update, you will find that the update process is very similar to the original installation process, except that you must already have a properly installed of Signal for Windows version 7 on your computer.

Updates will include both bug fixes and new features. We will notify you by email (if we know your email address) of new releases. You can also register for this service on our web site. To stop emails, reply to them and ask to be removed from the list.

### Removing Signal

You can remove Signal from your system: open the system Control Panel, select Add/Remove Programs, select CED Signal for Windows version 7 and click Remove. This removes all of the files installed with Signal; you will not lose any data or other files that you created.

# Getting started

This section of the Signal online help consists of a series of familiarisation exercises to help you become familiar with Signal. You can begin the exercises by clicking here.

## Signal overview

### What is Signal and what does it do?

Signal is a program used to collect and analyse information from (normally bio-medical) scientific experiments. The data that Signal works with are primarily waveforms - voltages that vary with time - each waveform is represented by a separate channel. Signal collects and manipulates waveform data in chunks that are called frames, the frames in a Signal data file are usually all of the same length - rather like a software oscilloscope.

#### Collecting data with Signal

Signal uses the CED 1401 data acquisition hardware to sample voltages and generate new data files, the 1401 can also be used to generate outputs (stimuli) during sampling. To sample new data using Signal you would normally read a sampling configuration from a disk file or edit an existing sampling configuration to generate a new configuration, there are many options available to control how Signal samples data and generates outputs. Once the sampling configuration is satisfactory you can make use of it by clicking on the Sample Now button in the sampling configuration dialog, or using the File menu New command and selecting a data file. This creates the new (empty) data file, gets the 1401 ready for use and makes the sampling control panel visible. Sampling is started, stopped and controlled using the sampling control panel, once sampling has finished the new data file can be saved to disk to make it permanent.

#### Viewing data with Signal

Signal data can be visualised in a number of different ways, from a simple display drawing waveforms as dots, line or cubic splines, to overdrawn channels, to overdrawn frames using 3D perspective. Waveform channels can be displayed using differing Y axis ranges and the appearance of both the Y and X axes can be extensively customised. The time or Y axis range shown can be interactively adjusted in various ways.

#### Analysing data with Signal

Signal can carry out many types of data analysis. Vertical and horizontal cursors can be used to mark positions of interest within the data and measurements can be taken using the current vertical cursor positions. Virtual channels can be used to create simulated channel data using an evaluated expression and data in other channels and memory channels can be created to hold temporary or easily modified data. New data views can be created holding the results of processing existing data, for example by waveform averaging, and memory channels or XY views can be generated holding measurements taken from other data views - the ability of vertical cursors to be active and automatically move to locate features in channel data is particularly useful with these measurements. Other available analysis techniques include curve fitting, digital filtering and interactive data modification across multiple frames.

### What can Signal be used for?

Hopefully you already know that Signal is suitable for the type of experiment that you want to carry out! However Signal is a very versatile program and can be used for a wide range of experiments. Here are some of its uses and capabilities:

#### Evoked response studies

Measuring responses to stimuli requires that the data acquisition be synchronised with stimulus presentation by suitably triggering the data acquisition and often that data frames be collected at a high rate to minimise discomfort for the subject. Automatic generation of an averaged response as part of the data acquisition process helps to ensure that the data acquisition is proceeding satisfactorily.

### Complex stimulus generation

Generation of complex stimuli including arbitrary waveforms are a useful tool in many types of experiment. The ability to generate multiple different stimulus sets and run through them in a predefined or randomised order allows more complex experiments to be undertaken. If you require extremely complex behaviour including reacting to the sampled data, the output sequencer allows still more control over what and when outputs are generated.

### TMS studies

Signal can make Transcranial Magnetic Stimulation studies much easier & quicker to carry out by controlling a Magstim stimulator and adjusting the stimulus parameters such as the power level automatically during an experiment. Errors in record keeping are avoided by automatically recording the stimulation settings along with the data and the Magstim health and temperature are automatically monitored throughout use.

### Clamping studies

Signal can be used for both voltage clamp and current clamp experiments and is able to provide a quick analysis of incoming data for the purpose of monitoring cell health or seal quality. Single-channel voltage clamping experiments are also supported, in particular by the inclusion of the SCAN analysis technique developed by Prof. David Colquhoun. When using advanced 1401 hardware such as the Power1401-3 is is possible to use dynamic clamping techniques to simulate the presence of (or cancel out) particular ion channels.

### System customisation

The Signal script language is a complete programming language which can be used to extend the capabilities of Signal by allowing specialised analyses, automatic analysis and interaction with the data acquisition to perform more complex experiments. The sample and scripts toolbar provide a simpler customisation mechanism by allowing frequently used sampling configurations & scripts to be loaded by a single mouse-click.

# Signal basics tutorial

This tutorial teaches you the basic operations that manipulate Signal data files. Signal is a large program with many features; this short tutorial will get you started finding your way around. It is intended that you should work through the tutorial sections in the order in which they appear, though some sections are optional as they are only relevant in certain circumstances.

### Run Signal

To work through this tutorial you need to be running Signal. If Signal is not already running, click the Start button and select Programs. Select the Signal entry in the list of programs and inside this you will find the Signal program. Select the program to start Signal.

### Clean up unwanted windows

If you have any windows open in Signal, close them before you start to follow the tutorial instructions. The quickest way to close all windows is use the Window menu Close All command (click on Window in the menu bar, then click on Close All). You can also close windows by double-clicking the control menu at the top left of a window, or with the File menu Close command.

# Opening a file to view

In this step you will open the demonstration data file that was shipped with Signal. Follow these steps:

1. Open the File menu and select the Open... command

2. Open the data directory (inside the Signal7 directory in My Documents for a Program Files install) and double click on example.cfs (if you have moved files since your installation you may have to search around to find this file, if all else fails, use any Signal CFS data file)

3. A new window will open. Arrange the help window and this new window so that you can see both.

4. To follow the tutorial, you should see a window holding at least one data channel and with a horizontal scroll bar at the bottom. If this is not the case your file is in a mess!

# Clean up a messy file

If the file does not display properly this is because the last user of this file didn't put it away tidily. You can tidy up by following these instructions:

1. Click on the window labelled `example.cfs`.

2. From the View menu select the Standard Display option.

3. If the fonts used seem too large or too small open the View menu Font dialog and select Times New Roman, Regular 10 point.

4. If the colours are grotesque open the View menu and select Use Black And White (which always works) or select the Change Colours... option and choose a better combination.

# Signal display overview
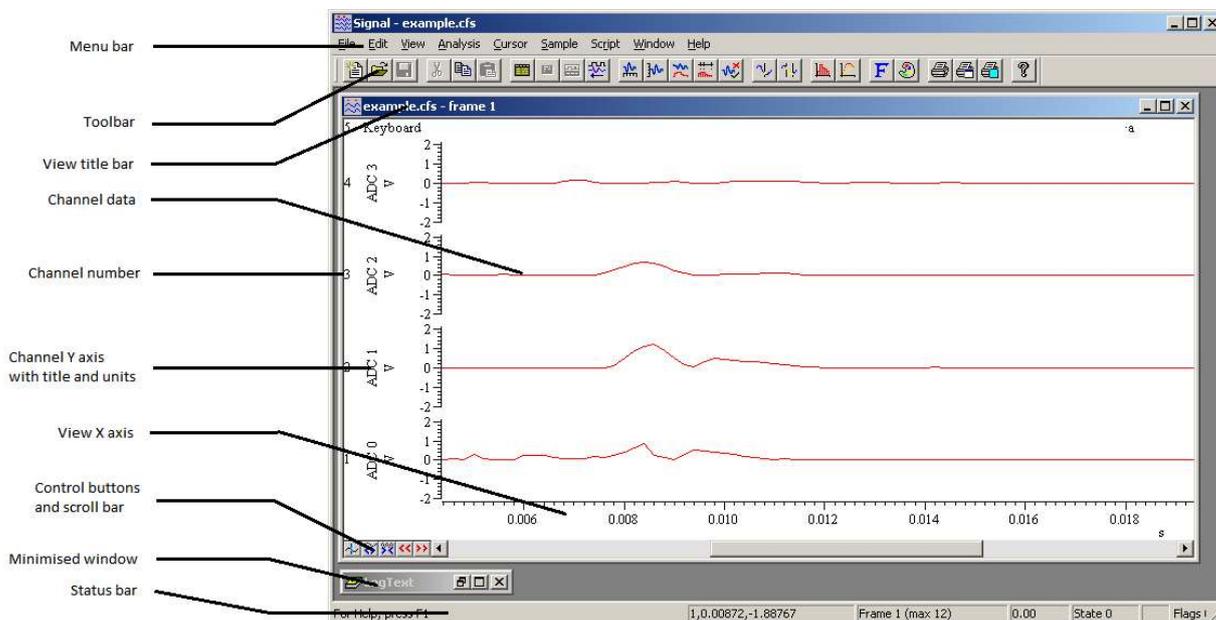


Here is the Signal program with the example data file open, Signal displays the file in the state in which it was last saved (as long as it can find a file with the same name and the extension `.sgrx` holding the previous settings). The `example.cfs` window shows data from a specific frame in the file. We call this a *file view* because it displays data from a file, along with *memory views* (which are described later), file views are examples of *data views*, so called because they display standard Signal data.

**The Signal window**
You can see a number of separate components that are part of the Signal window and are normally present regardless of what Signal is currently doing:

*Menu bar*
As normal in Windows programs, menus provide your main access to commands used to make Signal take an action. The menu bar contains a number of separate menus: File, Edit, View, Analysis, Cursor, Sample, Window and Help, each menu contains relevant commands, for example the File menu contains the Open command used to open a file. The menus and commands available can vary according to the type of view in use. There are separate chapters in the Signal help describing the commands available from the various menus.

*Toolbar*
Just below the menu bar is the toolbar; a line of buttons that, when pressed, carry out common actions. Each toolbar button matches an action from the menu. The buttons are shown in this manual with the menu items. To find out what a toolbar button does, leave the mouse pointer over the button for a few seconds. If you want, you can hide the toolbar (& show it again) using View menu commands. You can also drag the toolbar and "stick" it to any of the 4 sides of the application window (this is known as docking the toolbar) or simply leave it 'floating' in a separate window inside the main Signal window.

*Status bar*

The Status bar is at the bottom of the Signal window, it provides information about the current view ( and again can be shown and hidden using the View menu). The status bar holds a number of 'panes' plus an open area on the left. This leftmost portion shows menu item prompts; as you move the mouse pointer over the menus, it will show text describing the item underneath the mouse pointer. The other panes each show a particular item of information. If the space available is too small for all of the panes, panes disappear starting at the right hand side. From the left, the status bar panes are:

Cur Pos    If the mouse pointer is over a part of a data or XY view that has axes this pane displays the pointer X and Y positions, the first figure is the channel number from which the Y value is taken. For a text-type view it displays the current text cursor position in lines and columns.

Frame      This pane shows the current frame number for the current view and the maximum frame number in the view. If the current view is not a data document then this pane is blank. See below for a discussion on frames.

Start      This pane, adjacent to the frame number, shows the absolute start time for the frame - this is the time for the start of the frame relative to the time at which sampling started.

State      This pane shows the state code (described below) for the current frame in the current view as a decimal number or (if available) as a state label with the number appended. It is blank if the current view is not a data document.

Tag        If the current view is a data document and the current frame is tagged (described below), this pane shows the text TAG, otherwise it is blank.

Flags      This pane shows the flags for the current frame in the current view as an 8 digit hexadecimal (base 16) number (hexadecimal format makes the individual flag states visible), each digit shows the state of four flags with the highest on the left. This pane is blank if the current view is not a data document view.

Caps       If the keyboard Caps lock is on, this pane displays the text CAPS.

Num        If the keyboard Num lock is on, this pane displays the text NUM.

Record     This pane displays REC if Signal is recording user actions into a script.

**The data view window**

You can also see a number of separate components inside the `example.cfs` data view window:

*Title bar*

This shows the name of the data file used by the view and the current frame number, during sampling extra information relating to the state of sampling can also be shown here. At the right of the title bar are buttons used to minimise (show as an icon), maximise and close the view.

*Channel data*

These areas show the actual data from the file, which can be drawn in various ways.

*Channel number*

The channel number is shown to the extreme left (can be changed to right) of the channel data, it can be used to select or de-select channels (this is described later on).

*Channel Y axis*

The Y axis shows the range of data values that are being plotted in the corresponding channel data area, you can adjust the range shown as necessary (this is described later on) and also whether it is displayed to the left (as shown here) or right of the channel data. The Y axis also displays the channel title and the units for the data. Not all channels have a Y axis, for example the keyboard marker channel at the top of the data view.

*X axis*

The X axis shows the time range over which data is being shown, you can adjust the range shown as necessary (this is described later on). The X units are also shown, for normal Signal data you can choose between seconds (s), milliseconds (ms) and microseconds (us).

*Control buttons*

These buttons below the Y axes are used to zoom the X axis, add cursors and move to the next or previous frame, they are introduced in detail below.

*Scroll bar*

If the X axis range shown is less than the total available in the data frame the scroll bar will be activated (it's always present but may be disabled). The scroll bar 'thumb' visually indicates the section of data that is being shown, you can move back and forth through the data in the current frame by dragging the thumb.

# Data channels

There are several *data channels* displayed in the window. These channels can hold different types of data: in the example file channels 1 to 4 hold waveform data, channel 5 holds keyboard markers. Waveform channels are normally displayed with channel 1 at the top, marker channels are shown above waveform channels but you can re-arrange them as you like. Ordinary data channels are stored in the CFS data file, but there are also other types of channel: memory channels are stored in the resource file associated with the CFS data file (this allows them to be easily modified or deleted) and virtual channels are not stored anywhere - their data is calculated on-the-fly by arithmetic operations on data from other channels.

### Selecting channels

You select channels by clicking on the channel number to the left of the channel. Signal highlights the channel number for selected channels. Hold down the Shift key and click on a channel to select all channels between it and the last selection. Hold down Ctrl to select discontinuous channels. Many commands can operate on a list of selected channels (for example y axis display optimisation). To clear any selection, click on the rectangular area below the channel Y axes and to the left of the X axis.

From Signal 6.04, the mouse pointer changes when you move it over a channel number, or over the rectangular area at the bottom left if any channels are selected. to remind you that you can select the channel.

# Frame buttons

The bottom window edge holds five buttons and a scroll bar. The scroll bar controls movement through the frame when the displayed time range is less than the frame length. If you resize the window, the same data is drawn, scaled to the window. The two buttons to the left of the scroll bar change the currently displayed frame.

Click on these buttons to move to the previous or next frame in the file. All CFS files contain a number of separate frames which hold similar data, you can use these buttons to move from one frame to another. If the currently displayed frame is the first available, then the previous frame button is grayed-out, similarly for the next frame button. These buttons correspond to the View menu Previous frame and Next frame commands, there is also a Goto frame command.

### What buttons and scroll bar?

If the buttons and scroll bar are not visible, select the Standard Display option from the View menu, which should tidy up the display so that you can see them.

# About frames

Frames are a central concept within the Signal software. A CFS file or Signal data document contains one or more data frames, each frame corresponds to a sweep of sampling. A data document view normally only displays one frame at a time, this is the current frame for that view. You can have multiple views of the same data document, each view can have a different current frame so you can examine separate parts of the file simultaneously. It is also possible to overdraw multiple frames in the same view.

Each frame in a Signal data document has the same number and type of channels, but may have varying frame start and end times. Each frame holds channel data from the various channels. Usually, all of the waveform channels will have the same number of data points, while the number of markers can vary. In addition to this frame data there are a number of other data items attached to each frame; a frame comment, the frame state, tag and flags, 16 user frame variables and others:

Comment    a single line of text, of length up to 72 characters, that can be read or written using Signal scripts.

| State | a 32-bit number that will normally have a value from 0 to 256, it is intended for use in conditional analysis where each state value corresponds to a separate condition in the experiment, but may be used for any purpose. |
|---|---|
| Start | a floating point number holding the absolute start time for the frame. This value is the time for the frame x axis zero relative to the start of sampling. For files collected by Signal version 1.00 or collected using Fast triggers mode this will always be zero. |
| Tag | a single yes/no switch; each frame can be tagged or not. This is intended for any purpose where a set of frames need to be marked for analysis or attention. |
| Flags | a set of 32 flags, each of which may be set or clear. These flags are available for any user-defined purpose, currently they are only useable from the Signal script language. |
| User | a set of 16 floating point numbers that can be read or written using Signal scripts. They are intended for any purpose required. |
| Others | when using an auxiliary states device such as a Magstim, extra frame variables are created to show the Magstim power levels and other settings. When sampling with multiple states, a StateL variable is created, this holds the state label corresponding to the frame state or is blank if no state label is set. |

When you are working with Signal data, the current frame for the view is held in memory. This frame will be discarded when the view switches to a different frame. Any changes made to the frame data while it is held in memory must be saved before the new frame is loaded or the changes will be lost. You can write the changed data back into the file using the File menu Save command, or you can use the Preferences dialog Data page to select what happens if the frame is changed while data is unsaved. Changes made to non-channel data such as the frame state or flags are always saved.

Often in Signal you will need to specify the frame or frames to be used for an operation. There are a number of ways of doing this, you can select the current frame in the view or you can enter a single frame number directly. To specify more than one frame, you can enter a frame list such as '1..50,60,61,70..80', or you can select options such as All frames, Tagged frames or Frames with state n, giving a wide range of possibilities. These frame specification mechanisms are also available from within the Signal script language.

# Zoom buttons

The bottom window edge holds three buttons and a scroll bar. The scroll bar controls movement through the file. If you resize the window, the same data is drawn, scaled to the window. The two buttons to the left of the scroll bar change the time range.

 This button halves the time range (zoom in). The left edge of the display remains fixed. You can zoom in until the ratio between the frame length and the width of a pixel reaches 2,147,483,647.

 This button doubles the time range (zoom out). The left edge of the window does not move unless the start plus the new width exceeds the length of the frame. If the new width exceeds the frame length, the entire frame is displayed.

# Cursor buttons

These buttons add vertical or horizontal cursors to the display. Up to 10 cursors can be present in a window, it is also possible to have up to 9 horizontal cursors. A vertical cursor is a dashed line used to mark positions on the X axis, a horizontal cursor marks levels on a channel. You can remove cursors with the Cursor menu Delete option or by right-clicking on them. You add cursors in four ways.

1. Click and release the relevant button to add a cursor in the centre of the window or centre of the first channel with a Y axis (for horizontal cursors).

2. The Cursor menu New cursor command or it's Ctrl-| shortcut adds a cursor in the centre of the window. The New Horizontal cursor command adds a horizontal cursor in the centre of the first channel with a Y axis.

3. If you right-click on a data view, the pop-up context menu will include an item for a new vertical cursor. If you clicked on a channel with a Y axis, the channel-specific sub menu will include an item to create a new horizontal cursor.

4. From a script you can use the `CursorNew()` or `CursorSet()` commands to create vertical cursors, `HCursorNew()` to create horizontal cursors. Script users have full control over the cursors.

# Cursor style and pointers

Click on the cursor button so that at least one cursor is visible. The mouse pointer changes over a cursor or a cursor label:

This indicates that you can drag the cursor. If you drag beyond the window edge, the window scrolls. The further beyond the edge, the faster the scroll. Dragging hides the label unless the `Ctrl` key is down or you drag the label.

If you position the mouse pointer over the cursor label, the pointer changes to a 4-headed arrow to indicate that you can drag both the cursor and the label. This can be useful when preparing an image for publication and you need the cursor label to be clear of data. If you move the pointer to one side, or hold down the shift key, the pointer becomes a two-headed vertical arrow and you can drag the label, but not the cursor.

There are four labelling styles for the cursor: no label, position, position and cursor number, and number alone. You select the style with the Cursor menu Label mode option or by right-clicking on the cursor and selecting Set Label.

# Zoom in on an area

Move the mouse pointer to the waveform channel. Drag a rectangle round a waveform feature with the mouse and release the button. The window displays the area within the rectangle. If the rectangle covers more than one channel, only the time axis changes. If your rectangle lies within a channel and has zero width, only the y (vertical) axis changes, similarly a rectangle with no height changes only the X axis.

The mouse pointer changes to a magnifying glass when you hold the mouse button down in the data channel area to show that you are about to magnify the data.

If you hold down the Ctrl key as you drag, the mouse pointer changes to the un-magnify symbol and the rectangle holding the channel area shrinks to the rectangle you drag

Now move the mouse pointer over an axis; you will see it changing to show a scrolling or stretching axis. You drag the axis to directly shift or stretch it; if you are over the axis ticks you get a shift while over the numbers generates a stretch in or out (the mouse pointer changes to show what operation you will get).

Whatever method you use to scale the data, you can return to the previous display using the Edit menu Undo command or the keyboard short-cut `Ctrl+Z`. If you release the mouse button with the pointer in the same position from which you started the drag, the display does not change.
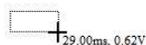
If you hold down the `Alt` key before you click and drag, Signal displays the size of the dragged rectangle next to the mouse pointer and does not zoom the display. Release `Alt`, then you can use the `C` key to copy the current measurement to the clipboard and the `L` key to copy it to the Log view.

# Zoom a channel

You can zoom one channel to use the entire display area!

Double click anywhere in the waveform channel. The channel will expand to occupy the entire window.

Double click in the waveform channel again and your previous display is restored.

You can also change the display to show a channel and all duplicates of it by holding down the `Ctrl` key, then double-clicking on a channel. This is useful with sorted spikes, where duplicates of a channel are used to display spike classes.

# Using x and y axes to scroll and zoom
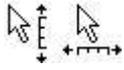
When the cursor is over the tick marks of an axis, you can drag the axis. This maintains the current axis scaling and the window moves to keep pace with the mouse pointer. You can do this with most x and y axes in Signal. This is particularly useful for y axes as they do not have a vertical scroll bar. The window does not update until you release the mouse button. If you hold down the `Ctrl` key, the window will update continuously.
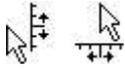
When the cursor is over the axis numbers, a click and drag changes the axis scaling. The effect depends on the position of zero on the axis. If the zero point is visible, the scaling is done around the zero point; the zero point is fixed and you drag the point you clicked towards or away from zero. If the zero point is not visible, the fixed point is the middle of the axis and you drag the point you clicked towards and away from the middle of the axis

In a data or XY view you can drag the y axis so as to invert the axis. You can prevent this happening by setting an option in the Edit Preferences dialog Display page. You are not allowed to invert the X axis.

# X Range dialog

Click this toolbar button or double-click the time (X) axis of the display to open the X Range dialog. Experiment with the time axis. You can type new positions or use the pop-up menus next to each field to access cursor positions and the maximum time in the file.

| Left | Sets the window start time. You can type times either using the selected X axis units, or as a number followed by 's', 'ms' or 'us' to force the units. In addition to typing times or selecting a time from the drop-down list, you can type in expressions using the maths symbols + (add), - (subtract), * (multiply) and / (divide). You can also use round brackets. For example, to display from 1 second before cursor 1 to one second past cursor 1 set Left to Cursor(1)-1 and Right to Cursor(1)+1. The Draw button is disabled if you type an invalid expression, or if the Right value is less than or equal to the Left value or if the new range is the same as the current range. |
|---|---|
| Right | Sets the window end time using the same format as the Left field. |
| Width | Shows the window width. You can either set the left and right positions, or the left position and the width. Check the box to keep the width the same when the Left field changes or to enter a width directly. |
| Large tick spacing | Use this to customise the X axis appearance by forcing the interval (in X axis units) between large ticks on the axis. If you enter a value that would produce an illegible axis it will be ignored. |
| Tick subdivisions | Use this to customise the X axis appearance by setting the number of small-tick subdivisions between large ticks on the axis. Again, any value entered that produces an unusable axis will be ignored. |
| Logarithmic | Check this box for a logarithmic rather than a linear X axis. |
| Auto adjust units | Check this to cause the units displayed on the axis to switch to multiples of powers of 10 when zoomed well in or well out in order to keep the figures shown on the axis sensible. |
| Show All | Expands the time axis to display all the file and closes the dialog. |
| Draw | If the axis range had been edited, use this to redraw the data to match the new range. You do not need to press Draw to see the effect of other changes made within the dialog. |
| Close | This closes the dialog; it does not update the axis range. |
| Cancel | This undoes all changes made with the dialog and closes it. |

# Y Range dialog

Click this toolbar button or double-click on the Y axis of a waveform channel to open the Y Range dialog for that channel. This dialog changes the y axis range and attributes for one or more channels. Experiment with the Y axis range.

| | |
|---|---|
| Channel | This is a pop-up menu from which you can select any channel with a y axis, or all channels with y axes, or all selected channels. You can also enter a list of channel numbers such as 1..3,6 directly. |
| Top | Enter the upper limit for the Y axis here. |
| Bottom | Enter the lower limit for the Y axis here. |
| Axis type | Select a linear, logarithmic or square root axis. |
| Large tick spacing | Use this to customise the Y axis appearance by forcing the interval (in Y axis units) between large ticks on the axis. If you enter a value that would produce an illegible axis it will be ignored. |
| Tick subdivisions | Use this to customise the Y axis appearance by setting the number of small-tick subdivisions between large ticks on the axis. Again, any value entered that produces an unusable axis will be ignored. |
| Auto units | This is used to select between the units displayed on the axis being static, switching to multiples of powers of 1000 in order to keep the figures sensible when zoomed well in or well out, or you can select the SI Prefix option which does the same thing by changing the units using SI prefixes, for example by converting V to mV when you are zoomed in. |
| Optimise | Press this button to change the Y axis ranges of the set channel(s) to fit the range of the displayed data. |
| Show All | Press this button to change the Y axis ranges to the data limits for the channel(s), if available. If the channel has no limits to its data this does the same as the Optimise button. |
| Draw | Press this button to redraw the data using the currently entered Y axis range. |
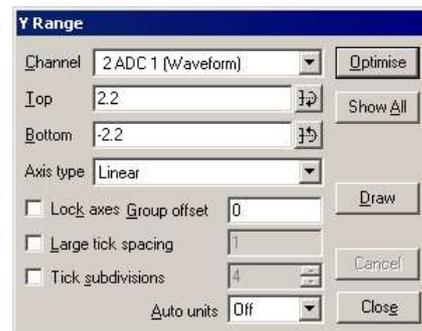
If no channel has a y axis, open this dialog from the View menu.

# Waveform draw mode dialog

Data files hold two basic channel types: waveform and markers. Waveform channels hold a list of values representing the waveform amplitude at successive time intervals. Marker channels hold the times at which something happened (and more data, depending on the marker type).

Open the View menu Draw Mode dialog. Experiment with different drawing modes for channels 1 to 4. Click Draw to update the display without closing the dialog. Click OK to close the dialog.
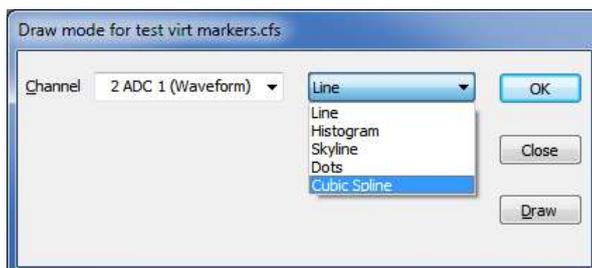
If you want to find out more about waveform drawing modes, you can find a complete description in the View menu chapter, Draw mode section.

# Customise display

Open the View menu Customise display dialog. This sets the channels to display in your window. With up to 80 channels in a file, this ability is quite important if you are to see any detail!

The list on the left of the dialog holds all the channels that can be displayed. Check the box next to a channel to include it in the display list.

You can also show and hide the axes, background grid and the horizontal scroll bar in the window from this dialog, and switch to displaying Y axes with horizontal labels.

Click the OK button to see the result. If you make a complete mess of your window you can use the View menu Standard Display command to clean things up.
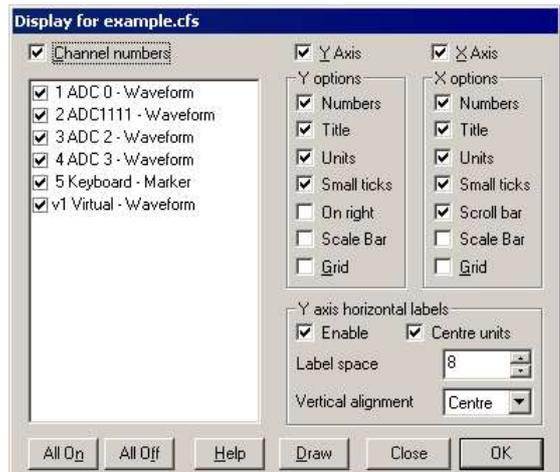
# Channel order

Make sure that the example file is the current window and use the View menu Standard Display command to tidy things up. Click on the ADC 0 channel number (1) and drag it down over the other channel numbers.

As the mouse pointer passes over each channel, a horizontal line appears above or below the channel. This horizontal line shows where the selected channel will be dropped. Drag until you have a horizontal line below channel 2 and release the mouse button. Channel 1 will now move to the middle of the channel list, between channels 2 and 3. Type Ctrl+Z or use the Edit menu Undo to remove your change.

You can move more than one channel at a time. Signal moves all the channels that are selected when you start the drag operation. For example, hold down Ctrl and click on the channel 3 number. Keep Ctrl down and click and drag the channel 2 number. When you release, both channels will move. The mouse pointer shows a tick when you are in a position where dropping will work.

The usual Signal channel order is with low numbers at the top of the screen. If you prefer low numbers at the bottom of the screen, open the Preferences dialog Display page and uncheck Standard Display shows lowest numbered channel at the top, then use the View menu Standard Display command.

# Channel spacing

Hold down the Shift key and move the mouse over the data area of channel 3. Keep the Shift key down and click. Drag up and down and release the mouse. You will see that this adjusts the vertical space allocated to the channel.

When you click with Shift down, the mouse jumps to the nearest channel boundary and you can change the boundary position by dragging. With Shift down, you can move the edge up and down as far as the next channel edge. You can undo changes or use Standard Display to restore normal sizes.

If you add Ctrl, all channels with a Y axis are scaled. If there are no channels with a Y axis above or below the drag point, then all channels scale. You can force all channels to scale by lifting your finger off the Shift key (leaving Ctrl down) after you start to drag the boundary.

# Channel overdraw

You can overlay channels of data in a file or memory view. This is done interactively by dragging channel numbers on top of each other or from a script by using the ChanOrder() command. To do this in the example file:

Click the "3" of channel 3 and drag it on top of Channel 1 and release.

Channels 1 and 3 now share the same Y axis space with the channel numbers stacked up next to the Y axis. The visible Y axis is for the top channel number in the stack. You can promote another channel in the stack to the top

by double-clicking the channel number. The channels retain their own y axes and scaling. You can remove a channel by dragging the channel number to a new position.

When you drag channels, and at least one of the selected channels has a Y axis, you can drop the channels with a Y axis on top of another channel with a Y axis. As you drag, a hollow rectangle appears around suitable dropping zones. You can also drop between channels when a horizontal line appears.

Merged channels are drawn such that the channel with the visible Y axis is drawn last. If you have a channel that fills in areas, such as a marker channel drawn as rate mode, put it at the bottom of the stack, as it will mask channels below it in the stack.

Use the View menu Standard Display command to tidy things up before you continue.

# Frame overdrawing

Open the View menu Overdraw settings... dialog. Specify frames 1..4 as the frame list and click OK. Use the View menu Overdraw frames command to turn overdrawing on and off. Experiment with different frame selections and with the colour cycling display mode.

The frame display list is a list of frames that will be shown in addition to the current frame when overdrawing in standard (non-3D) mode. When using colour cycling each overdrawn frame is drawn in a different colour, or all the display list frames are drawn in the colour specified by the Frame list traces item in the colour setup dialog, or you can fade to the background colour or channel secondary colour. The current frame will be displayed in its usual colour if it is not in the frame list. All the standard mechanisms for selecting frames are available. 3D overdrawing options are also available, this differs from the simple overdrawing mechanisms in that the current frame is always shown at the front and therefore only frames that are before the current frame are overdrawn - so that the 3D array shows a time history of frames with the oldest at the back.
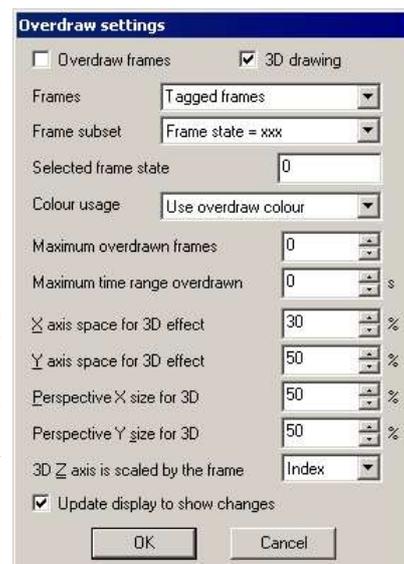
# Cursor measurements

Make sure there are cursors in the window.

Click the toolbar button (the toolbar is at the top of the Signal window, just below the menus, it holds buttons for common tasks) or use the Cursor menu Display Y Values option. The columns show the cursor times and values. For channels without a y axis, the value is the next marker time after the cursor.

The Time zero and Y zero check-boxes select relative rather than absolute measurements; the radio buttons set the reference cursor. Reference cursor values are unchanged; values at other cursors have the reference value subtracted.

If you move cursor or alter a channel display mode, the values change. If you add or remove channels in the display, they are added or removed in the dialog.

You can copy dialog fields to the clipboard. Click and drag for multiple selections. Click the top or left hand cells to select an entire column or row. Hold down the Ctrl key to make non-contiguous row or column selections.

# Cursor regions

Click the toolbar button or use the Cursor menu Cursor Regions command. Experiment with changing cursor positions and channel display types. The regions dialog looks at the data values between cursors. There are nine modes set by the drop down list at the bottom of the dialog:

Curve area  The area between a waveform trace and the line joining its end points, no measurement for markers.

| | |
|---|---|
| Mean | The average level of a waveform, or the number of markers divided by the width of the region. |
| Slope | The gradient of the least-squares best fit line to the waveform, no measurement for markers. |
| Area | The area between a waveform and the y axis 0.0 level, or the number of markers in the region. |
| Sum | The sum of all waveform points, or the number of markers in the region. |
| Modulus | The area between a waveform and the y axis 0.0 level, all values taken as positive, no measurement for markers. |
| Maximum | The maximum waveform (or marker as rate) value or interval between markers. |
| Minimum | The minimum waveform (or marker as rate) value or interval between markers. |
| Peak-Peak | The difference between the Maximum and Minimum. |
| SD | The standard deviation from the mean of the waveform values. |
| RMS amplitude | The RMS amplitude of the waveform values. |
| Extreme | The largest of the Maximum and Minimum, taken as absolute values. |
| Peak | The amplitude of the first peak found. |
| Trough | The amplitude of the first trough found. |
| Point Count | The number of sample points between the cursors. |
| SEM | The standard error in the mean of the waveform values. |
| RMS error | The RMS error relative to the mean of the waveform values. |

You can also make relative measurements by checking the Zero region box and choosing a reference region.

# Memory views

Windows holding data from a file are called *File views*. There is another type of data window, called a *Memory view*, that holds the result of analysing file view data, or other generated data. Memory views are very similar indeed to file views, the only difference is that they are wholly held in memory and do not have an associated CFS data file. Saving a memory view to disk will create a CFS data file which, if re-opened, will create a file view. The simple way of generating a memory view is to analyse file view data. There are two steps in the analysis:

1.  You set the type of analysis, the channels to analyse, the number of points or bins to generate and any other parameters required. This creates a new, empty memory view.

2.  You select frames from the file view to analyse and Signal calculates the result and adds it to the memory view.

You may repeat step 2 as many times as is required to accumulate results from different frames or selections of frames. The new window behaves like a file view containing a single frame of data.

# Make a waveform average

Close all windows except the original view of `example.cfs`. Then:

1.  Click this button on the toolbar or use the Analysis menu New Memory View command to select Waveform Average.

2.  Set the channels to analyse. The channel list in the pop-up menu includes suitable channels only. All Channels of the  example file is the best for this example.

3.  Set the width of the average, we suggest 0.04.

4.  Leave the first field set to 0.0 seconds. This is the offset from the start of the frame for the data to be averaged.

5.  Click the New button to create a new window and open the Process dialog.

If you want to find out more about waveform averaging, you can find a complete description in the Analysis menu chapter, New Memory View section, under Waveform Average.

# Process dialog

Now set the region of the data file to analyse:

1.  Set the frames for analysis and the frame subset to All frames.

2.  Check the Optimise Y axis after process box to scale the y axis to the data automatically.

3.  Click Process to analyse the data and display the result.

The Clear result view before process check box sets the memory view contents to zero before you analyse the data, otherwise each new result is added to the previous one. The Settings button takes you back to the previous step.

# Repeating analysis

You can add more data into the analysis or change the analysis settings and analyse again:

1.  Recall the analysis dialog by selecting the Process command from the Analysis menu.

2.  Click the Process button again. The data in the result window will not change, as it is an average, but the sweeps shown by the View menu Info command will double (if you have not checked the Clear result view before process box).

3.  Recall the process settings dialog by selecting the Process settings command from the Analysis menu.

4.  Change the width to 0.02, and the start offset to 0.01 then click the Change button.

5.  Select frames for processing with the new settings and process as before.

# Memory view usage

Experiment with this new window.

You will find that the memory view behaves in a very similar manner to the original file view, but has a single frame.

# XY views

In addition to File and memory views, there are XY views. These hold multiple data channels (up to 256) that share the same x and y axes. Each channel is a list of (x,y) co-ordinates. Each channel has its own point marking style, line style and colour.

Use the Script menu Run Script command and select the Load and run… command. Locate the Scripts folder (in the folder where you installed Signal), and open the file clock.sgs.

Signal will load and run this script which generates an analogue clock in an XY view. You can move and resize the clock window. You can stop the script running (and regain control of Signal) by clicking on the OK button at the upper right hand side of the Signal window. You can read more about XY views in the script language manual.

Signal can also create XY views holding data taken from measurements from data files. Use the analysis menu Measurements command to select XY View (Trend Plot) analysis. This will give you a dialog for the trend plot settings allowing the trend plot measurements to be defined. Once the trend plot settings are set a normal process dialog is used to select the frames from which measurements are taken. As for memory view processing, trend plot generation can be saved as part of a sampling configuration.

You can use the View menu to manipulate the an XY view. Try Customise display, XYDrawMode, Options and Colours to see XY-specific features.

If you want to find out more about trend plots, you can find a complete description in the Analysis menu chapter, Measurements section, under XY View(Trend plot).

*End of "Getting started tutorial"*
If you have worked through these operations, you have the basic skills required to make use of Signal for interactive data analysis.

To learn about using Signal to collect data to create new data files you should read the Sampling data, Pulse outputs while sampling and Sampling with multiple states chapters of the documentation and experiment with the sampling configuration dialog.

To learn about all of the various forms of data analysis that Signal can carry out you should read the Analysis menu chapter of the documentation.

To learn about using the script language for analysis you should read the Script language chapter of the documentation and investigate the example scripts provided with Signal.

# General information

There are several topics that are important throughout Signal. Channel lists are used in many dialogs and also in the script language, as are frame lists. Expressions can be used in many dialogs where x axis values are wanted and also more generally. There are many keyboard shortcuts in Signal; they are gathered together here. We have also included information on a number of other topics.

## View types

We often refer to *Views* when we describe features of Signal. The types of views that an interactive user of Signal has to deal with have a window on the screen with an associated menu and a data file on disk. These views come under the categories of Data based views and Text based views. If you write scripts you will also have to deal with two other types of views: External views (with no associated window) and Other. If you do write scripts, you should read the more technical description of views and view handles in the Script language section.

### Data file based views
There are three types of data views in Signal:

### File views (*.cfs)
These display data that is being sampled or has been sampled by Signal or imported into Signal and are associated with a .cfs file. They hold channels of various types of data, chiefly waveforms and contain multiple frames of data all using the same channels. They are created by the File menu New command (to create a new file for sampling based on the current sampling configuration), by opening an existing file or by importing a foreign format file or from the script language with the `FileOpen(name$, 0,...)` or `FileNew(0,...)` commands.

### Memory views (saved as *.cfs)
Memory views are much the same as File views, but do not have a data file associated with them. When they are saved to disk they are saved as .cfs files and when re-opened have become File views. They are generated from other File or Memory views or by scripts. You can open saved memory views with the File menu Open command and create new ones when the current view is a File or Memory view from the Analysis menu New Memory View command.

### XY views (*.sxy)
XY views are generated from analysis of File or Memory views or by the script language. They display multiple channels of data points defined by (x,y) co-ordinates. The data can be drawn as dots, lines, closed curves or as histograms, allowing a wide variety of displays to be created. Existing XY files are opened by the File menu Open command.

### Text based views
Text-based views correspond to standard text files. In Signal version 7, we expect text files saved on disk to be in UTF-8 format, but we will attempt to read text in an range of formats and translate as appropriate. We save text files in UTF-8. If you are firmly of the ASCII text persuasion, do not worry as this is a subset of UTF-8. We distinguish the varieties of text file we support by the file extension used to store them on disk. The Text file view types that Signal supports are:

### The Log view
The log view is unusual in that it does not have an associated data file, though you can save it as a text file. The Log view is used as a convenient place for Signal to display messages and there is a dedicated script command, `PrintLog()`, to make it easy to write to from the script language, regardless of the current view.

### Text views (*.txt)
These views are created by opening an existing .txt file or by creating a new text file with the File menu New command or from the script language with the `FileOpen(name$, 1,...)` or `FileNew(1,...)` commands. They display the text in the file and the text can be added to it with the script `Print()` command. You can choose to display a gutter margin which can hold bookmarks, and a line number margin, which is normally hidden. Text

views are normally used from the script language to build output files that the user can view. Script users can also create external text files, which are faster but have no display window.

### Script views (*.sgs)

These views are created by opening an existing script file or by creating a new one with the File menu New command or from the script language with the `FileOpen(name$, 3,...)` or `FileNew(3,...)` commands. Scripts are used to automate Signal operations. Most Signal interactive functionality can be controlled by a script, and there is a lot of functionality that is only available from a script.

Script views display the text in a window. Script views highlight your script files based on the syntax of the script language, which can make scripts easier to understand and write. They are also used to debug scripts. They have additional folding margin which shows the structure of the script and allows you to fold away blocks of code based on the script structure.

### Output sequencer views (*.pls)

These views are created by opening an existing output sequence file or by creating a new one with the File menu New command. Output sequences are used when sampling data with a 1401 interface to sequence DAC and digital outputs. They can also be used to respond to inputs very quickly, enabling real-time responses to external signals. Like scripts, output sequences support syntax highlighting and and folding.

There is an alternative way to control outputs while sampling using the Pulses editor. However the output of the pulses editor is converted to a sequence before it is used and a text sequence gives you the most complete control of the 1401 outputs.

## Grid views (*.sggx)

These were added as an experiment at version 6.04. They are mainly intended for use from the script language, but can be used interactively. They provide a spreadsheet-style grid that can be used to store data and to build tables for printing. They are not intended for vast quantities of data. You can read more about them here.

## External views

These views have data files and view handles, but no display windows. They are used for text and binary data files and are only available to users of the script language.

### External text files (*.txt, *.*)

These files are opened and created with the `FileOpen(name$, 8,...)` script command. You can read data from the files with `Read()` and write data with `Print()`; the files are accessed sequentially, that is you will read from the start of the file through to the end or write from the start to the end or append to the end. Because there is no associated screen image to update when you write or to scroll through on a read, external text files can be much faster to use than a text view with an associated window, particularly if the file is very big.

### External binary files (*.*)

These files are opened and created with the `FileOpen(name$, 9,...)` script command. You can read and write real, integer and text data to these files in a wide variety of formats. The files allow random access. See the `BRead()`, `BReadSize()`, `BWrite()`, `BWriteSize()` and `BSeek()` commands for details. You can read or write just about any type of file as a binary file, but to do it usefully you need to know the exact file format. The example for the `BWriteSize()` command shows you how to create a .bmp file using the script language. The example for `Spline2D()` has an example of generating a bitmap file.

## Other view types

These view types are for objects that are controllable by the script using view handles, but that do not fit the view types listed above. These are things like the various control bars, the Signal application window and so on.

# Data file and view topics

## Channel lists

In most places where Signal prompts you for a data file channel you can select a channel or group of channels from a drop down list or you can type in a channel list directly. A channel list is a list of channel numbers or channel ranges separated by commas, while a channel range is two channel numbers separated by two periods or a hyphen. For example 4..7 means channels 4, 5, 6 and 7 while the range 7-4 is equivalent to channels 7, 6, 5 and 4. In nearly all situations the order of channels is not significant and these two ranges are treated the same. The channel list 1,3..5,7 therefore means channel numbers 1, 3, 4, 5 and 7. Virtual channels can be specified by using the vn channel numbers shown by Signal, for example 1..3,v1,v3..v5. Memory channels can be specified in a similar manner by using mn (so m1, m2 etc.).

In most cases, Signal checks channel lists and removes channels that are not suitable for the operation. For example, if you open the example.cfs file supplied with Signal, select a waveform average and type in a channel list of 1..32 and then click on another field in the settings dialog, Signal will reformat the list as 1..4 as these are the only suitable channels in the data file. It is not an error for a channel list to include unsuitable channels, however it is an error for a channel list to include no suitable channels.

You can also specify groups of channels in other ways, for example dialogs that require you to enter a channel list will also, if any channels are currently selected, offer you a Selected channels option. Many of these dialogs will also allow you to select and de-select channels while the dialog is visible. You will normally also be offered All channels and All visible channels as other alternatives.

Channel lists can also be used in script commands, for example: ChanShow("1..4"). Script commands that will accept this format describe the argument as cSpc. You can find more information about channel specifications in the script language documentation.

## Frame lists

In many places, you will be asked to specify the frame or frames to be used for an operation. Dialogs that do this generally provide you with a pair of frame selectors plus an associated numeric value, for example in the display frame list dialog. Other dialogs behave in a very similar manner to this one though the precise options available may vary, the description below only mostly matches the display frame list setup.



You can use the upper Frames selector to choose from All frames, Current frame, Buffer, Tagged frames, Untagged frames, Frames state = xxx (with a separate field for entering the state value) and Last n frames (again with an extra field for entering n). When used online an additional field All sampled frames is available and the text shown for All frames changes to All filed frames. In addition to these options you can also directly enter frame numbers or a frame list such as 1..50,60,61,70..80.

Direct text entry can also be used to specify wanted or unwanted states; ST:<list> will select all frames whose state value is in the list, while !ST:<list> selects all frames whose state is not in the list. For example ST:1..8 will select all frames with state values from 1 to 8, while !ST:3,6 selects all frames whose state value is not 3 or 6. So this main selector already gives you a wide range of possibilities.

When the main Frames selector is set to All frames, Tagged frames, Untagged frames or a numerical frame list the Frame subset selector is shown. This allows you to select an extra criterion such as Frame state = xxx to apply to get the frames that are wanted. Using the two selectors together allows selection of, for example, all untagged frames with a certain state code, giving even more possibilities. These mechanisms are also available in the Signal script language where you can also build a frame list directly in an script array to get complete control.

# Dialog expressions

Many dialogs (and the equivalent script functions) in Signal accept an expression string in place of a number. These expressions can be divided into two types: *numeric expressions* and *view-based expressions*.

### Numeric expressions

A numeric expression is composed of numbers, the arithmetic operators +, -, * and /, the logical operators <, <=, =, >=, <> and ? and round brackets ( and ). The result of a logical comparison is 1 if the result is true and 0 if the result is false. You may not have come across ? which is used as:

```
expr1 ? expr2 : expr3
```

The symbols expr1, expr2 and expr3 stand for numerical expressions. The result is expr2 if expr1 evaluates to a non-zero value and expr3 if expr1 evaluates to zero. This can be used in the Cursor mode dialog to give a cursor position if a search fails based on some other information, for example:

```
Cursor(2)>Cursor(1) ? Cursor(2) : Cursor(1)
```

This evaluates to the position of the rightmost of cursors 1 and 2.

If you write expressions involving more than one operator, for example 1+2*3 you need to know if this is evaluated as (1+2)*3 or as 1+(2*3). This is determined by the operator precedence level which is described below.

### View-based expressions

These expressions follow the rules for numeric expressions and allow references to positions along the x axis or a channel Y axis. If a dialog field is documented as allowing expressions and the field supplies an x axis position (for example a time), then you can use the following in addition to all of the operators and options available in numeric expressions:

| | |
|---|---|
| Cursor(n) | Where n is 0 to 9, this returns the position of the relevant cursor in the view. If the cursor does not exist or the position is invalid, the expression evaluation fails. |
| C0 to C9 | This is shorthand for Cursor(0) to Cursor(9). |
| CursorX(n) | Where n is 0 to 9 returns the position of the cursor before the move to the current position. If the cursor does not exist the expression evaluation fails. This was added at Signal version 6.05. |
| C0X to C9X | This is shorthand for CursorX(0) to CursorX(9), and returns the previous position of cursors 0 through 9. This was added at Signal version 6.05. |
| XLow() | The left hand end of the view's visible x axis in x axis units. |
| XHigh() | The right hand end of the view's visible x axis. |
| MinTime() | The start time or x axis value for the frame. |
| MaxTime() | The end time or x axis value of the frame. |
| AbsTime() | The frame start time; the time relative to the start of sampling of the trigger time (generally zero on the X axis) for the frame. |
| F0 | This is shorthand for AbsTime(). |

### View-based channel expressions

If a dialog field is documented as allowing expressions and the field supplies a y axis position that is associated with a specified channel, then you can use the following. In addition, some expression fields that most often supply an X axis position can also make use of these:

| | |
|---|---|
| HCursor(n) | Where n is 1 to 9 returns the position of the relevant horizontal cursor in the view. If the cursor does not exist or is not on the relevant channel, the expression evaluation fails. |
| H1 to H9 | This is shorthand for HCursor(1) to HCursor(9). |
| HCursorX(n) | Where n is 0 to 9 returns the position of the horizontal cursor before the move to the current position. If the cursor does not exist the expression evaluation fails. This was added at Signal version 6.05. |
| H1X to H9X | This is shorthand for HCursorX(1) to HCursorX(9), and returns the previous position of horizontal cursors 1 through 9. This was added at Signal version 6.05. |
| YLow() | The bottom end of the view's visible y axis for the relevant channel. |

| | |
|---|---|
| `YHigh()` | The top end of the view's visible y axis for the relevant channel. |
| `At(t{, c})` | The value on the relevant channel at time t, which can be a view-based time expression. If the optional channel number c is supplied the measurement is taken from that channel rather than the channel associated with the dialog field. |
| `Mean(t1, t2{, c})` | The mean level on the relevant channel between times t1 and t2, both or either of which can be view-based time expressions. If the optional channel number c is supplied the measurement is taken from that channel rather than the channel associated with the dialog field. |
| `SD(t1, t2{, c})` | The standard deviation of the values on the relevant channel between times t1 and t2, both or either of which can be view-based time expressions. If the optional channel number c is supplied the measurement is taken from that channel rather than the channel associated with the dialog field. |
| `Meas(m, t1, t2{, c})` | Any available measurement (selected by m) made on the relevant channel between times t1 and t2, both or either of which can be view-based time expressions. If the optional channel number c is supplied the measurement is taken from that channel rather than the channel associated with the dialog field. See the `ChanMeasure()` script function for a list of the possible values of m and the corresponding measurements. |

You can add `View(-n).` before these expressions (apart from those referring to frame information) to force the expression to be evaluated for the n-th duplicate of the current data view, for example `View(-1).Cursor(0)` forces the use of cursor 0's position in duplicate number 1 of the current view. This is required when the current view is duplicated and you need to be sure which duplicate should be used.

## Times as numbers

Times are normally entered as a simple number specifying the preferred X axis units, as set in the Preferences dialog Display page. However, where a time is typed into a dialog field with a drop-down for preset strings such as XLow() you can use {{{days:}hours:}minutes:}seconds where the seconds may include a decimal point and items enclosed in curly brackets are optional. Each colon promotes the number to the left of the colon from seconds to minutes to hours to days. Times may only contain numbers and colons, white space is not allowed. One decimal point is allowed at the end of the time to introduce fractional values. We also allow a number with no colons to be followed by s, ms or us to force Signal to interpret the time entered as a value in seconds, milliseconds or microseconds. So the following are all equivalent: 1.6, 1.6s, 00:00:01.6, 1600ms, 1600000us.

## Operator precedence

In the table, `LHS` means the value of the expression to the left of the operator as far as the next operator of same or lower precedence, `RHS` means the value of the expression on the right up to the next operator of the same or lower precedence. Where operators have the same level, evaluation is from left to right. The order from high to low is:

| Level | | Name | Return value |
|---|---|---|---|
| 5 | () | Brackets | Everything inside a pair of brackets is evaluated before considering the effect of an adjacent operator. |
| 4 | *<br>/ | Multiply<br>Divide | LHS multiplied by RHS<br>LHS divided by RHS. It is an error for RHS to be zero |
| 3 | +<br>– | Add<br>Subtract | LHS plus RHS<br>LHS minus RHS |
| 2 | <<br><=<br>=<br>>=<br>> | Less than<br>Less or equal<br>Equal<br>Greater or equal<br>Greater than | If LHS less than RHS then 1 else 0<br>If LHS less than or equal to RHS then 1 else 0<br>If LHS equal to RHS then 1 else 0<br>If LHS greater than or equal to RHS then 1 else 0<br>If LHS greater than RHS then 1 else 0 |
| 1 | ? | Ternary operator | LHS?A : B  has the value A if LHS is not 0, and B is it is 0. Put spaces around the colon to distinguish it from a time. |

1+2*3 has the value 7 because multiply has a higher precedence level than add.

### Script language compatibility

The expressions are compatible with the script language except for use of `C0` to `C9`, `H1` to `H9` and `F0` as shorthand for `Cursor(0)` to `Cursor(9)`, `HCursor(1)` to `HCursor(9)` and `AbsTime()` and the use of colons, `ms` and `us` to denote times. If you use these in a script you will get syntax errors. However, you can use these constructs in strings passed as expressions to `CursorActive()` or `MeasureToXY()`.

# Data view keyboard shortcuts

The following shortcut key combinations can generally be used in file, memory or XY views, except where otherwise specified.

| Key | Operation |
|---|---|
| Left arrow | Scroll display left. |
| Right arrow | Scroll display right. |
| Ctrl+Left | Decrease X axis range. |
| Ctrl+Right | Increase X axis range. |
| Ctrl+Home | Show all X range. |
| Ctrl+X | Provide the X Axis range dialog. |
| Down arrow | Shift data down (scroll Y axis) on all selected or visible channels. |
| Up arrow | Shift data up (scroll Y axis) on all selected or visible channels. |
| Ctrl+Down | Increase Y range (data shown smaller) on all selected or visible channels. |
| Ctrl+Up | Decrease Y range (data shown bigger) on all selected or visible channels. |
| Home | Show all Y range on all selected or visible channels. |
| End | Optimise Y range on all selected or visible channels. |
| Ctrl+Y | Provide the Y Axis range dialog. |
| Double-click | Zoom or un-zoom double-clicked channel (file and memory views only). |
| Del | Hide the selected channels (file and memory views only). |
| Ctrl+Del | Provide the customise display dialog. |
| Ctrl+B | Toggle displaying the frame buffer (file and memory views only). |
| Ctrl+D | Toggle frame display list overdrawing (file and memory views only). |
| Ctrl+Shift+D | Provide the frame display list dialog (file and memory views only). |
| Ctrl+n | Where n is 0 to 9. Fetch vertical cursor 0 to 9. If the cursor does not exist it is created. Cursor 0 exists only in file and memory views. |
| Ctrl+Shift+ Left/Right | If cursor 0 is active, search for the next/previous feature and scroll the screen to make it visible (file and memory views only). |
| PgUp | Next frame (file and memory views only). |
| PgDn | Previous frame (file and memory views only). |
| Ctrl+PgUp | Last frame (file and memory views only). |
| Ctrl+PgDn | First frame (file and memory views only). |
| Ctrl+A | Open the experimenter's notebook (file and memory views only). |
| Ctrl+I | Open the File information dialog (file and memory views only). |
| Ctrl+G | Provide the Goto frame dialog (file and memory views only). |
| Ctrl+C | Copy the image of the view and data as text to the clipboard. |
| Ctrl+N | Open a new view. |
| Ctrl+O | Open the file open dialog. |
| Ctrl+E | Export data (file and memory views only). |
| Ctrl+P | Print the current data file. |
| Ctrl+L | Open the Evaluate window to run single line script commands. |
| Ctrl+T | Toggle the frame tag (file and memory views only). |
| Ctrl+Z | Undo the last undoable operation. |

`Ctrl+Break`      Break out of long drawing or calculation operations.

There are also a large number of keyboard shortcuts for various analysis and data manipulation operations for file and memory views only. These are documented separately at the end of the Analysis menu chapter. There are also a number of shortcuts for control of sampling.

# Text view topics

# Text view features

# Drag and drop

The editor supports drag and drop of text both within Signal and between Signal and other applications that support it (for example the Signal Help system). Signal also supports drag and drop for rectangular text areas.

| Operation | Method |
| --- | --- |
| Move block | Select the text to move. Move the mouse pointer over the selected text and hold down the left mouse button and drag. The mouse pointer will indicate that you can now drag the text and the text caret will show the insertion point. Drag the text to the desired insertion point and release. |
| Copy block | Select the text to copy. Hold down the `Ctrl` key and move the mouse pointer over the selected text, click and drag. A small + symbol indicates the copy operation and the text caret will show the insertion point for the duplicate. Drag the text to the target position and release the mouse button to duplicate the text. The `Ctrl` key must be down when you release the mouse button or the operation will move the text. |

# Virtual space

Prior to Signal version 4.06, the text caret could only be positioned between or next to existing characters in a text line. From version 4.06 onwards, you can position the caret beyond the end of the text in a line by clicking in a blank area with the mouse, or using the cursor right key. You cannot position the text caret below the last line of text. When the caret is beyond the end of the line, it is said to be in *virtual* space. If you type with the caret in virtual space, space characters will be added to fill in the virtual space up to the text you type.

There are two main uses for virtual space: to add comments without having to space along to the required column, to make rectangular selections without having strange visual effects due to short lines.

If you use the script language to manipulate the text caret and make selections, virtual space is ignored; a caret in virtual space will be treated as if it is at the end of the line. There are no script commands that will move the caret into virtual space.  If there is a requirement for the script to report or use virtual space, we will extend the script in a compatible way to incorporate it.

# Multiple selections

From Signal version 4.06 onwards, you can make multiple selections in a text view. To do this, hold down the `Ctrl` key and click and drag. Each time you make a new selection in this way it becomes the current selection; all previous selections are shown with a different selection colour. When you have a multiple selection, each selected area has a flashing text caret at the insertion point. If you type characters, these will appear at all insertion points, replacing all the selected text. If you use the delete or backspace key, all selected text will vanish. Note that pasting into a multiple selection will clear all the selected text, then insert the pasted text at the current (last made) selection.

Multiple selection can be useful when you want to move several non-consecutive script functions to make them consecutive.

# Select rectangular text area

You can select, cut, paste and drag rectangular selections within Signal. To select a rectangular area hold down the `Alt` key then select text with the mouse. The point where you hold down the mouse button will be one corner of the selection, the point where you release the mouse will be the other corner. You can use this feature to change the alignment of comments in a script, or to convert a single column of numbers into multiple columns. You can also paste such text into other applications as plain text.

A rectangular selection will paste within Signal as a rectangular selection. Beware that `Ctrl+X` (cut) on a rectangular selection followed by `Ctrl+V` (paste) will not leave the text unchanged (unless you select the text from bottom to top). The cut operation will leave a vertical flashing line (assuming a fixed pitch font) with the insertion point marked by a more visually obvious caret. The paste operation will paste the cut text as a rectangular block at the insertion point.

A rectangular selection is a multiple selection.

You can also easily select whole lines of text by holding the mouse pointer just off the edge of the margin on the left of the text view. The mouse pointer changes to a rightwards-leaning arrow when you are in the line selection area. A selection that is a number of whole lines of text can be moved up and down by using `Alt+Up` and `Alt+Down`.

# Read only text

If you open an output sequence or script file that is held in a read only file, the text window will also be marked read only and you will not be allowed to change it. This is to allow users to protect sequences and script from inadvertent change. If you want to edit such a text file you have two options:

1) Close the file, remove the read only status on disk, then open it

2) Save the file to a new file with a different name

You can use the `Modified()` script command to get and change the read only status of a text file. However, if the original disk file is marked read only, using `Modified()` to allow you to edit the text will not change the status of the disk file, so you will not be able to save it to the same file name.

# Text view keyboard shortcuts

Text views have more keyboard short cuts than any other area of Signal. We have grouped them by function to make the huge list more digestible. There are more keyboard shortcuts for data view manipulation, for analysis, and for control of sampling.

# Cut, Copy, Paste, Delete, Undo and Redo

Some of these operations are also available from the Edit menu and the main toolbar.

| Key | Operation |
|---|---|
| `Ctrl+A` | Select all the text in the document. |
| `Ctrl+C` `Ctrl+Insert` | Copy selected text to the clipboard. If no text is selected, nothing is copied. Some keyboards have `Ins` in place of `Insert`. |
| `Ctrl+Shift+T` | Copy the current line to the clipboard. |
| `Ctrl+V` `Shift+Insert` | Paste the contents of the clipboard into the text at the caret. If there is a selection, the selection is replaced. |
| `Ctrl+D` | Duplicate the selection. |
| `Ctrl+X` | Cut the selected text and copy it to the clipboard. |

| | |
|---|---|
| `Shift+Del` | Cut the selected text and copy it to the clipboard. |
| `BackSpace` | Delete the selection or the character to the left of the text caret. |
| `Del` | Delete the selection or the character to the right of the text caret. |
| `Ctrl+Del` | Delete word right. Add `Shift` to delete to the end of the line. |
| `Ctrl+D` | Duplicate the selection. |
| `Ctrl+Shift+L` | Delete the current line. |
| `Ctrl+Z` `Alt+Backspace` | Undo the last interactive text operation. The editor supports more or less unlimited levels of Undo. |
| `Shift+Ctrl+Z` | Redo the immediately previous Undo operation. |

# Find, Replace and Bookmarks

The Find and Replace commands can be accessed from the Edit menu, from the Edit Toolbar and by keyboard short cuts:

| Key | Operation |
|---|---|
| `Ctrl+F` | Open the Edit menu Find dialog. In addition to searching for text you can also use this dialog to bookmark all matching text. |
| `Ctrl+H` | This shortcut key opens the Edit menu Replace dialog. |
| `F3` | Repeat the last find operation in the same direction. You can use the toolbar to search forwards or backwards. |
| `F2` | Move the text caret to the next bookmark. You can use the edit toolbar to move to the next or previous bookmark. |
| `Ctrl+F2` | Toggle bookmark on the current line. You can use the edit toolbar to set or clear a bookmark and to clear all bookmarks. |

Bookmarks tag a line for future reference. They are displayed as a blue mark to the left of the text. Bookmarks are kept as long as the current file is open; they are lost when you close the file. The easiest way to use a bookmark is from the Edit Toolbar. You can show and hide this from the Edit menu (when a text-based window is active), or by clicking the right mouse button on any toolbar or on the Signal application title bar and using the pop-up context menu that appears.

# Indent and Outdent

The structure of Signal scripts and other text files can often be made clearer by indenting program structures. To make this easier, you can indent and outdent selected blocks of text to the next or previous tab stops. The tab size is set in the Edit menu Preferences Display page for script, sequencer and text files.

| Key | Operation |
|---|---|
| `Tab` | If there is a multi-line selection, all lines included in the selection are indented so that the first non-white space character is at the next tab stop. If there is no selection, a tab character is inserted (or spaces to the next tab stop depending on the Edit menu Preferences settings). |
| `Shift+Tab` | If there is a multi-line selection, all selected lines are out-dented so that the first non-white space character on the line is at the previous tab stop. If there is no selection, the text caret moves to the previous tab stop unless it is already at one. |

# Text caret control

The text caret is a flashing vertical bar that indicates the current position. Do not confuse this with the I-beam mouse pointer which does not flash and which indicates the mouse position. Each time you click and release the left mouse button (we assume you haven't swapped the mouse buttons), the caret moves to the nearest character position to the click point. To select text with the mouse, click at one end of the text you want to select and drag (move the mouse with the button held down) to the other end of the text. You can also use the keyboard to move the caret and select text:

| Key | Operation (`+Shift` to extend a text selection, `+Alt+Shift` for rectangular) |
|---|---|
| `Left arrow` | Move the caret one character to the left. At the start of a line it wraps to the end of the previous line. |
| `Right arrow` | Move the text caret one character right. You can move it into uncharted territory beyond the end of the line. It does not wrap to the next line. |
| `Up arrow` | Move up one line. |
| `Down arrow` | Move down one line. |
| `Ctrl+Left` `Ctrl+Right` | Move one word to the left/right. This means move the caret to the next boundary between a word character and a non-word character. Word characters are defined to be useful and vary depending on the view type. In a script view, they are `A-Z`, `a-z`, `0-9`, `_`, `$` and `%`. In an output sequence, they are `A-Z`, `a-z` and `0-9`. All other text views use `A-Z`, `a-z`, `0-9` and `_` as word characters. |
| `Ctrl+Up/Down` | Scroll the text window up and down, leaving the selection unchanged. |
| `Alt+Up/Down` | Move the lines containing the selection up or down by one line. |
| `End` | Move the caret to the right of the last character on the line. |
| `Home` | Move the text caret to the start of the current line. |
| `Alt+Home` | Move the text caret to the start of the current line. |
| `Ctrl+End` | Move the text caret to the right of the last character in the file. |
| `Ctrl+Home` | Move the text caret to the left of the first character in the file. |
| `Ctrl+]` (`[`) | Start of next (previous) paragraph (after empty line) |
| `Ctrl+\` (`/`) | Word part right (left). |
| `PgUp` | Move upwards by one window of lines. |
| `PgDn` | Move downwards by one page of lines. |
| `Insert` | Swap between insert mode caret \| and over-type caret _ |

# Miscellaneous

These commands do not fit into any other category!

| Key | Operation |
|---|---|
| `Ctrl+U` | Convert the selection to upper case. Add `Shift` for lower case |
| `Ctrl+Add,Sub` | Change font size (`Add` and `Sub` are numeric keypad + and − ). You can also change the displayed text size by holding down the `Ctrl` key and using the scroll wheel on your mouse (if you have one). The **View** menu **Standard Display** command will remove any zooming. The script equivalent is `ViewZoom()`. |

# Grid view topics

You can create a Grid view interactively with the File menu New command or from a script with the `FileNew(17, mode%, rows%, cols%);` command.

You can save and restore grid views with `.sggx` files (these are XML files internally). Grids support cut and paste of text using a tab character `"\t"` to separate columns and an end of line character `"\n"` to separate rows. There is also the `GrdXxx...()` family of script commands to support the grid.

## What the Grid is and what it is not

Our first implementation of a Grid is not attempting to be a spreadsheet program. It is implemented as a utility to make it relatively easy to generate tabulated data. The initial implementation stores data as text, but we may extend this to allow data to be stored as integers, or real numbers with some user-control of formatting. There is no capability yet in the grid to refer to other cells or to allow expressions to be evaluated.
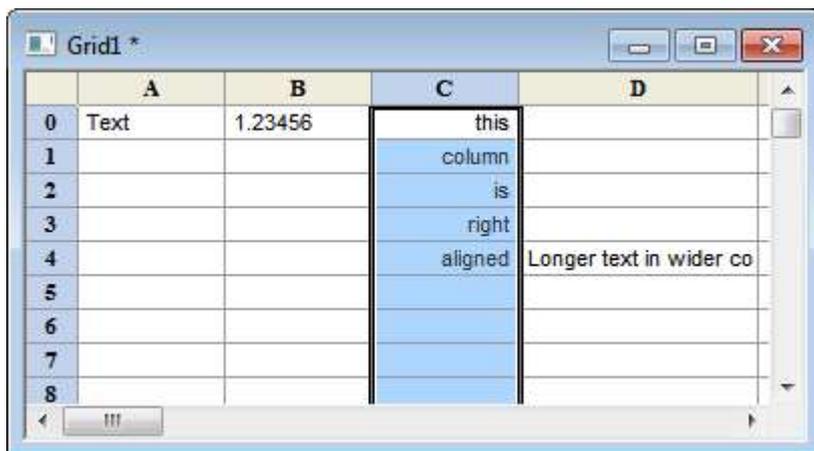


## Grid properties

The underlying grid implementation is capable of giving each cell its own font, alignment and type. To give us a chance of implementing the grid in a reasonable time, we have restricted the grid as follows:

### Size

The grid is currently restricted to 10000 rows and 1000 columns. We may change this to a limit based on the product of rows and columns (so you can have more or one at the expense of the other). Rows and columns are numbered from 0,0 at the top left. There is a default size for grids created interactively; from a script you can create a grid with a defined size with `FileNew(17, ...)` and you can resize it with `GrdSize()`.

### Headers

The grid has one header row at the top and one header column on the left. You can choose to hide or show these headers. By default the header column displays the 0-based row number and the column header displays columns as letters, A-Z for columns 0 to 25, AA-AZ for columns 26 to 51 and so on.

### Font

All cells in the body of the grid use the same font. The headers use a bolder version of the same font.

### Row and column spacing

For simplicity, all rows are the same height, which is based on the font. Columns can be sized individually, or you can set a standard size, or you can optimise the columns widths to match the data displayed in them.

### Alignment

You can apply Left, Centred or Right alignment on a column by column basis.

# Information window topics

The information window is a specialised type of view which can display times and special information using a very large font for extra visibility. Information windows are attached to a time, memory or XY view and when saved to disk these views will also save the information needed to re-create any associated information windows when the file is re-opened.

You can create an information window interactively with the View menu New Info window command or from a script with the `InfoNew(mode%)` function, other script functions can be used to manipulate information windows. An information window display is set by a string which includes special fields that are replaced by other information (for example, the current frame number).
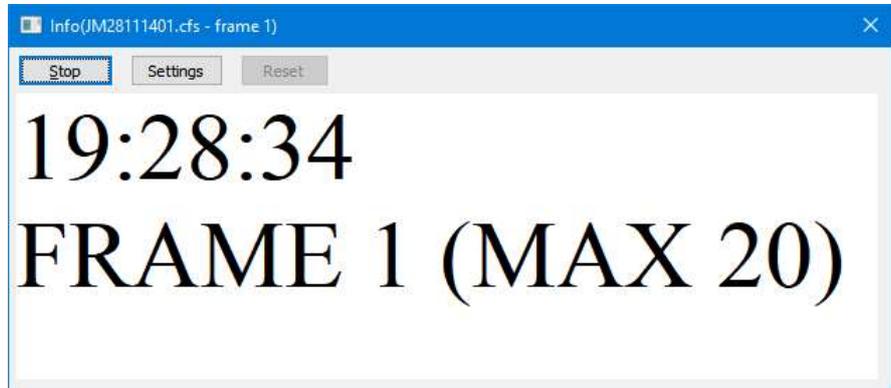
Information windows can also display an image file instead of the standard background and either the area containing the buttons or the window title area (or both) can be hidden if desired. These and other options are available via a menu generated when you right-click on the information window.

# Copy current file path to clipboard

When working with file views (data, result, XY or text-based) you can copy the full path to the file (if the view has an associated file) by right clicking on the view title bar and then selecting the Copy path item that appears. This is especially useful if you have set the operating system to not display full path names.

# The Signal command line

When Signal starts, it checks the command line for option switches and for files to load. If there is no command line, Signal looks in the folder that it ran from for a script called startup.sgs and, if it exists runs it. If startup.sgs is found and run or if the command line loads a file, any start up messages that wait for a user response are suppressed. The items in the command line used to run Signal are also available to running scripts by using the System$() script function.

The command line holds option switches and file names, separated by white space characters (space and tab). If a file name contains spaces, you must surround the file name with quotation marks. Option switches start with a / or – character followed by a character to identify the option. The switches understood by Signal are:

/M  When you start Signal, it checks if there is already a running copy. If there is, the new one quits. This option removes the check, allowing multiple copies to run on a single system. You need a Signal license for each copy except when using multiple synchronised 1401s to capture related data on one computer under the control of a single operator, when one license is sufficient. To do this, you must set separate file names for each 1401 in the Automation tab of the Sampling Configuration dialog.

/Un  n is 1-8 to select a 1401 when you have more than 1. The default is /U0, which uses the lowest-numbered unused 1401. You set a device number in the CED 1401 device settings in the Device Manager (My Computer->Properties->Hardware->Device Manager).

/Q  Quiet startup. Suppress all message boxes and the Signal "splash screen".

The remaining items in the command line are assumed to be file names. Signal attempts to load the files in command line order (from left to right). The files must have extensions so that the file type is known. If a script file is included in the command line, Signal runs it before continuing with the remainder of the command line.

As an example, suppose we want to launch Signal so that it automatically opens a data file called example.cfs and runs doit.sgs to process it. Follow these steps:

1.    Create a short cut to cfsview.exe (this is the Signal program).

2.    Right-click on the new short cut and select Properties and open the Shortcut tab.

3.    Add example.cfs doit.sgs to the end of the Target field.

4.    Set the Start in field to the folder that contains your files.

5.    Click OK.

This example assumes that both files are in the same folder. You could also have included the full path to each file in the command line.

# 64-bit operating systems

Before Signal version 6, all versions of Signal were 32-bit programs, even if the operating system was 64-bit. From Signal version 6, by default, we will install a 64-bit version of the program on 64-operating systems and a 32-bit version on 32-bit operating systems. The advantages of 64-bit code on a 64-bit operating system are:

1. The 64-bit version of the CPU has more registers and instructions; "typical code" runs maybe 10% faster.

2. Calls into the operating system do not transition from 32-bit Signal code to 64-bit system code and back.

3. A 32-bit program is limited to a 4 GB memory space, and of this, quite a bit (typically 1GB) is reserved for the 32-bit operating system. With a 64-bit program you can use a lot more memory.

There are some disadvantages, too:

1. 64-bit code is typically larger (for example the cfsview.exe file is 20% larger as 64-bit code).

2. Pointers in memory are double the size, so memory usage increases.

Unless your system has limited installed memory, we recommend that you use the 64-bit version of Signal on a 64-bit operating system.

# Installation and data directories

This section applies to Signal version 6 and 7, so we refer to installation directories as `SignalN` where `N` stands for 6 or `7` as appropriate.

Note:    The example file paths in this section relate to English language versions of Windows. `Program Files` can have different names depending on the language set for the system. Similarly, `My Documents` can be different in different languages.

Previously, we installed Signal by default in a top level folder such as `C:\SignalN`. While this practice was satisfactory when used with earlier versions of Windows, it has become deprecated, mainly for security reasons. We have altered Signal so that it can be installed in the protected `C:\Program Files` folder as expected by Microsoft. Some users will prefer to keep their existing arrangements so we still support installation into `C:\SignalN` or similar. This support will be removed at some future point.

The main change caused by installing inside `C:\Program Files` is that this location is expected to be read-only as seen by users. While there is no problem in storing data files, sampling configurations, scripts and other files in `C:\SignalN`, you are not allowed to modify anything inside `C:\Program Files`. Therefore as part of the Signal installation new folders are created for the current user's data, for data shared between all users and for data generated by the application.

## New folders created by the installer

### *User data directories*
User data directories are used to hold data that is directly saved or loaded by the user, for example data files, sampling configurations and scripts. Two such directories are created by the Signal installer:

- Data for the current user (the user who installed Signal). This directory is called `SignalN` and is created inside the `My Documents` folder for the current user. Windows has separate `My Documents` folders for each user of a computer system, this directory is only created inside `My Documents` for the current user. When installing inside `C:\Program Files` the Signal installer puts all data intended for the user's own use inside this directory, the folder contains the `Data`, `Scripts`, `Sequence`, `ExtraDoc` and `TrainDay`

folders plus all of their contents, in addition an empty Include directory is provided to hold your own script include files. This folder can also be used to hold your own files though of course you can use any directory you choose for this.

- Data for all users of this computer. This directory is called `SignalNShared` and is created inside the `Documents` folder provided for all users of the system. The installer does not put any data inside this directory but it does create an empty `Include` directory inside it to hold any shared script include files that should be generally available. If there are multiple Signal users on the system this directory ensures that there is always a suitable directory for user's data.

When searching for the user data directory, Signal first looks for the current user's data directory and if this is not found will use the user data directory for all users. If neither of these directories can be found Signal uses the current user's `My Documents` folder as a last resort. If there are other Signal users on the system apart from the one that installed Signal they can create their own `SignalN` directory inside their `My Documents` folder if they want, or they can use their own data directory somewhere or make use of the shared directory for all users.

*Application data directories*

Application data directories are used to store data that is generated by the application without any direct user involvement, for example the filter bank settings and the default and last-used sampling configurations. In the same manner as the user data directories, two application data directories are created by the Signal installer:

- Specific to the current user (the user who installed Signal). This folder location varies with the operating system version but in Windows7 the folder is `C:\Users\Username\AppData\Local\CED\SignalN`.

- Shared between all users. This folder location varies with the operating system version but in Windows7 the folder is `C:\ProgramData\CED\SignalN`. If there are multiple Signal users on the system this directory ensures that there is always a suitable location available for application data.

When searching for the application data directory, Signal first looks for the current user's application data directory and if this is not found will use the application data directory for all users. If neither of these directories can be found Signal uses the current user's `My Documents` folder as a last resort. If there are other Signal users on the system apart from the one that installed Signal they should create their own `CED\SignalN` directory inside their own `AppData\Local` folder unless they are happy for Signal to use the shared application data directory for all users.

## Moving your files to a new-style location

After an installation inside C:\Program Files, Signal and the data files that are installed with Signal will be in their new locations but any files in the original C:\SignalN directory (or whatever has been previously used) will be left unchanged. This applies to both your own files and the previous installation of Signal. There is no reason why you have to do anything much about this - you can continue to load sampling configurations from and save data files into C:\SignalN just as before and many users may prefer to do this, perhaps after cleaning it up (see below). Alternatively you can choose to keep your data (and other files) in the new location for user data files. This can be done by moving all .cfs, .sxy, .sgr, .sgrx, .sgc, .sgcx, .sgs and .pls files (plus any other files you might want to keep: .txt? .jpg?) that are inside C:\SignalN plus any folders of your own into the `SignalN` directory inside your `My Documents` folder. Once you do that you will be able to use your data files, scripts, sampling configurations and other files from the new location. There are a small number of possible issues involved in moving these files (see below for a list of these) but generally things will Just Work. After moving all of your data to a new location you should check the following:

*Application-level issues*: While the Signal application will run in its new location without any problems, there may be a number of minor issues relating to moving your files or installation into the new location:

- The Sampling page of the Preferences dialog contains the path to a directory which is used to store the temporary CFS data file which is created by sampling. Previously this was set by default to be the directory in which Signal was installed, so you will probably see that it is something like `C:\SignalN`. While there is no problem in leaving this setting unchanged even if you have deleted `C:\SignalN`, if you want you can get Signal to set it to the application data directory by clearing the path in the preferences dialog and pressing OK. You should avoid setting this to a directory inside `C:\Program Files` as this might cause problems.

- Global resources - some options for the global resource file location use the directory where Signal is installed. While this location can still be used you will have problems if you try to make use of a directory inside `C:\Program Files`, so Signal now uses a list of directories which are searched in turn for the required file. The list starts with the current user's data directory inside `My Documents` - if you are using

global resource files inside the directory where Signal is installation you should move the resource files to this location or somewhere else suitable and adjust the global resource setting to match.

- The Sample and Script bars hold lists of sampling configurations and script files. These lists hold the complete path to the file so if you use either of these you should update them so that they use the new file locations.

*Data and XY file issuess*: There should be no problems with the data files themselves as they do not contain any information relating to directory locations. However if a background image file has been set up for a data or XY file the background image settings will contain a complete path to the image file which may need to be updated.

*Sampling configurations*: Sampling configuration files can contain a number of directory paths which may need to be adjusted. Note that (as documented just below) Signal will only write sampling configuration files using the new XML file format and `.sgcx` file extension so any sampling configurations you change will have to be converted to the new format (by using them for sampling before) they can be saved. As changed sampling configurations will use the new file name extension your old configuration files will remain unchanged; if you want an original sampling configuration you should use the `.sgc` file while if you want updated configurations you should use the `.sgcx` file. The things you might have to change are:

- Sequence file.Where the sampling configuration Outputs page specifies a sequence file to be used during sampling, the entire path to the sequence file is stored and will point to the previous directory for the `.pls` sequence file, this should be changed to point to the new file location.

- Automatic file-naming. Where a sampling configuration Automation page specifies a directory for automatically-named files this path might now point at a non-existent directory and should be suitably adjusted.

- Dynamic clamp model tables in text files. Some dynamic clamp models use user-generated text files to customise their behaviour. The model settings include the text file name which can optionally include a directory path. The location of these files are normally relative to the sampling configuration location, in which case Signal will be able to find the files but any absolute text file paths (paths starting with a drive letter and colon) should be adjusted to point to the new file locations.

*Sequence files*: Apart from the references to sequence file names in sampling configurations that are described above, there should be no problems with these and no changes to these files will be required.

*Script files*: Depending upon how they are written and what they do, script files may or may not have problems. Any problems that do occur will probably be caused by file path string constants in the scripts which may need adjusting and by the use of the `FilePath$()` function. Any use of `FilePath$(2)` to get the directory where Signal is installed is likely to give problems as you should not create or write to files inside Program Files and you should adjust the script so that it uses a different location. The parameters to `FilePath$()` have been adjusted to allow you to find the current user data folder - we suggest that you change to using this location.

## Cleaning up the old installation location with or without moving your own files

If after installing Signal into a new location inside C:\Program Files, you want to leave your files in the old install location but get rid of the previously installed Signal application you should do the following:

- Delete the `1401`, `BaseDemo`, `Export`, `import` and `include` folders inside the old install location. If you have not put any of your own files into them, you can also remove the `data`, `ExtraDoc`, `scripts`, `sequence` and `trainday` folders.

- Inside the old Signal installation folder itself, delete all `.exe`, `.dll`, `.chm`, `.sgl`, `.t14` and `.tip` files. All of these are files specific to the Signal installation and will have been replaced by newly installed files in the new installation location.

# Unicode

Before Signal version 6.03, all text in Signal was represented by 8-bit characters with codes in the range 0 to 255 (hexadecimal 0x00 to 0xff). In the 8-bit character world, characters with codes 0 through 127 are pretty much universal, for example character 32 (0x20) is a space anywhere on the planet. These are sometimes referred to as the ASCII character set. However, character codes 128 to 255 (0x80 to 0xff) have meanings that depended on what code page your computer is set to. As some cultures have many more characters than can fit in the 255 available, in some cases pairs of 8-bit characters are used to represent characters. The result of this was that you could use characters with codes above 127 to represent national characters in text views, but if you sent a data file or text file

that used such characters to someone with a computer set to a different code page, the result was unpredictable (and often gibberish).

The solution to this problem is to use a character encoding that does not limit us to 8-bit characters. The Unicode Consortium has defined a standard set of characters (1,112,064 valid code points) that can encompass all the cultures on the planet. The characters are grouped into 17 'planes' of 65536 characters, of which the most important is the first, known as the Basic Multilingual Plane (BMP). Character codes 0-127 of the BMP have the same meanings as the ASCII codes 0-127, which is very convenient for backwards compatibility. The BMP holds the vast majority of the characters used in everyday communication. The remaining planes hold more infrequently-used and specialist characters, such a music clefs, historic scripts (hieroglyphs), playing card symbols and the like.

There are a variety of ways of storing Unicode characters in memory and in data files. Only two need concern us: UTF-8 and UTF-16LE.

## UTF-8

This is a method of storing Unicode characters using a sequence of 8-bit characters. It has the property that if your text only uses character codes 0 to 127 (0x00-0x7f), then it is identical to ASCII text. So any Signal script that you have that does not use any characters with codes above 127 is already coded in UTF-8. Character codes in the range 128 to 2047 (0x80-0x7ff) are coded in two bytes, character codes in the range 2048-65535 (0x800-0xffff) need three bytes and character codes greater than that take 4 bytes. If your text is mainly ASCII, or uses only European characters, this is the most compact coding. The encoding is arranged so that you can always tell if a particular byte is the start of a character or the continuation of a character.

We have standardised on UTF-8 as the encoding we will use for all external files (scripts, output sequences, XML resources, text in Signal data files). The text editor within Signal uses UTF-8 internally (though this is of no consequence to most users). UTF-8 is probably the most common format used for storing Unicode in files; in October 2014 it accounted for 81.6% (with an upward trend of 4% over the year) of all web pages with a known encoding. The next most common web format was ISO-8859-1 (which is an 8-bit character set covering European languages) with 9.8%, down about 2% over the year.

## UTF-16LE

This uses 16-bit Little Endian values to store the characters. This is the format that Windows uses internally for all text. Unicode builds of Signal uses this format internally everywhere except where we have to use 8-bit characters (1401 communications, serial line input and output, external text files, resource files, the script editor, data import, MATLAB interface). In UTF-16LE, characters in the BMP are all represented by one 16-bit value. Characters in all other planes require a pair of 16-bit values. Character codes in the range 0xd800 to 0xdbff are reserved for the first (lead) of a two word pair and characters in the range 0xdc00 to 0xdfff are reserved for the second (trail) part of a two word pair. These are known as surrogate pairs.

### Updating to a Unicode version of Signal

Signal version 6.03 and onwards uses Unicode characters internally and stores output text in UTF-8 compatible files. When you update from an ASCII (8-bit character) to a Unicode version of Signal our aim is that you should see no difference. We achieve this by assuming that any characters that we read above code 127 that are not valid UTF-8 sequences are code page characters, and we convert text appropriately:

- When we open a script or sequencer or text file in a text editor, we check read file to attempt to guess the format. If it starts with a BOM (Byte Order Mark: a special code that indicates the file format) for UTF-8 or UTF-16LE, we assume that is the format. Otherwise, if it holds no characters above code 127 we can already treat it as Unicode. If it holds correctly formed UTF-8 (this is easy to detect), we assume it is UFT-8. If the text is an even number of bytes long and holds 0 bytes in odd positions, we assume it is UTF-16LE. If it is none of these, it must hold 8-bit characters above code 127 and we assume these are code page characters and we translate the text using the local code page set on the computer. It is possible that we might be fooled into interpreting code-page text as UTF-8, but this is unlikely for text of any length.

- When we read text from a resource or from a Signal data file we test to see if it holds correctly formed UTF-8. If it does not, we assume it holds code-page characters.

However, for performance reasons, we do not convert text from #included script files. If you have included a script file that use characters codes above 127 you must convert these by opening it in the script editor and saving it. Note that we do not count changing the format of a text, script or sequencer file from code-page based to Unicode as a change (the result should appear identical). To force a converted file to write you will need to use the File Save As command or edit the file.

### Backwards compatibility

If you update to Unicode by saving a script or a modified data file or a resource and you used character codes above 127, the modified files will no longer read correctly into non-Unicode versions of Signal. This will not stop you opening the files or resources, but any character codes above 127 will not appear correctly. If you write all your text in English this will not be a problem for you; you will see no change.

The .cfs file format has limited size fields for channel titles, units and comments. The channel units field can be a problem as this is limited to 5 8-bit characters. If you use extended European characters, these typically code as 2 UTF-8 bytes each, so you are limited to 2 of these, or 1 of these and three ASCII characters. If you use a Far East typography, such as Japanese, each character uses 3 UTF-8 bytes, so you only have space for one such character and 2 ASCII codes.

If you require 100% backwards compatibility with old versions of Signal you must use only ASCII character codes for channel titles, units comments and file comments.

### Character set used in scripts and sequences

The characters that are accepted as numbers, punctuation, keywords and variable and constant names in the script language and in the output sequencer are restricted to the ASCII set. There are additional sets of numbers, punctuation and a-z and A-Z in Unicode, for example Japanese defines wide versions of numbers and A-Z; these cannot be used. You do have a free choice of Unicode characters for comments and for string literals in the script language.

# Resource limitations

There are some limitations that are imposed on us by the operating system and the computer environment. Some of these are obvious, some less so:

### Memory

Your computer will have a fixed amount of memory (typically several GB). This is shared out between all the competing programs as they require it. If you use a lot of memory, for example by using Memory buffer channels and copying vast quantities of data into them, Windows will try to cope with this (especially if you run under a 64-bit operating system), but at some point the operating system will start swapping allocated memory to disk. This makes things much slower. If you keep allocating more memory, at some point Signal will start reporting that it is out of memory. The more memory your system has, the more you can use.

If you run under particularly restrictive rights you may find that you get error -544 when Signal tries to allocate the memory Working Set.

### Disk space

This is a pretty obvious limitation. Once you have filled your disk, you cannot use more space. However, you may well find that once a disk becomes significantly full, disk operations start to get slow. It is usually well worth periodically cleaning up your disk system by deleting unwanted files. It may also be worth de-fragmenting your disk (though this seems less important in modern versions of Windows).

### GDI and User Handles

Windows tracks the use of GDI objects (things like fonts, bitmaps, brushes, pens and drawing surfaces) and User objects (things like desktops, windows, menus, cursors, icons and menu short cuts). For reasons tied up in the history of Windows and for backwards compatibility, these have 16-bit identifiers, which meant that there can only be 65535 GDI and 65535 User handles. Each application is limited to 10,000 GDI objects and 10,000 User objects. Most programs do not use a huge number of either (typically a few hundred of each), so this is not usually a problem.

It starts to be a problem when an application creates lots of windows as each window will use quite a few handles. You can track the the number of GDI and User objects with the `App()` command. Each new view (visible or not) uses some 20 to 100 of each of these objects. This means that you can run out of handles by opening 300-400 data views (or even fewer if they all have a lot of channels). We refuse to allow you to create new views if you have used more than 9900 of either object. If this happens, you are probably running a script that is not deleting windows that it has created.

**Symptoms of resource exhaustion**

Windows does not manage running out of resources very gracefully. The symptoms include:

- Messages from Signal warning that there was not enough memory or resources to complete an action
- Missing areas or channels in window or screen repainting problems
- Text appearing in degraded fonts
- Poor system performance (to the point of appearing to stop)
- Crashes

If you start to suffer from these problems while using Signal, do the following:

- Save any volatile data in case things get so slow that you cannot continue.
- Check for a lot of hidden windows in the Windows menu Show list or in the Windows... command. Close down (and save, if required) unwanted, hidden windows.
- If you are running a script, check that you are closing all windows that you have opened. Each `FileNew()` or `FileOpen()` should have a matching `FileClose()`.
- If you are running a script, have you used a huge array space, and could you reduce it? Script users can use `DebugHeap()` and `App(-4)` commands to get information on used memory resources.

# Sampling data

If you worked through the Getting started section you already have most of the skills to sample a new data document. Sampling a new document is the same as working with an old document, except that the new document grows in length as new frames are added. There is also a special frame zero that shows incoming data from the 1401 as it is sampled and before it is written to the file, you would normally display this frame so you can see what the sampling is doing.

## Inputs and outputs

Before we discuss the sampling configuration itself, we need to provide some information on the types of input and output signal available to users of the 1401. There are two basic types of signal, analogue and digital.

Analogue signals are varying voltages that can have any value within the available range, normally this range is -5 to +5 volts. Because the 1401 handles these signals as a 16-bit binary number, the actual values available are quantised to the available integer values. The 1401 provides analogue inputs via the ADC inputs and analogue outputs via the DACs.

Digital signals use TTL-compatible voltages that can take only two values; low and high. The 1401 provides both digital inputs and digital outputs.

## ADC inputs

The 1401 ADC (Analogue to Digital Converter) measure varying voltage signals in the range of ±5 volts, these can be optionally changed to ±10 volts if required - usually this requires the unit to be returned to CED. The Power1401 mkII, Micro1401-3, and later 1401 models all incorporate software control of the ADC input range - you can switch these units between 5 and 10 volts using the 1401 options dialog in the Try1401 utility installed with Signal. The ADC generates a 16-bit integer value representing the voltage seen, with -32768 corresponding to the lowest voltage possible and 32767 corresponding to (just less than) the highest voltage. Zero corresponds to zero volts. Signal uses the ADC inputs to generate waveform channels through sampling.

### ADC input connections

To sample waveforms, connect your signals to the 1401 ADC input ports that you are sampling. Ports 0-7 (0-3 for an unexpanded Micro1401) are the labelled BNC connectors on the front of the 1401.

For early-model Power1401s without an ADC expansion topbox, ports 8-15 are on the rear panel 37-way "D-type" connector labelled Analogue Expansion. The connections on this are:

| ADC port | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | Ground |
|---|---|---|---|---|---|---|---|---|---|
| Pin number | 35 | 34 | 33 | 32 | 31 | 30 | 29 | 28 | 1-19 |

For late-model Power1401s and Power1401 mk IIs and above the Analogue expansion socket is a high-density 44-way "D-type" connector. On this connector each signal has its own separate ground return. The connections are:

| ADC port | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|
| Pin number | 33 | 34 | 35 | 36 | 37 | 38 | 39 | 40 |
| Ground return | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

For port numbers above 15 you will require an expansion topbox for your micro or Power1401. If you have a Micro1401 ADC expansion box installed, ports 4 to 15 are BNC connectors on the expansion box. If you install a Power1401 expansion box, it adds new ADC ports starting at number 8 and the port numbers on the rear expansion connector are adjusted to start after the expansion box ports, so if you add an ADC-16 topbox ports 0-7 are on the main unit, ports 8-23 are on the topbox and ports 24-31 are on the rear connector. If you try to sample using a port above the number available on your 1401, Signal will generate an error message.

# DAC outputs

The 1401 DACs (Digital to Analogue Converters) produce varying voltage outputs in the range ±5 volts, these can be optionally scaled to ±10 volts if required. DACs can be used to generate pulses with arbitrary initial values and amplitudes (as long as they lie within the DAC output voltage range), they can also generate ramps, sine waves and arbitrary waveforms.

To generate complex DAC outputs and particularly for arbitrary waveform output, the DACs must update repeatedly with new output values. The faster this is done, the more accurate the output pulses or waveforms. However, very high DAC output rates may interfere with data acquisition. Signal pulse output is limited to a time resolution of 100 microseconds, which is not fast enough to cause any interference with data acquisition.

The Power1401s have four DACs, numbered 0 to 3, while Micro1401s have two (0 and 1). Micro1401s have the DAC outputs available on BNC connectors on the left-hand side of the 1401 front panel. Power1401s have DACs 0 and 1 on the front panel and has DACs 2 and 3 on pins 36 and 37 respectively of the rear-panel analogue expansion connector. The Power1401 mk II and later versions of the Power1401 have a high-density 44-pin analogue expansion connector with DACs 2 and 3 on pins 43 and 44.

If your Power1401 has a Signal expansion top box fitted, then 8 DAC outputs are available via BNC connectors on the 1401 front panel.

# TTL compatible signals

In several places in this manual we refer to TTL compatible signals. TTL stands for Transistor-Transistor Logic and is a method of passing logical (High/Low) information between devices using voltage levels. Levels above 3.0 volts are in the High state, levels below 0.8 volts are in the Low state. Levels in between 0.8 and 3.0 volts are undefined.

The TTL inputs and outputs on the 1401 are the digital inputs and outputs, the event inputs, the clock F external frequency input, the ADC external convert input, the clock output and the trigger input. On Micro and Power1401s, the event and trigger inputs are not actually TTL but can be treated as such (they are actually heavily protected analogue inputs from which a TTL signal is derived).

The event, trigger and ADC external convert inputs are edge-sensitive; they respond to either a falling or rising TTL input, the edge to which they respond can be controlled using the Rising edge trigger option in the General tab of the sampling configuration dialog.

Do not subject TTL inputs to voltages above 5.0 volts or less than 0.0 volts. CED hardware has special circuits on TTL compatible inputs to provide some protection, however determined abuse will damage them.

The 1401 TTL compatible inputs are pulled up by a resistor to 5 volts. They require a current of some 0.8 mA to pull them into the Low TTL state. Alternatively, you can connect them to ground to pull them low (useful for the Event inputs).

See the *Owners handbook* of your 1401 interface for full details of each input port.

# Trigger input

All modern 1401 systems have a Trigger input BNC on the front panel, it is actually the 1401 event zero input but re-routed to the front of the 1401. This is a TTL input that normally responds to low-going pulses. It is possible to configure the trigger input so that it responds to high-going TTL pulses instead.

Signal generally always uses a pulse on the trigger input as the sweep trigger except in Peri-triggered sampling mode when, in addition to the Event trigger mode (which uses the standard trigger input), other forms of sweep triggers are available.

# Digital inputs

1401 systems have 16 bits of digital input available, numbered 0 to 15. They are normally used separately as two eight-bit input sets, which can cause some confusion between overall bit numbers and bit numbers within the low or high digital inputs.

Digital input bits 8 to 15 are used in External digital states mode and in Peri-trigger sampling, they can also be used as triggers for some of the dynamic clamping models. Digital input bits 0 to 7 are read by the output

sequencer and are used for digital markers. To keep the various discussions simpler, Signal refers to both sets of inputs as bits 0 to 7, the context determines if we actually mean bits 0-7 or 8-15.

The 1401 digital inputs are on a 25-way 'D-type' plug on the rear of Power1401s and Micro1401s.

Power1401s and Micro1401s have digital inputs 8 and 9 on 1401 front panel BNC sockets for easy access. These are labelled as event inputs 0 and 1 to match Spike2 usage. If the Spike2 digital I/O expansion top box is fitted, it has digital inputs 10 to 15 on front panel BNC sockets labelled as events 2 to 7 (again to match Spike2 usage).

**Digital input connections**

| Signal input bit number | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Gnd |
|---|---|---|---|---|---|---|---|---|---|
| External digital / Peri-trig pin / DC trigger pin numbers | 1 | 14 | 2 | 15 | 3 | 16 | 4 | 17 | 13 |
| Sequencer / Marker pin numbers | 5 | 18 | 6 | 19 | 7 | 20 | 8 | 21 | 13 |

# Digital outputs

The 1401 digital outputs are TTL-compatible and can be set high or low. When high they generate a voltage in the range 2.6 to 5 volts; the usual lightly loaded level is about 4.5 volts. When low they generate a voltage between 0 and 0.6 volts.

The Signal pulse output system and the `output sequencer DIGOUT` instruction control 1401 digital output bits 8 to 15 of the 16-bit digital output port. If you use the output sequencer, you can also set bits 0 to 7 with the DIGLOW instruction. Signal refers to both sets of outputs as digital outputs bits 0 to 7, the context determines which set we mean.

The 1401 digital outputs use a 25-way 'D-type' socket on the rear of Power1401s and Micro1401s.

Power1401s and Micro1401s also have digital output bits 8 and 9 on 1401 front panel BNC sockets for easy access, they are labelled 0 and 1 (the labels match Signal usage). If the Spike2 digital I/O expansion top box is fitted, it has front panel BNC sockets for digital output bits 10 to 15 labelled 2 to 7 (also matching Signal usage).

**Digital output connections**

| Signal output bit number | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Gnd |
|---|---|---|---|---|---|---|---|---|---|
| DIGLOW pin numbers | 5 | 18 | 6 | 19 | 7 | 20 | 8 | 21 | 13 |
| Pulses and DIGOUT pin numbers | 1 | 14 | 2 | 15 | 3 | 16 | 4 | 17 | 13 |

# Types of channel

Signal can directly sample two types of channel: waveform and marker. Waveform channels are generated by sampling from 1401 ADC inputs, marker channels can be generated using the keyboard or from the 1401 digital inputs. Channels in a Signal data file are numbered starting with number 1, the channel numbers are contiguous. Memory channels (either marker or real marker) can be created during sampling by measurements processing, while virtual channels will automatically work with incoming data during sampling.

# Waveform channels

The waveforms that Signal records and displays are continuously changing voltages sampled using the 1401 ADC inputs. Signal stores waveforms as a list of numbers that represent the waveform amplitude at equally spaced time intervals. These numbers are 16-bit integers, they are scaled using calibration values to produce the floating point data values that Signal uses and displays. The process of converting a waveform into a number at a particular time is called *sampling*. The time between two samples is the *sample interval* and the reciprocal of this is the *sample rate*, which is the number of samples per second. A set of samples taken at regular intervals is referred to as a *sweep*, a sweep of sampling generates a frame in a Signal data file.

Signal is also able to handle waveform channels (for example data that is the result of analysis or generated by a script) where the underlying data is floating point values; these are indistinguishable from channels using integer data in nearly all circumstances. Floating point data is more accurate and does not require calibration values, but it requires twice as much disk space. Though Signal can create and use channels based on floating-point data, Signal

data created by sampling is always based upon 16-bit integers from the ADC hardware and calibration values taken from the sampling configuration.

### ADC ports and channels

Waveform channel data is generated as described above by sampling from ADC inputs. A 1401 has a number of separate ADC inputs, these are called ports and are numbered from zero upwards – so a Micro1401 with four ADC inputs has ADC ports numbered 0 to 3. When you define a Signal sampling configuration the ADC ports that will be used are specified as a list, for example ports 0, 1 and 3. Note that these port numbers do not have to be contiguous (or even in ascending order). The ports are sampled in the order given to produce waveform data channels, the number of channels being set by the number of ports in the list. The data channels are numbered in ascending order starting with channel 1 and these channel numbers are contiguous. So when the port list given above is used to sample data, data from port 0 generates data file channel 1, port 1 generates channel 2 and port 3 generates channel 3.

### Minimum sample rate
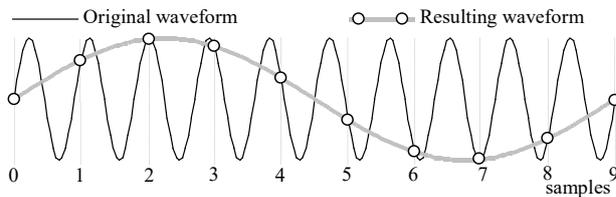


The sample rate for a waveform must be high enough to represent the data correctly. You must sample at a rate at least double, and preferably 2.5 to 5 times, the highest frequency contained in the data. If you do not sample fast enough, high frequency signals will be aliased to lower frequencies, as illustrated above. The dots in the diagram represent samples; the thin line shows the original waveform while the thicker grey line shows how the sampled data will appear. Just to keep things interesting, you also want to sample at the lowest frequency appropriate to keep the size of your disk files down.

### Voltage range

The 1401 ADC (Analogue to Digital Converter) measure varying voltage signals in the range of ±5 volts, these can be optionally changed to ±10 volts if required - usually this requires the unit to be returned to CED. The Power1401 mk II, Micro1401-3 and later 1401 models all incorporate software control of the ADC input range - you can switch these units between 5 and 10 volts using the 1401 options dialog in the Try1401 utility installed with Signal.

### Use of filters

Many users pass waveform data through amplifiers or signal conditioners with filter options to limit the frequency range. Some transducers have a limited frequency response and require no filtering. If you are concerned about measuring the frequency content of your data, you should ensure that your waveform data is filtered to remove frequencies above 50% of the sampling rate – the *Nyquist frequency*.

# Marker channels

Signal can sample two types of marker data: keyboard markers and digital markers. Signal treats both marker types identically once the data has been captured; they differ only in their source. Marker channels can only be logged by Signal when not using fast sweep modes.

A Marker is a 32 bit time value, in units of the sample interval on waveform channels for keyboard markers and of the output resolution for digital markers. In addition to the time a marker has 4 bytes of marker code data. The first of these 4 data bytes is the ASCII code of the keyboard character pressed by the user (for a keyboard marker) or an 8 bit digital code read by the 1401 from the digital inputs (for a digital marker). The remaining 3 bytes are normally zero.

### Keyboard markers

Keyboard markers time events to an accuracy of, at best, around 0.1 second, you should use digital markers if you require precise timing. The upper and lower case characters a-z and the numbers 0-9 are logged, but *only when the*

*new document window or the sampling control panel is the current window*. The keyboard marker channel, if created, is the first channel number after the sampled waveform channels.

### Digital markers

These are not available for the Standard 1401. Digital markers are timed as accurately as the outputs and record 8 bits of TTL data. These can be used as 8 separate channels of on/off information or one channel of 8 bit numbers or any combination in between. Digital marker data is sampled when a low going TTL compatible pulse is detected as described below. The data is read from bits 0 to 7 of the 1401 digital input. Digital markers can also be generated by the Pulses and Sequencer output systems, for example to show when something was done, in these cases the marker data can either be read from the 1401 digital inputs or set directly by the outputs. The digital marker channel, if created, is the first channel number after the keyboard marker channel or after the waveform channels if the keyboard marker is not used.

### Digital marker connections

The digital marker data is read from the 1401 digital inputs bits 0 to 7. These inputs are found on the 1401 Digital inputs connector; a 25-way 'D-type' plug located on the rear of Micro1401s and Power1401s. In addition to the data lines a TTL pulse is required on the digital inputs Data Available input on pin 23 to log a digital marker.

### Digital input: bits 0 to 7

| Digital input bit | bit 7 | bit 6 | bit 5 | bit 4 | bit 3 | bit 2 | bit 1 | bit 0 | Gnd |
|---|---|---|---|---|---|---|---|---|---|
| Digital input pin | 5 | 18 | 6 | 19 | 7 | 20 | 8 | 21 | 13 |

### Digital input: other signals

| Signal | All 1401s |
|---|---|
| Data Available | pin 23 |

To log a digital marker, apply a low going TTL pulse at least 1 µs wide to the Data Available. When the 1401 detects the falling edge of the input, it latches the input data so the data inputs do not need to remain stable for more than a few microseconds after the Data Available falling edge.

# Marker codes

When Signal displays data from the keyboard marker, digital marker or other marker channels, it shows a marker code value as well as the marker time. The code displayed is normally the first marker code but can be selected using the draw mode dialog.

Marker codes have values from 0 to 255. This is the same range of numbers that the ASCII character set uses, and it is sometimes convenient to treat the codes as ASCII character codes (for instance when dealing with keyboard markers). At other times it is more convenient to deal with the codes as numbers.

Whenever Signal displays a marker code that has the same value as the ASCII code of a printable character, it displays the code as a character, otherwise it displays the marker code as a two digit hexadecimal number. Hexadecimal (base 16) numbers use the standard digits 0 to 9, but also use a to f (for decimal 10 to 15). Thus 00 to 09 hexadecimal is equivalent to 0 to 9 decimal. 0a to 0f is equivalent to 10 to 15 decimal. 10 to 1f hexadecimal is 16 to 31 decimal, 20 to 2f is 32 to 47 decimal and so on.

| + | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | a | b | c | d | e | f |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 20 |   | ! | " | # | $ | % | & | ` | ( | ) | * | + | , | - | . | / |
| 30 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | : | ; | < | = | > | ? |
| 40 | @ | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O |
| 50 | P | Q | R | S | T | U | V | W | X | Y | Z | [ | \ | ] | ^ | _ |
| 60 | ` | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o |
| 70 | p | q | r | s | t | u | v | w | x | y | z | { | \| | } | ~ |  |

The printing characters are 20 to 7e hexadecimal, 32 to 126 decimal, as in the table. Character 7f may or may not print, depending on the character set. To find the hexadecimal code of a printing character, add the number above it to the number to the left of it. For example, the code for A is 41. To convert a code to a character, look up the first digit in the left column and the second in the top row. For example, 3f codes to ?, the intersection of the row for 30 and the column for f.

# Sampling configuration

A Signal sampling configuration contains all the information and settings that Signal needs in order to create a new data file by sampling. This information includes fairly obvious things like how fast to sample waveforms in the 1401 and what 1401 outputs to be generated during sampling, and also extra information such as how the sampled data is to be displayed and what online processing to create measurement channels or derived data documents should be carried out. Signal always holds a sampling configuration in memory, this is the current sampling configuration. The current sampling configuration is used for all sampling, you can change the current sampling configuration by using the File menu Load Configuration command. When Signal starts up, it normally loads the last-used sampling configuration into memory so that the setup will be the same as when it was last used.

Before you start to sample data with Signal you must set up the sampling configuration. This is done through the Sample menu Sampling Configuration dialog, which is also available by using a toolbar button.

The sampling configuration dialog is a tabbed dialog - it contains a number of tabs for selecting different sections of the parameters. Click on a tab to display the corresponding page of the dialog. Some sections of the sampling configuration dialog are always present, while others are shown or hidden according to the circumstances. The sections that are always available are General, Port setup, Outputs and Automate. The other sections, Peri-trigger, States and Clamp hold extra information not relevant to all sampling configurations. The Peri-trigger tab appears only when the sampling mode selector in the General section is set to Peri-trigger. Similarly, the States tab only appears when the General section Multiple frame states item is checked. The Clamp tab is only available if clamping experiments are enabled using the Clamp page of the Preferences dialog.
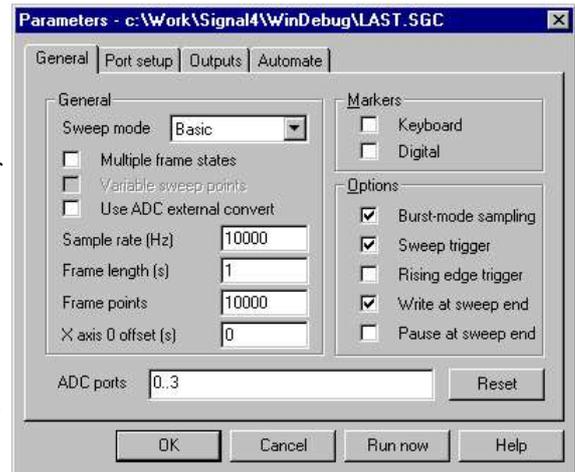
The title of the dialog shows the filename and path of the configuration file from which the configuration data was read. A '*' character is appended to the file name if the current sampling configuration contains unsaved changes.

Press OK to keep any changes you made to the sampling configuration or Cancel to discard changes. The Run now button keeps any changes and then immediately starts off sampling.

Signal sampling configurations are saved on disk as files with a `.sgcx` filename extension using the File menu `Save configuration as` command. Older version of Signal used a different sampling configuration file format that used a `.sgc` filename extension. Signal can read and use these older sampling configuration files, but before you can save them in the new format it is necessary to convert them to the new form by using them for sampling.

# General configuration

The General section holds a selector for the sweep mode, the multiple frame states check box, fields to define the waveform sampling rate and the frame width or points, check boxes to control the creation of marker channels and various other options, plus a list of the ADC ports to sample to generate waveform channels.

**Sweep mode**

The Sweep mode selector defines how sweeps of data are taken and triggered and how sampling sweeps relate to the outputs system. The modes available are:

Basic
: the trigger for a sweep of data is a TTL pulse at the start of the sweep, and pulse outputs start and finish at the same time as the sampling sweep. We have tried to keep this mode simple to use to help beginners with the Signal software. It is useful for a wide variety of straightforward sampling requirements.

Peri-trigger
: the trigger point can be before the start of the sweep, at the start of the sweep or at any point within the sweep. Pulse outputs start at the trigger point and finish at the end of the sampling sweep. This mode allows a wide variety of triggers including threshold crossings on a sampled waveform channel, and allows data to be collected before the sweep trigger, which no other sweep mode can achieve. The trigger point and type of trigger are set in a separate peri-trigger configuration page.

Extended
: the pulse outputs are triggered rather than the sweep. The outputs can be longer than the sampling sweep, occur both before the sweep starts and after the sweep is over and the sampling sweep is triggered by the outputs; the sampling sweep trigger time is set along with the outputs in the pulses dialog. This is a very powerful sweep mode while still being useful for general-purpose use.

Fixed interval
: similar to Extended, but the sweeps are internally timed so that they occur at the specified interval; a random variation in the interval can optionally be provided. Both the sweep interval and any required random variation are set using the pulses dialog that is used to define the outputs. External sweep triggers are not used in this mode, needless to say. The fixed interval must be longer than the overall sweep length so that Signal has enough time to get things ready for the next sweep. The amount of time required varies a lot with the sampling and computer system, 100 to 200 milliseconds is usually sufficient.

Fast triggers
: like Basic mode except that marker logging, multiple frame states and variations in the pulse outputs are not available, in addition the absolute start time for a sampled frame cannot be measured and is set to zero. This keeps the inter-sweep interval to a minimum - less than 100 microseconds usually.

Fast fixed interval
: has the same limitations as Fast triggers but uses a fixed interval between sweeps rather than requiring an external trigger; in this mode no random variation in the interval is available - the interval is set using the pulses dialog. The fixed interval still has to be longer than the overall sweep length, but only by a millisecond or two.

Gap free
: is like Fast triggers mode but only the first frame is actually triggered; subsequent frames start immediately after the previous frame finishes with no loss of data, change in sampling interval between points or any other variation in sample timing. It is used when you want to sample continuously but are happy to break up the data into frames. If you want full-blown continuous sampling then you should consider using the CED *Spike2* software, Signal cannot do this very well.

Useful extra information relevant to some of these modes can be found in *Pulse outputs during sampling* – the pulses dialog is used to set the fixed interval sweep timing and other parameters.
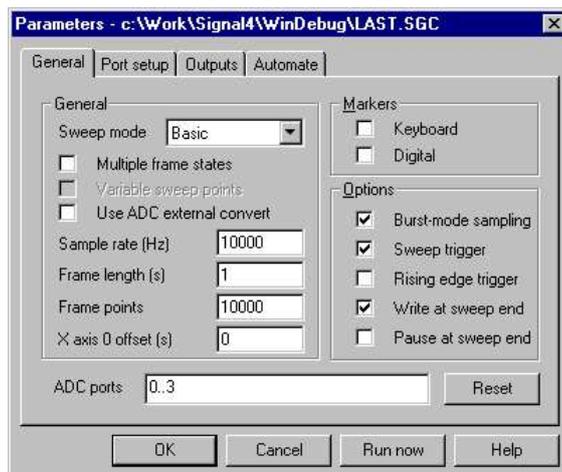
## Multiple frame states

This check box enables sampling with multiple frame states. With multiple frame states disabled, all sampling sweeps are the same, the same pulse outputs are generated and the new data frames are set to state zero. With multiple frame states enabled, each sampling sweep can be different from other sweeps in a number of ways and the data frame states are different to indicate what happened during sampling. This can be used to achieve a variety of useful effects.

The use of multiple frame states is a complex topic which is covered separately in *Sampling with multiple states*.

## Variable sweep points

This check box is only available with Extended and Fixed interval sweep modes with multiple frame states enabled; it allows different sampled sweeps (with different states) to have different numbers of ADC data points. The number of data points for each state is set in the pulses outputs dialog, the sweep points set in this page sets the upper limit for sweep data points, this will also be the maximum display width for such a file. When you change the frame length or points (using the controls below) with variable sweep points in use, the variable points values are checked to make sure that they are legal, but note that when you change the data sampling rate the variable points values are <u>not adjusted</u> to keep the individual sampling sweep durations unchanged - you will have to do that yourself if you want the same sweep lengths.

When variable sweep points data is displayed, the allowed X range is set by the maximum number of sweep points which is set here, so you may find yourself looking at frames of data that do not reach the limits of the X axis. But apart from that they will behave normally.

## Use ADC external convert

This check box selects the use of an externally generated pulse to trigger ADC conversion rather than the internal 1401 clock that is normally used. When this check box is set, the Sample rate and Frame length controls are replaced by an X axis units field which sets the X axis (which would normally be time) in the generated data and a Points per X unit field that sets the separation of the sampled ADC data points on the X axis.

The external ADC convert  pulse should be connected to pin 6 of the Events socket on the back of the 1401, this is the ADC Ext Conv input. The pins are numbered from right to left with pin 6 being the 6th hole along on the top row. Normally, the ADC Ext Conv input is sensitive to a falling edge - an ADC conversion is triggered when the signal level changes from TTL high to TTL low - the polarity of this signal is inverted by the use of the Rising edge trigger control, below.

## Sample rate

The Sample rate field sets the sampling rate for all waveform channels, in Hz. The rate displayed will not always be exactly the preferred rate that was entered; it shows the closest achievable rate given the 1401 clocks and the number of ADC ports to be sampled. This occurs because the 1401 ADC converter clock can only generate sample intervals that are an exact multiple of the clock input frequency. In all modern 1401 designs an input frequency of 10 MHz is available (in older 1401 designs the maximum input frequency is 4 MHz) so the ADC clock will generate sample intervals that are a multiple of 100 nanoseconds. This limits the higher sampling rates available. In very modern 1401 types (the Power3A and the Micro4) a 200 MHz internal frequency is available, giving sample intervals that are a multiple of 5 nanoseconds and much closer adherence to your desired sampling frequency. It is important to emphasize that, though you may not get exactly the sampling rate that you want, this will not cause inaccuracies in your sampling - Signal knows exactly what sampling rate it is actually using and will ensure that the timing of all ADC data is correctly derived from this actual sampling rate.

### Getting the correct sample rate

As described above, the ADC clock interval has to be an exact multiple of 100 nanoseconds and this can make it impossible to get the exact sampling rate that you want. For example suppose you want to sample at exactly 3 KHz. To do this the ADC clock would have to generate an interval of 333 and 1/3 microseconds. This cannot be done as 1/3 of a microsecond is clearly not a multiple of 100 nanoseconds and so Signal will use the closest achievable sampling rate (approximately 2994.012 Hz).

If you cannot get the correct rate, it is often possible to fix this, or at least a rate much closer to the one you want, by enabling the Burst mode check box described below. This is because of the way burst mode sampling operates. Consider a sampling configuration that samples 8 ADC ports at a rate of 20 KHz - this means a 50 microsecond interval between samples on each ADC port. With burst mode disabled the ADC clock must generate a sample interval of 6.25 (50/8) microseconds, clearly this cannot be achieved when all intervals have to be an exact multiple of 100 nanoseconds. However with burst mode sampling is enabled the ADC clock has to generate a 50

microsecond interval which is of course easily done. For this reason we recommend always using burst mode sampling unless you need to exactly match the behaviour of previously-used sampling configurations.

*Maximum sample rates*
The overall sampling rate in the 1401 is the Sample rate times the number of ADC ports. With a Power1401 625, the maximum overall sampling rate is 625 kHz. A Micro1401 mk II and Micro1401-3 will sample at up to 500kHz, while a Power1401 mkII and Power1401-3 will go up to 1MHz. With an original micro1401 the maximum overall sampling rate is 333 kHz.

The sampling configuration dialog does not apply hardware-specific limits to the sample rates that you enter - you can enter any sampling rate that you want. If you use a sampling configuration with an overall sampling rate beyond that achievable with your hardware a 1401 sampling error will occur and be reported by Signal. If Signal detects a Power1401 (or other more modern 1401 type) during program startup, it enables the use of higher timing resolutions. You can force Signal to allow this higher timing resolution by checking the Assume Power1401 hardware box in the Sampling page of the Edit menu Preferences dialog.

*Variable sweep points*
If variable sweep points are in use, note that sampling rate changes are not compensated for by adjusting the variable points values for each state and so the individual sampling sweep durations will change.

## Frame length and points
The Frame length and Frame points fields set the length of the sampled frame. The frame length is always shown in the appropriate time units. Changes made to one of these fields automatically cause an appropriate change in the other. The Frame length field also updates whenever the sampling rate changes.

The maximum frame length possible varies with the model of 1401 and the amount of 1401 memory installed; each sampled point requires two bytes of memory. For a standard micro1401 the limit is about 480,000 while for a Power1401 the limit depends upon the amount of extra memory installed but is at least 15 million. For a Micro1401 mk II or -3 the limit is either about 480,000 or 1 million depending on the amount of memory the unit was built with. The sampling configuration dialog does not apply any limits to the frame length that you enter, when sampling starts the 1401 memory required is checked against the memory that is available.

## X axis zero offset
Normally the X axis zero appears at the start of the frame, or at the trigger time for peri-triggered sampling mode. This position can be moved by entering a non-zero value in this field. This does not alter how and when sampling takes place, only the way in which times are displayed on the x-axis.

## ADC ports
This field sets a list of ADC ports to sample, up to 80 ports can be specified in the list. You can enter individual ADC ports separated by commas or spaces or a range of ports such as 0..7 or both (for example "0,7,1..6"). Port numbers between 0 and 127 are accepted, the list of ports can be in any order that you want, unwanted ports can be freely omitted and duplicated port numbers are allowed.

Each ADC port in the list creates a separate waveform channel in the resulting data document. The ports are sampled in the order they are given in the list, so the first port in the list generates data channel number 1, the second channel 2, and so forth.

## Keyboard marker
The Keyboard marker check box enables the creation of the keyboard marker channel and logging of keyboard markers. If the keyboard marker channel is enabled then it is the first channel in the data document after the waveform channels. Keyboard markers cannot be logged when using fast sweep modes.

## Digital marker
The Digital marker check box enables the creation of the digital marker channel and logging of digital markers from the 1401 digital inputs, digital marker data items can also be created by Pulses or Sequencer outputs. If this channel is enabled it is the first channel after the keyboard marker channel or the waveform channels if the keyboard marker channel is not present. Digital markers cannot be logged when using fast sweep modes.

### Burst mode

Check this box for burst mode sampling, leave it clear for equal interval sampling. In equal interval sampling the waveform data points are sampled individually in turn and the interval between samples is 1/(Sample rate * number of ADC ports) - this gave some slight advantages with very early 1401 designs. In burst mode sampling all of the ADC ports are sampled in a burst as close together as possible and the interval between bursts is 1/Sample rate. Burst mode sampling is generally preferable as it ensures that the interval between samples on adjacent ADC ports is kept to a minimum and it very often allows the required ADC sampling rate to be achieved precisely, as described above.

If the first two ADC ports sampled are ports 0 and 7 (or 0 and 3 for a micro1401, Micro1401 mk II or Micro-3), then the second sample and hold circuit optionally fitted to 1401s is enabled. If fitted this option causes the sampling on ports 0 and 7(3) to be exactly simultaneous. With Peri-triggered sampling mode burst mode is always used for efficiency reasons.

### Sweep trigger

This check box sets the initial state of the Sweep trigger check box in the sampling control panel, this check box enables and disables sweep triggers. With sweep triggers enabled, a sampling sweep will not occur until a trigger has been detected, the sampling configuration determines what a trigger is. With sweep triggers disabled, a sampling sweep starts immediately. For Extended sweep mode, the trigger starts the outputs rather than the sampling sweep.

Except when using Peri-trigger sweep mode, the sweep trigger is a low-going (or high-going if a rising edge trigger is selected below) TTL pulse supplied to the Trigger BNC input on the 1401 front panel. For Peri-triggered sweep mode the trigger can be any of a number of signals.

### Rising edge trigger

Sweep triggers normally occur on the falling edge of the supplied TTL pulse on the 1401 Trigger input, check this box to make them occur on the rising edge. This also controls the polarity of the ADC convert input used if Use ADC external convert mode is selected (see above) and the event 1 input that cab be used for a precisely timed start to sampling. Both of these inputs are also normally sensitive to a falling TTL edge and to a rising edge if this box is checked.

### Write sweep to disk

This check box sets the initial state of the Write to disk at sweep end check box in the sampling control panel. When this is set, sampled sweeps are automatically written to disk when the sweep finishes.

### Pause at sweep end

This check box sets the initial state of the Pause at sweep end check box in the sampling control panel. When this is set, Signal waits at the end of a sweep instead of immediately starting the next sampling sweep.

### Reset

Pressing this button resets all of the sampling configuration parameters to the default state.

# Peri trigger configuration

The Peri-trigger section holds information that is specific to the Peri-trigger sampling mode. It is only available when Peri-triggered sweeps are selected in the General section.

At the top of the dialog is a selector for the type of trigger and a field for the pre-trigger points. Below this is a section holding details of the trigger parameters. The contents of this section changes with the type of trigger, the individual fields will be described along with the various types of trigger.

**Trigger type**

This can be set to one of +Analogue, -Analogue, =Analogue, Digital or Event. The three analogue types monitor the last ADC port in the sampled ADC ports list for a trigger. The trigger levels are shown with the sampled data as a pair of cursors which can be moved, without stopping the sampling, to alter the levels. The Digital trigger waits for a specified state on a bit in the 1401 digital inputs, while the Event trigger is a TTL pulse just as for the Basic sample mode triggers. Each form of trigger has different parameters:

+Analogue     Trigger on a positive-going level transition on the last ADC port in the sampled list. The parameters are Threshold level and Hysteresis, both in units set by the channel calibration. The trigger process first waits for the sampled data to go below (Threshold - Hysteresis) and then triggers when the sampled data value rises above Threshold. The hysteresis acts to prevent false triggering by noise as the sampled data passes downwards through the threshold level, triggering can only occur after the sampled data has clearly been below the threshold. If you find that you are having problems with false triggers due to noise, increase the Hysteresis value.

-Analogue     Trigger on a negative-going level transition on the last ADC port in the sampled list. This is identical to +Analogue, but in the opposite direction. The trigger process first waits for the sampled data to go above (Threshold + Hysteresis) and then triggers when the sampled data value falls below Threshold.

=Analogue     Trigger on signal moving outside a pair of levels on the last ADC port in the sampled list. The parameters are Upper threshold and Lower threshold. The trigger process first waits for the sampled data to go between the thresholds. It then monitors the sampled data and triggers when the sampled data value is above the upper level or below the lower level.

Digital     Trigger on a digital input bit state. The parameters set the digital input bit, from 8 to 15 and select triggering on a high bit or on a low bit. The trigger occurs when the bit is in the correct state. There is no requirement for the bit to be in the other state first. The digital inputs are found on the 1401 digital inputs connector, the pins for the digital bits are:

| Digital input bit | bit 15 | bit 14 | bit 13 | bit 12 | bit 11 | bit 10 | bit 9 | bit 8 | GND |
|---|---|---|---|---|---|---|---|---|---|
| Digital input pin | 1 | 14 | 2 | 15 | 3 | 16 | 4 | 17 | 13 |

Event     Trigger on a TTL pulse. There are no parameters; the trigger occurs when a TTL pulse is detected on the Trigger BNC input on the 1401 front panel. The Rising edge trigger option in the general sampling configuration operates normally.

**Pre-trig. time**

This parameter sets the amount of data in the frame that will be sampled before the point at which the trigger occurred. This can have any value from -(1,000,000 * sample interval) to the length of the frame -(2 * sample interval). If the value is negative, this means that points sampled after the trigger occurs are discarded before the first point in the frame is kept. If the value is positive, then the specified time must have elapsed before the search for a trigger begins and the resulting frame contains points sampled before the trigger occurred.

When a non-zero pre-trigger time is specified the resulting data x axis adjusts to start at -pre-trig. time. Thus a negative value gives an x axis starting at some positive value because the first point in the frame was sampled some time after the trigger. Similarly, a positive value gives an x axis starting at a negative value as some points sampled before the trigger are shown.

# Ports configuration

The port setup section defines settings for the individual ADC ports used to sample waveform channels. You can set the scaling and units for data sampled from a port, the name of the data channel created by sampling and specify online processing options for data from a port. This section defines settings for each ADC port, note that the actual ports that will be used for sampling are set in the General page of the configuration dialog.

The main dialog displays the current settings for all ADC ports (note that this can include ports that are not available with the current 1401 hardware). Double-click on the entry for a particular port to open the parameters dialog for that port and change the parameters. The entries for each port (both in the main dialog and in the parameters dialog) are:

Zero
: The value (in the specified units) corresponding to a zero volt reading from the ADC. This value, along with Full, is used to convert ADC data into the floating-point values used by Signal.

Full (5.0V)
: The value (in the specified units) corresponding to the full scale reading from the ADC. To scale the data in volts, this will be 5 for a 5 volt 1401 and 10 for a 10 volt 1401. The 1401 ADC range in use is shown alongside the title.

Units
: The units for calibrated data. This is a string from 1 to 6 characters long, you can set any units string that you want.

Name
: The port name. This is a string from 1 to 19 characters long, it sets the title of the waveform data channel sampled from this port.

Options
: This is a string of 0 to 8 characters that holds online processing options for data from the port. Characters corresponding to various processing options can be entered into this dialog. Currently, only one processing option is available; enter an 'R' character to cause online rectification of sampled data. If the port scaling is being altered or controlled by an amplifier telegraph, this field shows Tel (or ATel if an auxiliary telegraph) in red to indicate this fact. If the port is being used by the standard 1401 telegraphs to control the scaling of another port this field shows T>n in red, where n is the port number that is controlled.
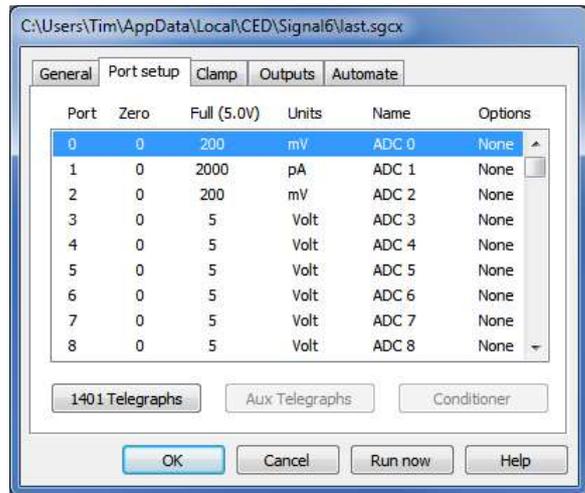
### Amplifier telegraphs

The 1401 Telegraphs and Aux Telegraphs buttons on the bottom provide dialogs that allows you to configure amplifier telegraph support. Amplifier telegraphs are signals, usually analogue outputs, from an amplifier that indicate amplifier settings such as gain and offsets. By collecting and interpreting the telegraph signals, Signal can automatically adjust for changes in the amplifier settings. Signal supports a standard telegraph mechanism using analogue voltages sampled using the 1401 and it can also use a custom DLL to support alternative mechanisms.

The standard telegraph mechanism using a voltage signal is accessed and configured using the 1401 Telegraphs button on the left. If support for an auxiliary telegraph system has been installed then central button will be enabled and its label will change to indicate the type of auxiliary telegraph in use. For information on setting up and using amplifier telegraphs, see *Amplifier Telegraphs*. Only a few types of amplifier provide telegraph signals; if you do not have such an amplifier you can ignore these option.

### Conditioner or quick calibration

The Conditioner… button on the bottom right opens the signal conditioner setup dialog if a signal conditioner channel for the currently selected port has been found (see *Programmable Signal Conditioners*). If the name of the signal conditioner hardware is known to Signal the label on this button will change.

If no signal conditioner support has been selected, the Conditioner... button will change to Quick Calibration. The Quick calibration dialog offers an easy way to set up the calibration of the current port as a voltage based upon the amplifier gain and the desired units for data from that port. To set your calibration, select the desired units from the available choices of kV,Volt, mV, μV and nV, and enter the total amplifier gain applied to the signal between the preparation and the 1401 ADC input. If you have a headstage or other pre-amplifier don't forget to include that in your calculation of the gain.
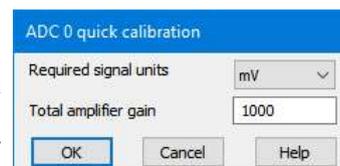
*More on channel scaling and amplifier gains*

The Zero, Full and Units fields are important; they define the way that 1401 ADC data is converted into the values that you will see throughout Signal. Initially these values are set to give readings in volts at the 1401 ADC inputs, you can keep them as they are but often it is more useful to change them to show values more closely connected to the experiment.

For example, imagine you are using an amplifier with a gain of 100 to amplify the signal from an electrode before the signal is fed to the ADC input. A gain of 100, for those that are unfamiliar with electronics, means that the amplifier output voltage at any point will be the input voltage multiplied by 100. Now you could leave the calibration of this signal alone and just log the voltage connected to the 1401 but it is usually much neater, and harder to make mistakes, if you adjust the signal calibration to take the amplifier gain into account so that the calibrated signal levels show the voltage at the electrode. To do this you have to set the Full value to show the electrode voltage that would result in a full-scale reading (we will assume your 1401 uses a 5-volt range in this discussion). As gain is a simple multiplier we can work out what electrode voltage gives a 5-volt amplifier output by dividing 5.0 volts by the gain in use; 100. That gives us 0.05 so you would set the Full value to that.

That will work fine but numbers running from -0.05 to +0.05 are not particularly convenient; it would probably be much nicer to work in millivolts. To do this you would set the units to "mV" and multiply the Full value by 1000. That gives us an input range of -50 to +50 mV which is a lot easier to work with. The same arithmetic applies to whatever gain your amplifier provides - just divide the actual 1401 full-scale value (5 or 10 volts, in volts, millivolts or microvolts as you prefer) by the amplifier gain to get the Full value to enter. But in nearly all circumstances, the Quick Calibration dialog described above will take care of this for you; this description is provided to give you a more complete understanding of what is going on.

Only a few amplifiers have an offset control that allows removal of standing DC levels while still retaining DC measurements (one of the few that do is the 1902 from CED). Unless you are using one of these you will never need to do anything with the Zero value and can leave it set to zero (the 1902 control software will take care of the Zero setting for you).

Going beyond altering the port settings to take account of amplifier gain, you can also use them to generate correctly calibrated values from a transducer which measures a quantity other than voltage. The arithmetic required is much the same; calculate the transducer input that will result in a full-scale value at the 1401 ADC input and set that as the Full value (and don't forget to set the correct units). For example consider a pressure transducer that generates 1 volt for each 50 mmHg applied. Then a 5 volt 1401 input will be generated by 250 mmHg pressure and you should set the Full value to that and set the port units to "mmHg". If the transducer electronics includes a selectable gain then you apply the gain in the same way as for voltage - by dividing the Full value by the gain. So if your transducer gain was changed to 5 you should change the Full value to 50 (250/5).

# Clamp configuration

This section of the sampling configuration dialog is only available when clamping support is enabled in the Clamp page of the Preferences. It is described under *Sampling with clamp support*.

# Outputs configuration

The outputs section of the sampling configuration is used to set the outputs required during sampling, which DACs and digital outputs are available for use and to set the DAC units and scaling. The leftmost area is used to configure the type of outputs. It contains a selector for the type of outputs required plus items specific to the currently selected type of outputs. The right-hand areas enable and set up the output ports.

### Outputs type

This control selects the type of outputs to use from either None, Pulses or Sequence. The controls in the area below the selector vary according to the selection.

## Outputs type: None

This disables outputs during sampling. When selected, only a single control is shown:

### Timer period (ms)

This item sets the period of the internal timer used to measure the absolute frame start time and to time digital markers. A value of 1 to 10 ms is usually appropriate for these purposes. Values from 0.1 microseconds to 250 ms can be entered and they are rounded to the nearest 0.1 microseconds.

## Outputs type: Pulses

This selects pulse outputs during sampling. The pulses can be controlled by the script language or interactively using a dialog. The details of configuring and using pulse outputs are covered separately under *Pulse outputs during sampling*. When pulse outputs are in use, a number of controls to configure the pulses are shown:

### Resolution (ms)

This sets the timing resolution of the output pulses in milliseconds or microseconds and also sets the period of the internal timer used to measure the absolute frame start time and to time digital markers. Values are rounded to the nearest 0.1 microseconds. The practical limit to the resolution depends upon the type of 1401 in use; for the micro1401 values down to 0.1 ms can be used, for the Micro1401 mk II and Micro1401-3 25 microseconds, while for Power1401s you can go down as far as 6 to 10 microseconds.



### Maximise wave rates

The design of the 1401 is such that it is possible to maximise either the timing precision with which the pulses are generated or the highest possible arbitrary waveform output rates, but not both. Normally you should leave this check box clear but if you want high waveform rates and are getting errors indicating the rate in use is too high, setting this check box may help, at the cost of a very small increase in the timing jitter in the DAC outputs. We have also found that setting this check box allows slightly higher rates of sampling with dynamic clamping before slow dynamic clamp updates occur.

### Absolute levels

This selects between absolute and relative pulse levels. With absolute pulse levels, the pulse amplitude sets the level directly, with relative levels the pulse amplitude is added to the level before the pulse to get the actual pulse level.

### Absolute times

This selects between absolute and relative pulse times. With absolute times, the pulse dialog allows you to enter the pulse start time directly; with relative times you use the delay since the start of the previous pulse. This control only affects the way in which the pulse dialog handles pulse start times, not the other times shown in the dialog, the underlying pulse data or the generation of pulses.

### Configure pulses

Press this button to configure the output pulses using the pulse configuration dialog. Details of doing this are covered in *Pulse outputs while sampling*.
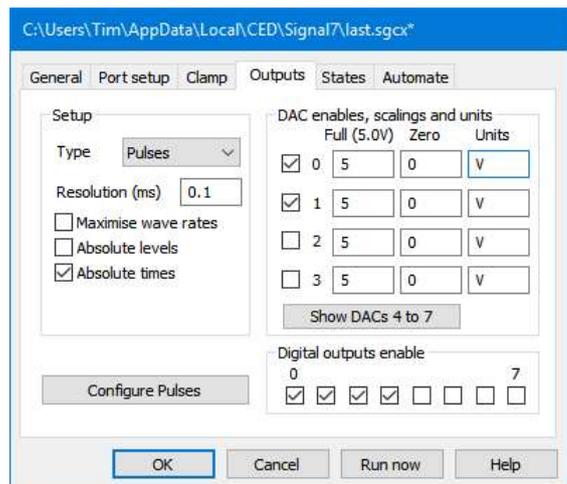
### DAC enables, scaling and units

This section contains four sets of controls, one for each DAC (users of micro1401s and Micro1401 mk IIs should ignore DACs 2 and 3). These control if a DAC is available for use and set the scaling and units with which DAC values are defined.

Enable    These check boxes enable the DACs for use. Set a check box to use a DAC, leave it clear otherwise. The fewer DACs are enabled for output the more space is available for the display of each DAC in the pulse dialog.

Zero      The value (in calibrated units) corresponding to a zero value output from the DAC. This value, along with Full, is used to convert the floating-point values used by Signal into the integer quantities actually used by the DAC hardware. This conversion process occurs when generating pulse outputs, when waveform data is pasted into an arbitrary waveform pulse and when compiling pulse sequences.

Full      The value corresponding to the full scale output from the DAC. For DACs calibrated in volts set this to 5 for a 5 volt 1401 and to 10 for a 10 volt 1401. If your 1401 has a patch clamp scaling card fitted, this scales DAC 1 in a Power1401 or Micro1401. To calibrate the DAC in volts, set Full to 2.048 or 10.24 depending on the scaling card setting.

Units     The units with which the DAC output scaling is specified. This is a string from 1 to 6 characters long.

On the Power1401 DACs 2 and 3 are available on pins 36 and 37 respectively of the rear-panel analogue connector. If a Signal top-box is fitted, DACs 2 to 5 will be available on the top box front panel, with DACs 6 and 7 available on pins 36 and 37 respectively of the rear-panel analogue connector. If a Spike2 top-box is fitted then DACs 2 and 3 will be available on the top box front panel, with DAC's 4 and 5 available on pins 36 and 37 respectively of the rear-panel analogue connector.

### Digital outputs enable

This section contains a set of check boxes to enable and disable the individual digital outputs for use. Set the check box to use this digital output port, leave it clear otherwise. The fewer digital outputs are enabled for output, the more space is available for the display of each output in the pulse configuration dialog. See *Pulse outputs during sampling* for details of the digital outputs.

# Outputs type: Sequence

This type of operation generates pulses and other outputs using a list of sequencer instructions that are executed inside the 1401 at a specified rate. Each instruction carries out a simple function such as setting a DAC to a given value, waiting for a specified time or looping. The sequencer instructions are generated using a output sequence file; a form of text document that is edited and viewed within Signal using a Sequence view. Though writing a sequence is a more complicated task than generating outputs using the pulses dialog, the sequencer allows for more complicated behaviour and can be very useful for the more demanding operations.

The sequencer includes 256 variables that can hold values and a table of data that can be quickly read and updated by scripts running in Signal.

The details of the sequencer language and instructions are covered separately under *Sequencer outputs during sampling*.

When Sequencer outputs are selected, a number of sequencer controls are shown:
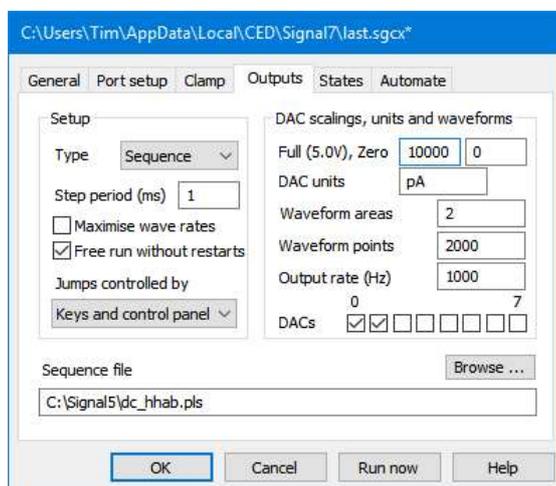


### Step period ms

This item sets the clock interval for sequencer instruction execution and thus the rate at which sequencer instructions are executed. It functions identically to the Pulses outputs Resolution (ms) control, it has the same hard limits of 0.1 microseconds to 250 milliseconds, the same rounding to the nearest 0.1 microseconds, and the recommended limits for the various types of 1401 are the same; for the micro1401 values down to 0.1 ms can be used, for the Micro1401 mk II and Micro1401-3 25 microseconds, while for Power1401s you can go down as far as 6 to 10 microseconds. Note however that some instructions will have a speed penalty that will start to limit how

fast you can run the sequencer on modern 1401's. `DIV` and `RECIP` in particular may take twice as long to execute as most other instructions.

### Maximise wave rates

The design of the 1401 is such that it is possible to maximise either the timing precision with which the sequencer executes or the highest possible arbitrary waveform output rates, but not both. Normally you should leave this check box clear but if you want high waveform rates and are getting errors indicating the rate in use is too high, setting this check box may help, at the cost of a very small increase in the timing jitter in the DAC outputs. We have also found that setting this check box allows slightly higher rates of sampling with dynamic clamping before slow dynamic clamp updates occur.

### Free run without restarts

If this item is left clear, sequencer execution will be restarted at the first instruction at the start of each sampling sweep (specifically, at the time that the data point at time zero is sampled). This allows you to easily produce sequencer outputs at a particular time in the sampled data, but the sequencer is halted between sampling sweeps. If this item is checked, the sequencer starts running at the time that sampling starts, before the first sampling sweep is started, and continues to run until sampling is stopped.

### Jumps controlled by

Sometimes you may want to stop users activating sequence sections with the keyboard or from the sequencer control panel, for example when an inadvertent change in a DAC output controlling a force feedback device might hurt the subject. This item allows you to do this; you can choose between Keys and control panel, Control panel and Script only. The script language equivalent is `SampleSeqCtrl()` The script language `SampleKey()` command can always activate sequencer sections.

### Sequence file

This control defines the file holding the instruction sequence to be used. You can either enter a file name directly or you can use the Browse button to select the sequence file directly.

### DAC scaling and waveforms

This section contains fields that define how DAC outputs are calibrated for use with sequencer outputs and how waveforms used by the sequencer are calibrated. It also sets the arbitrary waveform output rate, number of waveform areas, the length of each area and the DACs used. Both the sequencer itself and the arbitrary waveform output system assume that all the DACs have the same scale factors and units setting. See `SampleSeqWave()` for the script language equivalent.

Full and zero
These fields define the full scale output level of the DACs in the units that you wish to use corresponding to a full-scale output from the DAC and similarly the value corresponding to a zero-volt output from the DAC. They are used to convert from the user units entered into the sequence into actual DAC values and are also used by the dynamic clamping system to calibrate DAC outputs when dynamic clamping is used with sequencer outputs.

DAC units
This defines the DAC units that you wish to use. This value is provided mainly for your convenience (it is not used within the sequencer outputs system itself) but it is used by the dynamic clamping system to check and calibrate DAC outputs when dynamic clamping is used with sequencer outputs.

Waveform areas
This sets the number of arbitrary waveform output areas, you can set any number from 1 to 256.

Waveform points
This defines the length of each arbitrary waveform storage area, in points. Values from 0 to 4 million can be entered.

Output rate (Hz)
This defines the rate at which the arbitrary waveform is played out through the DACs. In conjunction with Waveform points, this sets the maximum duration of waveform replay. Values from 1 to 10 MHz can be entered. The maximum achievable rate depends on the type of 1401 and the number of DACs used.

DACs
These check boxes set which DACs will be used for waveform output. If one DAC is selected, the waveform data consists of a simple list of values; for more than one DAC the data for the DACs is interleaved starting with the lowest-numbered DAC.

# Pulses or sequencer?

You can define the outputs that Signal will generate during sampling in two different ways; by defining pulses using a graphical editor and by generating a sequence, a text file listing the operations that will be carried out in the form of a simple program. Pulses output is easier to get started with and supports multiple states directly, this form of outputs will be suitable for most users of Signal. You should consider using the sequencer only if you are unable to achieve the effect you require with Pulses output. The table below summarises the main differences between these two forms of output.

|  | Pulses | Sequence |
|---|---|---|
| Edited with | Built-in graphical editor | Built-in text editor |
| Visualise output | Yes | No |
| Stored as | Part of the sampling configuration | Output sequence `.PLS` files |
| Implemented by | Drag and drop editing | Machine code like language |
| Ease of use | Very easy to learn and use | Takes time to learn |
| Flexibility | Uses pre-set building blocks | All features available |
| User interaction | Pulse editor while sampling | Buttons trigger jumps |
| Script interaction | Add, delete and modify pulses | Variables and table data |
| Arbitrary waveform | Data in sampling configuration | Data loaded by script |
| Sweeps | Locked to sampling sweeps | Can be sweep independent |
| Timing | Several instructions per item | One instruction per text line |

Though the way in which the required outputs are defined in Pulses and Sequencer outputs are very different, the actual generation of outputs is carried out identically. When you use Pulses outputs, the pulses information is used to build sequencer data that is loaded into the 1401 and executed in exactly the same manner as Sequencer outputs. The timing requirements and limits for these two forms of output are therefore identical.

The following table summarises the use of the various output methods in the different sweep modes. More details can be found under *Pulse outputs during sampling* and *Sequencer outputs during sampling*.

|  | **Pulses** | **Sequence (triggered)** | **Free-run sequence** |
|---|---|---|---|
| **Basic** | Sampling sweep and outputs start together and are of same length. | Sweep and sequencer triggered (or just start) together so that sequence is repeatedly restarted. | Sequencer starts immediately and runs throughout. Can trigger sweeps or react to sweep progress. |
| **Peri-trigger** | Pulse outputs triggered when sweep trigger is recognised, same length as remaining sweep. | Sequencer triggered when sweep trigger is recognised, so sequence restarts at time of trigger. | Sequencer starts immediately, runs throughout. Can trigger sweeps via outputs or react to sweep progress. |
| **Extended** | Pulse outputs are triggered or free-running, can be longer than the sampling sweep, outputs trigger the sampling sweep at a defined point. | Sequencer triggered (or just starts) and triggers the sampling sweep at the required point. | Sequencer starts immediately and triggers sweeps as required. No external trigger available. |
| **Fixed interval** | Outputs (which are triggered by an internal timer) starts the sweep at a specified time. | Not available, simulate with **Extended** mode with a free running sequence. | Not available, simulate with **Extended** mode with free running sequence. |
| **Fast triggers** | Single set of outputs of same length as sampling sweep. Sweep and outputs triggered (or just start) together. | Sweep and sequencer triggered together (or just started) so that sequence repeatedly restarts. | Sequencer starts immediately, runs throughout. Can trigger sweeps itself or react to (triggered or free-running) sweep progress. |
| **Fast fixed interval** | Single set of outputs, same length as sampling sweep. Sweep and outputs triggered together by internal timer. | Not available, simulate using Fast triggers mode with free running sequence. | Not available, simulate using Fast triggers mode with free running sequence. |
| **Gap-free** | Single set of outputs, same length as sampling sweep. First sweep only can be triggered. | Sequencer re-triggered at start of each sweep so that sequence repeatedly restarts. | Sequencer starts immediately, runs throughout. Can react to sweep progress, first sweep can be triggered. |

# Automation configuration

This section of the sampling configuration dialog controls the Signal automation features. There are two areas of the dialog: Filing, which controls automatic file name generation and automatic filing, and Sampling limits, which can be used to restrict the amount of data sampled or filed. There is also a button used to access the artefact rejection dialog.

### Path

This sets the directory where the automatic file naming will look to produce a unique file name and where new files are saved when sampling has finished. This is distinct from the directory for new files that is set in the Sampling page of the Preferences dialog, that sets the location for the temporary files used while sampling. If this field is blank, automatic file name generation and file saving use the current directory which is generally not a good idea as this directory can change. You can enter a directory path directly, or use the Browse button to select a directory.
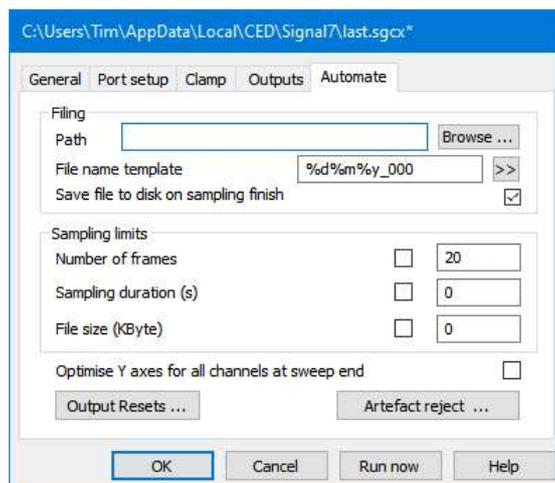
### File name template

This sets a filename template for automatic file name generation (which should not be more than 22 characters long), if left blank automatic file name generation is disabled and normal document names (Data1, Data2, …) are used for sampled data. If a template is provided, it generates a sequence of unique file names based on a numeric code. If the template ends with one or more digits, these set the length and initial value of the numeric code. If the template does not end with digits, Signal adds "000" before using it. Signal increments the code until it finds an unused name in the directory set by the Path field. Thus, a template of "testdat" generates "testdat000" to "testdat999", while "testdat10" generates "testdat10" to "testdat99".

You can insert fields holding the time and date that the data was sampled by using % followed by a character code. The button marked >> to the right of the template text can be used to insert these codes, which is easier than remembering them. The codes are:

| | | | |
|---|---|---|---|
| %T | Time as numbers (hhmmss) | %D | Date as numbers (ddmmyy) |
| %H | Hour in 24 hour format (00-23) | %d | Day of month (01-31) |
| %M | Minute as number (00-59) | %m | Month as number (01-12) |
| %S | Seconds as number (00-59) | %y | Year without century (00-99) |
| | | %Y | Year with century, e.g. 2007 |

If you use one of these codes right at the end of the template, Signal will add a '_' after the date information so that the automatic filename generation does not change the date or time information. If a file name template is set, the generated name is used automatically when the data file is saved without the user being prompted to confirm the name. A different name can be specified using the File Save As command.

### Save file to disk

If this check box is set, the new data file will automatically be saved to disk when sampling finishes. If automatic filename generation is in use, the generated filename is used, otherwise the usual prompts for a file name from the user are generated. Therefore, if you set the path, a filename template and this check box your data files will be automatically saved without your having to enter a file name at the end of sampling.

### Sampling limits

This part of the dialog controls three limits that will cause sampling to stop automatically. These are Number of Frames, Sampling duration and File size. Each option has a check box to enable the limit plus a field for entry of the limit value. If the check box for a particular limit is clear, or if the corresponding limit value is set to zero, then that limit is disabled. Note that all of these limits cause sampling to stop, not finish, sampling can still be continued after a limit is reached.

In addition to these user-defined limits, Signal has a built-in sampling limit based on the available free disk space. If the available free disk space drops below 0.5 Mbytes, sampling stops automatically. This limit cannot be disabled as it is important not to allow hard disks to get completely full, both because this can slow down file accesses considerably and also because of the trouble a full disk gives to an operating system such as Windows that uses spare disk space for virtual memory management. The user-defined limits are:

Number of frames     If this limit is enabled, sampling stops when the set number of frames have been written to the data file.

Sampling duration     If this limit is enabled, sampling stops when the set time has passed since sampling was started.

File size (Kbyte)     If this limit is enabled, sampling stops automatically when the file size reaches or exceeds the set size.

If you press the 'More' button in the sampling configuration dialog after sampling has stopped after reaching a preset limit, sampling will allow another limits-worth of sampling to occur before stopping again. So if the frame limit was set to 25, pressing More would cause sampling to continue until the data file had 50 frames. Pressing More again would give 75 frames and so on.

### Automatic Y axis optimisation

Setting the Optimise Y axes for all channels at sweep end check box will cause the Y axes of all channels to be optimised at the end of each sampling sweep. Optimisation causes the Y axes to be adjusted so that they cover the sampled data range in each channel, this gives you a better view of your data but may make it harder to see the differences between frames.

### OutputResets...

Press this button to open the dialog for the configuration output reset settings. These allow you to define DAC and digital outputs that will be set before and after any sampling, for example to ensure that a stimulator is reliably turned off.

### Artefact reject...

The Artefact reject … button opens a dialog that allows you to configure artefact rejection. This provides mechanisms for automatically rejecting or tagging sampled frames if they contain an artefact, normally caused by stimulation.

# Output Resets

In most cases, when sampling is not in progress the signal levels on 1401 outputs are of no interest to the researcher, but if an output is driving equipment that needs to be held in an inert state (for example a skin stimulator) it can be very important (if only for the peace of mind of the experimental subject) to ensure that the 1401 outputs are in the 'off' state before sampling starts and are reliably restored to this state when sampling finishes. This can be a particularly difficult problem if sampling stops during the middle of a sweep as the 1401 outputs will be left in whatever state they were at the point where sampling stopped. To avoid such problems, Signal provides mechanisms to define and apply forced output resets. There are two sets of forced output settings; application-wide settings which are controlled by a dialog generated by the Sampling page in the preferences and these settings in the sampling configuration dialog which are held as part of the sampling configuration information. The two sets of information are very similar; any reset level defined for an output in the sampling configuration settings overrides the corresponding application-wide settings for the same output. Note that if you do not have outputs that need to be controlled in this manner you do not need to make use of these settings.
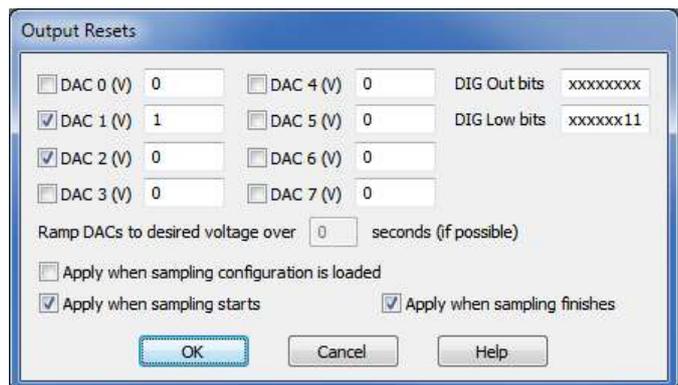
**DAC outputs**

All reset output DAC values are defined in volts, each DAC has a check box which enables the use of the forced output value.

**Digital outputs**

There are separate fields for the main digital outputs used by the pulse outputs and the DIGOUT sequencer instruction and for the lower digital outputs used by the DIGLOW sequencer instructions. Each field is eight characters long with the first character controlling the highest output bit and the last character controlling the lowest bit. An 'x' character leaves the corresponding output bit unchanged, a '0' clears the output bit (sets it to TTL low) while a '1' sets it (TTL high). So, to set the digital output 0 BNC on the front of a modern 1401 high and the digital output bit 1 BNC low, you would enter "xxxxxx01" into the `DIG Out bits` field.

**Ramp DACs to desired voltage over ...**

It may be necessary to avoid simply setting DAC outputs to the required voltage and instead ramp the output level to the correct level over a period of time. This option is not currently available but if required by significant numbers of users will be implemented in a future release.

*Apply when sampling configuration is loaded*

If this check box is set then the sampling configuration output reset settings will be applied immediately when the sampling configuration is loaded into Signal by any means.

*Apply before sampling starts*

If this check box is set then the sampling configuration output reset settings will be applied when sampling is about to begin.

*Apply after sampling ends*

If this check box is set then the sampling configuration output reset settings will be applied when sampling has finished.

# Artefact rejection

The artefact rejection dialog can be used to configure Signal to automatically examine newly sampled data and, if the data has reached the ADC limits, to reject or tag the new frame. Artefact rejection is important when generating averaged evoked responses, particularly from the EEG, where artefacts often occur and where the mathematical rigor of the averaging process will be affected by the presence of signals at the ADC limits.

The Artefact rejection mode item controls what form of artefact rejection to apply. It can be set to None for no artefact rejection, Tag frames to label frames with artefacts, or Reject frames, to discard frames with artefacts by not writing them to disk. The Artefact level is the percentage of the ADC range outside which data values will be regarded as an artefact.

The next two items set the time range for the search for artefacts; note these are specified as offsets from the start of the data rather than as absolute frame times. The final item, Allowed bad points (percent), sets the limit before the frame is rejected or tagged, allowing you to avoid rejecting frames with a trivial amount of bad data.

If artefact rejection is enabled, each frame of data sampled is scanned for artefacts. All sampled waveform channels are scanned over the time range specified and the points that exceed the Artefact level are counted. If the percentage of bad points found during the scan exceeds the limit of Allowed bad points then the frame is rejected or tagged as appropriate.
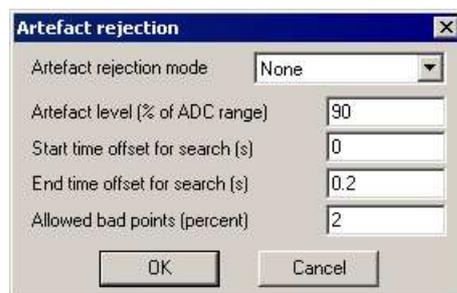
# Creating a new document

Once you have set up the sampling configuration you can create a new sampling data document. You can do this by clicking on the Run Now button in the sampling configuration dialog or the Run Now button on the Toolbar, or by selecting New from the File menu, then Data Document for the file type.

The exact appearance of the new document view varies, depending on the configuration. The name for the new document will either be `Data1`, `Data2,` and so on or a name automatically produced from the file name template in the sampling configuration. You can customise the view by adjusting the x axis, y axes, channels and other aspects of the view.

### Frame zero

A sampling document is different from all other types of data documents because it starts at frame zero, while the others start at frame 1. Frame zero is a special frame that holds the transitory data making up the sweep currently being sampled; frames 1 onward hold data that has been made permanent by being written to the disk file. Sampling is a cyclical process of collecting a new sweep of data into frame 0, deciding if it is to be written to disk and writing it if necessary, clearing frame 0 and starting the next sweep off. This process continues until enough frames have been written to disk or until sampling is stopped.

Data is shown in frame zero for as long as it is the most recent data. Frame zero is cleared when pulse outputs for the next frame start or when sampling of the next frame begins. For Basic and Peri-trigger sweeps, this means that data is displayed right up until data from the next sweep starts to be drawn. For Extended or Fixed interval sweep modes, the frame zero data will be cleared when the pulse output for the next frame starts and the new data starts to be drawn when the sampling is triggered. There may be a period when no data is shown, which provides the user with a clear indication that the next sweep has started. This is valuable because once the pulse output or sampling has started, the Accept/Reject button in the control panel cannot be used on the previous sweep of data.

For fast sweep modes (Fast triggers, Fast fixed interval or Gap-free) the most recently sampled sweep is displayed, starting when the first data points for the frame are sampled.

The sampling is controlled using a small floating or docked window; the *Sampling control panel*. This contains buttons and other controls to interact with the sampling. Sampling will not start until you click Start in this control panel (the Sample menu duplicates the panel controls). If the Event 1 start box is checked, sampling waits for an external signal after the start button is pressed.

## Online analysis

You can create memory channels and result and XY views to hold the result of analysis carried out during data acquisition in exactly the same way as when working offline; using the Analysis menu Measurements to data channel, New Memory View and New XY View commands. The dialogs through which you specify what analysis is to be performed are exactly the same as when offline, however the Process dialog that specifies what data is to be processed is different from it's offline equivalent.

This dialog controls which frames to include in the processing and how to update the memory view holding the analysis result (see under *Analysis menu* for a full description of this dialog). The most common mode is All filed frames, with an update every 0 frames (every frame). This dialog closes when you select either OK or Cancel. You can recall it with the Process command in the Analysis menu or by right-clicking in the result view.

Other variations on the standard Process dialog are provided when carrying out online processing to an XY view, particularly when the data source is a memory view which is itself being generated by online processing.

# Sampling control panel

The sampling control panel holds several buttons and check boxes used to control and interact with data sampling. When the control panel is a floating window it looks like the picture to the right, or it can be docked to any edge of the Signal application window (most often the top). Click Start to begin sampling, or Abort to give up immediately. Once sampling has started the buttons change but most check boxes are always present and can be changed at any time during sampling.

The Event 1 start check box allows you to trigger the entire sampling process externally. If this check box is set, clicking on Start will not start sampling directly, it enables the initial start of sampling on an event 1 pulse. This allows precise control of the time of the start of sampling; the time from which the frame absolute start times are measured. While Signal is waiting for an event 1 start pulse, the Start button will flash 'Waiting for E1'. The E1 input is on pin 2 of the 1401 rear panel Events connector, with a suitable ground on pin 9. Connect the E1 input to ground (or pull it below 0.6 V from above 3V) to start sampling. The polarity of this input can be inverted using the Rising edge trigger option in the General page of the sampling configuration dialog, when inverted you should disconnect the E1 input from ground (or raise it above 3 V from below 0.6V) to start sampling.

The Sweep trigger check box is initially set from the Sweep trigger item in the sampling configuration. It enables or disables triggered sweeps, while Signal is waiting for a sweep trigger the check box text will show 'Waiting (TR)'.

The check boxes in the Sweep end section of the dialog control what happens at the end of a sweep of sampling and how data is written to disk. If the Write to disk at sweep end check box is set, then each frame zero is written to the disk file when the sweep finishes. If the check box is clear, then the frame is not written - the check box text is drawn in red in this case to remind you that data may be lost. You can override this behaviour for individual sweeps using the Accept/Reject button in the sampling control panel.

The Pause at sweep end check box controls whether a new sweep is started automatically once the previous sweep has ended. If the check box is set then sampling will pause until the Continue button is pressed, or until the check box is cleared again, allowing the user to pause for a while or to inspect the data and accept or reject each frame. If the check box is clear then the next sweep starts immediately.

**Sweep end check box combinations**

| Write to disk | Pause at | Effect |
|---|---|---|
| No | No | Continuous (free-running or triggered) sweeps for signal monitoring |
| Yes | No | Continuous (free-running or triggered) sweeps written to disk |
| No | Yes | Interactive sweeps (optionally triggered) written to disk if accepted |
| Yes | Yes | Interactive sweeps (optionally triggered) written to disk unless rejected |

The sampling control panel can be hidden or shown using the Sampling menu, the Signal toolbar or the pop-up menu provided by right-clicking the mouse on unused parts of the Signal window. It can also be hidden by clicking on the **x** button at the top right of the control panel, or docked at any edge of the Signal window.

# During sampling

Once sampling has started, the sampling control panel changes to show buttons suitable for the sampling process. If Event 1 start was checked before you click Start, the word Waiting flashes until a suitable signal is applied to the E1 input to enable sampling. This is distinct from the Sweep trigger. Use this method to synchronise the start of sampling with an external event. Sampling starts within 1 or 2 µs of the external event signal. The buttons available are:

Start   This is only shown before data capture starts and after sampling has been restarted. When you click on the button, sampling of data begins. If the Sweep trigger box is checked, the 1401 system waits for the sweep trigger before starting to collect data.

Continue   This button is labelled Start before data capture starts. Its name is changed and it is enabled when sampling is paused at the end of a sweep. When you click on the button, sampling of the next sweep

is enabled. If the Sweep trigger box is checked, the 1401 system waits for the sweep trigger before starting to collect data.

More      This is labelled Start before data capture starts. Its name is changed and it is enabled when sampling has been stopped or when one of the Automate limits has been reached. Pressing it will cause sampling to resume, if it had stopped because an Automate limit was reached the limit is adjusted upwards. So if sampling had automatically stopped because 100 frames had been collected, pressing More will cause a further 100 frames to be collected before sampling again stops automatically/

Stop      This button is displayed after data capture starts. If you click on this button, the data capture stops, in the same manner as if one of the Automate limits was reached. Once the data capture has been stopped it is possible to resume sampling again using the More button.

Finish      This button is displayed when data capture has stopped but could be resumed by pressing More. If you click on this button, data capture stops completely and it is not possible to resume sampling again.

Accept      Clicking on this button writes unwritten frame 0 data to disk. If frame zero is still being collected, it overrides the Write to disk at sweep end check box so that the sweep is written to disk at sweep end; it does not affect subsequent sweeps. If the frame has been collected and sampling is paused, this writes the frame to disk immediately. The button acts upon a sweep up until the point that the next sweep is triggered or pulse outputs for the next sweep begins.

Reject      If Write to disk at sweep end is checked, or sampling is paused at the end of the sweep and frame 0 has already been written, then the label on the Accept button changes to Reject. Clicking on Reject either overrides the Write to disk at sweep end check box to cause the currently sampling frame not to be written automatically, or removes the current frame 0 data from the end of the data file, as appropriate. This button acts on a sweep up until the point when the next sweep is triggered or pulse outputs for the next sweep starts.

Abort      This button abandons sampling and discards the file. You can use this button before sampling starts or while sampling. You are warned if this will lose saved data.

Restart      This button is available once sampling starts. It stops sampling, discards any data that has been written to disk, then waits for you to start sampling again with the same document by pressing Start. You are warned if this will lose saved data.

## Other interaction with sampling

If a keyboard marker channel is being sampled, you can insert markers into the sampled data by pressing keys on the keyboard. You can do this if the new data view is the current view, or if the sampling control panel is current. The current view or window has a highlighted title bar, you can make a view current by clicking on it. If the sampling control panel is current, you should be careful about pressing the space bar, which is equivalent to pressing whichever button is currently highlighted, or pressing Enter, which is equivalent to pressing the Start/Continue button.

If you are using Peri-triggered sampling in any of the analogue trigger modes, frame zero of the sampling document displays a pair of horizontal cursors that indicate the positions of the two trigger thresholds. These cursors are displayed on the highest-numbered waveform channel (which is the last port in the ADC ports field), as this is the trigger channel. For +Analogue and Analogue trigger modes, the cursors show the trigger level and the trigger level minus (or plus) the hysteresis, this latter level is shown as the Arm level. For =Analogue trigger mode the two separate trigger levels are shown. During sampling you can adjust the trigger levels used by moving the cursors; they cannot be moved to a different channel as the trigger channel is fixed.

If you use the keyboard command Ctrl+PgUp, Signal will switch to displaying the last frame that was saved to disk.

If you are using a programmable signal conditioner such as the CED 1902 programmable amplifier then you can use the signal conditioner control panel to adjust  the amplifier settings – see *Programmable Signal Conditioners* for more details of these. Note that, if you alter the amplifier gain or offset settings, the channel calibration will be adjusted in such a way to keep the incoming data correctly calibrated – cancelling out the gain and offset changes – so you will not normally see any change in the displayed data. The Edit menu Preferences dialog has a setting (in the Sampling page) to allow the display range shown to be adjusted, if required, to reflect changes in the ADC range caused by signal conditioner changes.

You can tag data frames as they are sampled by pressing `Ctrl+T`, or frames can be automatically tagged by the artefact detection system.

Most standard display manipulation mechanisms work in exactly the same manner online, but frame overdraw mode works differently on frame zero; it never erases old data but just redraws it in grey and the frame display list is ignored. This provides a very nice 'storage oscilloscope' style display, but if any part of the view is redrawn the previous traces are lost. You can use the Edit menu Clear command to erase all the previous traces.

## Stopping sampling

Sampling can be stopped by clicking on the Stop button or by the sampling limits being reached. If a (non fast mode) sweep is in progress when Stop is pressed then sampling will continue until the end of the sweep is reached. When sampling has stopped Signal is in between sampling and finishing sampling. The sampling control panel is still present, but the button labels will change:

More
: This is the button previously labeled as Start or Continue. Click it to resume sampling as if the Stop button had not been pressed. If sampling stopped due to the frame count reaching a limit then pressing More resets the frame count and sampling runs until the count again reaches the limit, otherwise the limit is disabled and sampling continues indefinitely.

Finish
: This is the button previously labeled as Stop. If you click on this button, the data capture is terminated and the sampling control panel disappears.

Click on the Finish button to end sampling. If the sampling configuration has automatic saving to disk enabled, the new data file will be saved at this point, using either the automatically generated filename from a template or a name entered by the user as appropriate. You will also be prompted for a comment for the new file, if this feature is enabled. Once the sampling has actually shut-down, the sampling control panel is removed. In the new data view, frame zero of the data document disappears and the view changes to show the last filed frame if it was previously showing frame zero. If there are no saved frames, the data document and view are destroyed, giving the same effect as pressing Abort.

## Saving new data

A sampling document that has stopped sampling is essentially the same as a document loaded from disk. However, unless you are using automatic saving to disk, the data has not yet been saved in a permanent disk file, though it is stored on disk as a file without the .CFS file extension. To keep the data, you must save the new data using the File menu Save command. If you try to close the document view without saving the data Signal will check that you really want to do this. If you do not save the new document the data file will be deleted but, to try to protect you against accidentally losing important data, it will be moved into the recycle bin.

Data documents are always stored on disk while other document types are kept in memory until you save them. We keep data documents on disk because they can be very large. When you use the File menu New command, Signal creates a temporary file in the directory specified in the Sampling page of the Edit menu Preferences dialog. If you do not specify a directory, the location of the temporary file is system dependent. When you save a new data document after sampling, Signal moves it to the directory you specify.

## Saving configurations

To avoid setting the sampling, analysis and screen configuration each time you sample, you can save and load sampling configurations from the File menu. The configuration includes:

- The basic sampling parameters in full, including port information for all ports.
- The position of all windows associated with the new file (including duplicated windows and the various control panels) plus all windows generated by processing.
- The displayed channels and display modes of the channels in the windows.
- The outputs to be generated during sampling if pulse outputs are used - sequences are saved in separate .pls files.
- The  multiple states information, including the protocols.
- The processing parameters and update modes of all memory views.

If sampling ends without failing or being aborted, then Signal will save the sampling configuration that was used as `last.sgcx`. This last-used sampling configuration is used the next time data is sampled unless a default sampling configuration is found, which overrides this. When Signal starts, it searches for and loads the configuration file `default.sgcx`. If this cannot be found, it uses `last.sgcx` so generally you will get the same sampling setup as was used last time. If sampling fails or is aborted the `last.sgcx` file is not updated and the sampling configuration switches back to the configuration in use at the time you started sampling.

Both of these files are saved in the current user's application data directory but in order to be as compatible as possible they are searched-for in both the current user's application data directory and the application installation directory.

It is a good idea not to rely on `last.sgcx` as this may have been changed by someone else doing some sampling, instead you should save important sampling configurations to disk using suitable names and reload them before starting an experiment. You can make this much easier by using the Sample Bar which holds sampling configurations so that they can be quickly loaded using a single mouse click.

## Sequence of operations to set and save the configuration

This describes a sequence of operations that build a new sampling configuration from scratch. You will find that once you have built a few configurations, it is simpler to load an existing configuration and change the sections that do not fit your requirements, rather than re-build entirely. The steps are:

1.  Open the Sampling Configuration dialog using the Sampling menu or toolbar and set the sampling configuration, limits and port information.
2.  Press Run Now from the configuration dialog or press OK and select the New command in the File menu and choose Data Document.
3.  Arrange the new file view as you require and add or remove duplicate windows.
4.  Use the Analysis menu New Memory view and Measurements view commands to add result views, XY views and memory channels as required and set their update mode and position on screen.
5.  You can use the File menu Save Configuration As command to save the configuration to disk at this point.
6.  Sample, adjusting any positions, display configurations and so forth as required. Again, you can save the configuration as adjusted at this point.
7.  If sampling has finished without being aborted or failing, the sampling configuration is held in memory for use the next time you sample. You can use the File menu Save Configuration As command to save the configuration used to disk.

You can re-use a saved configurations by loading it with the File menu Load Configuration command before you use the File menu New command to create a sampling document.

# Pulse outputs while sampling

Signal can generate pulse and waveform outputs from your 1401 during sampling using the DACs and the digital outputs.

Signal pulse outputs provide a simple way of configuring outputs that will be generated during sampling. Pulse outputs, like Signal data acquisition, is arranged as fixed-length frames. Depending upon the sampling sweep mode, the pulse outputs may be the same length as the sampling sweep, longer, or shorter (Peri-triggered mode only). In all circumstances, the outputs are fixed in time relative to the sampling sweep. In Basic mode and the three fast-trigger modes (Fast triggers, Fast fixed interval and Gap-free) the pulse outputs start at the same time as the sweep (triggered or un-triggered) and is of the same length. In Peri-triggered mode, the pulse outputs start at the time of the trigger (which can be before or after the start of the sampled data, depending upon the pre-trigger points), and again runs to the end of the sampling sweep. In Extended and Fixed interval sampling modes, the pulse outputs can be any length greater than or equal to the sampling sweep and the sampling sweep starts at a defined point within the pulse outputs.

## Pulses dialog

If you press the Configure Pulses button (only available with Pulses outputs selected) in the outputs page in the sampling configuration, Signal will display the Pulses dialog which allows you to view and edit the pulses.

This dialog can also be used to control and adjust the pulses while Signal is sampling, in which case it is accessed using the sampling menu, the Signal toolbar or by using the right mouse button popup menu.

The Pulses configuration dialog can be used to define square pulses and pulse trains, ramps, sine waves, markers and arbitrary waveforms which will be generated during sampling. Many of these pulses can automatically vary by incrementing their amplitude or duration by fixed amounts. This provides a straightforward way of generating a repeating set of pulses from one definition. If you are using Extended frame or Fixed interval sweep modes, this dialog is also used to set and control the length of the pulse outputs, the start of the sampling sweep within the outputs period, the sampling sweep ADC points if Variable points are enabled, and the fixed interval sweep interval and interval variation.

You can resize the Pulses dialog by dragging the bottom right-hand corner of the dialog.
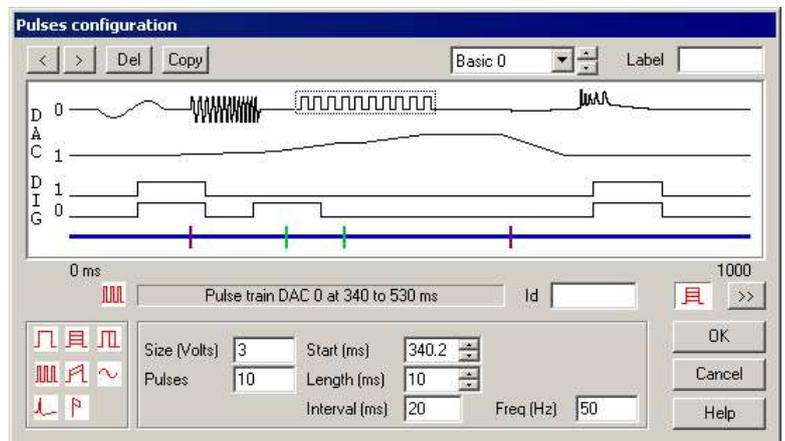
## Pulses dialog layout

The dialog is divided into a number of sections:

### Pulse and state controls

These controls occupy the topmost area of the dialog and provide buttons used to interact with the dialog, information about any timing problems and a state selector and state label for multiple states use:

These buttons are used to toggle through the pulses on the current output trace in forwards or reverse time order, selecting each pulse available in turn.

This button is used to delete the currently selected pulse.

Copy    This button is only enabled when multiple states controlling dynamic outputs are in use (see *Sampling with multiple states*). It is used to copy sets of pulse data from the currently displayed state to other states. See below for more details on using this.

Timing faults: 3

This area shows if there are any timing problems with your pulses, it will be blank if there are no problems. Timing problems are caused by the fact that the Signal outputs system can only do one thing within each time step, a non-zero timing fault count indicates that at some point or points the system is trying to do more than one thing at a time, cannot do so and therefore an action will be delayed. Timing faults will not cause sampling to fail, they just indicate that the pulse timings will not be perfect, to get rid of them you will have to alter your pulse times. Red dots are drawn at the bottom edge of the pulse display area (below the blue control track) to indicate where the timing problems are happening - you can see three of them in the Pulse display area picture below.

Basic 0    This pair of controls is only visible when multiple states controlling dynamic outputs are in use (see *Sampling with multiple states*). They are used to select the state shown by the dialog: you can only display and edit pulses for one state at a time.
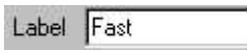
Label Fast    This control is also only visible when multiple states controlling dynamic outputs are in use (see *Sampling with multiple states*). It is used to set a label for the currently displayed state; a descriptive name for the state up to 10 characters long that will be used in the state control bar buttons and elsewhere within Signal. If you leave this field blank the standard state names "Basic 0, State 1 .." will be used.

### Pulse display area

This occupies most of the upper part of the dialog and shows the currently defined outputs as graphical traces. At the bottom of the display area is a blue line, the control track, that is used to hold special pulse items and provide access to special information. The control track line also indicates what portion of the pulses output period that



is covered by the sampling sweep, the line is thick where sampling is happening (if you are using Extended or Fixed interval sweeps) and thin where no sampling is taking place.
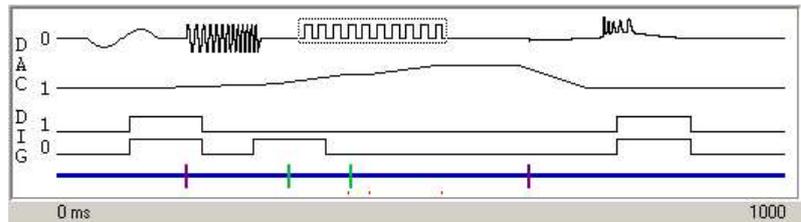
Numbers indicating the start and end times of the pulses are shown just below the pulse display area itself. If you are dragging a pulse about an extra indicator appears here showing the current drag time.

The outputs themselves are displayed starting with DAC zero at the top and finishing with the digital outputs at the bottom, with the higher-numbered digital outputs first. Only traces for enabled outputs are displayed, they are labeled with the DAC or digital output bit number. If you double-click on a trace in the display area it will expand to use all of the space available for traces, hiding all other outputs. Double-clicking on the display a second time will restore the display of all traces. If you stretch the dialog by dragging a corner or edge, the pulse display area will expand to fill the space available and give you more space to see what the pulses are doing.

This display also includes a dotted rectangle drawn around one of the pulses, or around the initial level of an output. This indicates the currently selected pulse whose numerical parameters are displayed at the bottom of the dialog (see the *Values* section below). You can select a pulse by clicking on it with the mouse. Alternatively you can toggle the selection through the various pulses on an output using buttons in the *Controls* section. You can change the start time of a pulse by dragging it around the display area.

### Review controls

These are at the right of the central area of the dialog, just below the numbers showing the start and end times of the pulse display area. The controls are:

Click on this control to see the effect of pulse variations. While you hold down the mouse button on this control, the limits to pulse variations plus three intermediate values are displayed.

Click on this button to animate the display to show the effect of pulse variations. The display updates to show the effect of each variation in turn. Click again on the button to turn animation off.

### Pulses toolbox

This is the area at the bottom left of the dialog. It holds pulse icons which can be dragged into the display area to add a new pulse into the outputs. Each icon represents a different type of pulse:

a square pulse without variations, digital or DAC.

a square pulse with varying amplitude, DACs only.

a square pulse with varying duration, digital or DAC.

a train of square pulses without variations, digital or DAC.

a ramp pulse with varying amplitude at start, finish or both, DACs only.

a sine-wave without variations, DACs only.

an arbitrary waveform on one or more DACs.

a digital marker generation item, this is not an output but is added to the sampled data.

### Pulse values

This is the area at the bottom centre of the dialog, it holds the parameters defining the currently selected pulse which can be changed by editing the parameter values. The parameters shown vary according to the type of pulse, the box title shows the type of pulse and the start and end times for the pulse, again using the currently selected time units. The details of the parameters are covered under the individual pulse types. If the control track is the current selection this area will show other parameters according to the sampling mode, such as the length of the outputs, the time of the sweep trigger , the fixed interval times and the sampling sweep points.
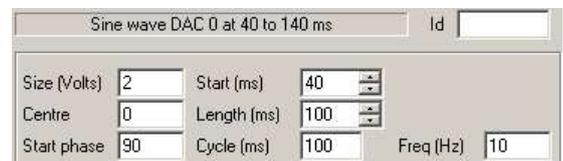
The Id field at the top right of the values area displays the name for the currently selected pulse, which can be edited as desired. The main reason for giving a pulse a name is to make it easy to access the pulse from the script language.

# Dragging and dropping

A number of operations in the pulse configuration dialog are carried out by dragging and dropping. A major advantage of drag and drop is its graphical, visual nature. For this reason, it is increasingly commonly found as a way of carrying out operations in Windows software.

To drag and drop a screen object place the mouse pointer on top of the object, then press and hold down the left mouse button. The mouse pointer may change to indicate the start of a drag operation and an icon may be attached to the pointer to indicate that the object has been 'grabbed'. Still holding the mouse button down, move the mouse pointer to drag the object to its required destination. The mouse pointer may change while you do this, a common effect is for the pointer to change to a no entry sign (a circle with a diagonal line through it) to indicate that you cannot drop the object at this point. When the object is at the required destination, release the mouse button to drop it in place.

# Adding a new pulse

To add a new pulse into the outputs, select the pulse required from the pulses area of the dialog. Drag the new

pulse from the pulses area into the display, the mouse pointer changes to a ⊕ (a hand with a plus sign) to indicate that you are adding a new pulse. As you move the mouse pointer about the display, a vertical line indicates where the new pulse will go and the drop time is displayed in the control area. Once the pulse is correctly positioned, drop it into place by releasing the mouse button.

If you cannot get the display area to accept the new pulse (the mouse pointer is always No Entry), this could be because of the following reasons:

- You are trying to drag a DAC-only pulse such as a ramp, sine wave or waveform into the digital outputs.
- You are trying to drag an arbitrary waveform or  marker generation item onto an output track, these can only go on the control track.

# Moving a pulse

To move a pulse, select the pulse in the display area. Then drag the pulse into the new position required (you are allowed to drag a pulse onto a different output as long as you do not change from DAC to digital or vice-versa).

The drop position is shown as for adding a new pulse, the mouse pointer changes to ⬚ (a hand) to indicate taking hold of something without addition or removal.

If the Ctrl key is held down during the drag operation the mouse pointer will change to ⬚ *(a hand with a + in the centre to indicate addition) and the pulse will be copied instead of moved.

To avoid problems with precise positioning of the mouse Signal does not recognise a drag-and-drop operation until the mouse has moved a certain amount away from the initial position. The mouse pointer shows this by only including the pulse icon and showing the drop position when the amount of movement is sufficient. You can give up on a move by returning the pointer close to the initial position. If you want to move a pulse a small amount, but find that Signal will not recognise a drag operation that short, move the mouse pointer a larger amount vertically to convince Signal that you mean it and smaller horizontal movements will be accepted.

If you cannot find or click on the pulse to start dragging it (usually because it is too short or completely hidden by another pulse), see *Finding a pulse* for how to select it. Once the pulse is selected, you can change the start time directly by editing it in the values area.

# Removing a pulse

To remove a pulse, select it in the display area, then drag the pulse out of the display area completely. The mouse pointer changes to ⬚ (a hand with a minus sign) to indicate a remove operation once you are outside the display area. Drop the pulse to complete the removal.

Alternatively, once the pulse has been selected, you can remove it by clicking the Del button in the controls area. If you cannot find or click on the pulse to start dragging it, see *Finding a pulse* for how to select it.

# Finding a pulse

It may be difficult to see a pulse or to click on it because the pulse is too short to see or because it is hidden by another pulse. Click on the appropriate output trace, then use the < > buttons to toggle through all of the pulses on that output. At the appropriate point, your hidden pulse will be selected and can then be edited or deleted directly.

# Copying pulses

You can copy a single pulse to the same or a different output by holding down the Ctrl key while dragging a pulse, without the Ctrl key this moves a pulse.

When working with the multiple sets of pulses provided by multiple states with dynamic outputs, it is often useful to copy a set of pulses or settings to create duplicated data for other states. The Copy button opens a dialog where you can specify items within the currently selected state to be copied and a range of destination states to copy them to. The left-hand side of the dialog sets the items that will be copied, the right hand side sets the destination, with check boxes for each DAC and digital output plus another for the control track that select the pulse items to be copied. If the Length, trigger check box is checked, the outputs length, the sweep trigger time within the outputs and any variable points settings are copied from the currently selected state into the target states, otherwise these values are left unchanged. Similarly, checking the Fixed interval check box causes the fixed interval and interval variation values to be copied. For the DACs and digital outputs selected, all existing pulses in the destination states are deleted before the new data is copied in.

# Pulse types

The pulses dialog can handle a number of different pulse types, both varying and not, some of which can only be used on DAC outputs or are attached to the control track:

a square pulse without variations, on digital or DAC outputs.

a square pulse with varying amplitude, on DACs only.

a square pulse with varying duration, on digital or DAC outputs.

a train of square pulses without variations, on digital or DAC outputs.

a ramp pulse with varying amplitude at start, finish or both, on DACs only.

a sine-wave without variations, on DACs only.

an arbitrary waveform on one or more DACs, the pulse item is on the control track.

a digital marker generation item on the control track only.

in addition to these there is extra information setting the initial level of an output. For the control track, this sets extra information about the outputs, for example the outputs length and the sweep interval in Fixed interval mode.

# Initial level

This item specifies the state that the outputs are set to at the beginning of the pulse outputs, this item is always present and cannot be deleted or moved.

The item has a single parameter; Level, that sets the level that the DAC is initially set to. The level entered is scaled before use as defined by the DAC settings in the output page. The three other parameters at the bottom of the values area; Step change, Repeats and Steps, are used to define a built-in variation in the initial level. Built-in variations are described under *Pulses with variations*.
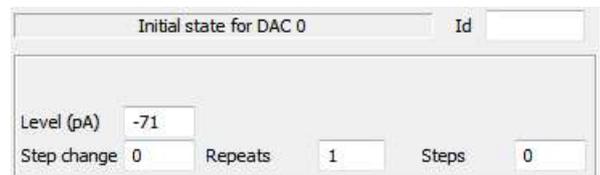
For digital outputs only a single initial level parameter is available, set this to 1 for a high output and 0 for low.

### Control track values

The initial level item for the control track provides other parameters that control the overall outputs, these available vary with the sampling sweep mode, they are provided as part of the pulses outputs setup because they can vary with the state in multiple states sampling and can interact with the outputs in some cases.
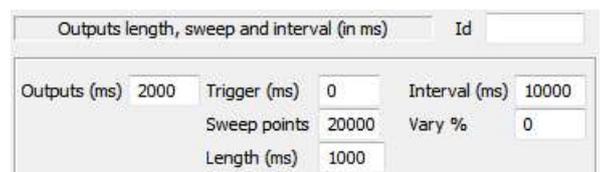
When using Extended and Fixed interval sweep modes, the outputs length and start time for the sampling sweep are shown. The Outputs parameter sets the length of the outputs, while the Trigger parameter sets the time, relative to the start of the pulse outputs, of the start of the sampling sweep.

With Fixed interval mode two more parameters; Interval (s), and Vary (%) appear (with Fast fixed interval mode the Interval field is available but not Vary). These set the timed interval between output frames (which cannot be less than the pulse output frame length) and the percentage random variation in the interval, from 0 to 100, for the selected state. If the variation is non-zero, the frame interval used while sampling will vary randomly from Interval-Vary% to Interval+Vary%.

If variable sweep points are enabled (only available with Extended and Fixed interval modes when multiple states are used), other fields are provided to set the number of points sampled per channel for the selected state. This ADC points field can be set to any even number less than or equal to the sweep points value set in the General section of the sampling configuration, the Length field does the same but in terms of the sweep duration.

# Simple square pulse

This specifies a square pulse without any variations. This is the simplest type of pulse and is available for DACs and for the digital outputs, in addition to generating simple pulses multiple items can be used to generate simple pulse trains or biphasic pulses. Many of the parameters used for this type of pulse are also used for all other types of pulse, to save space these common parameters are described in detail once only.

The Size parameter sets the pulse size, in calibrated units as defined in the outputs page. If absolute pulse levels are in use, the item changes to Level, and is the level that the pulse goes to, for relative levels the size is added to the level prior to the pulse to get the pulse level. Either positive or negative values can be used.

For digital pulses, the Size parameter disappears; for outputs with a low initial value a high-going pulse is produced and vice-versa. The use or otherwise of absolute pulse levels does not affect digital pulses and overlapping pulses do not invert each other.

The Start parameter sets the start time for the pulse. If absolute times are not in use, this parameter changes to Delay and sets the delay from the start of the previous pulse to the start of this pulse. The Length parameter sets the length of the pulse. Both of these fields will use the time units selected in the Display page of the Preferences dialog.

If the No return check box is checked, the pulse does not return back to the initial level when it ends but just stays at the pulse level, giving us a single step-change. In this circumstance the length parameter has no effect.

# Varying amplitude pulse

This specifies a square pulse whose amplitude varies as it is used. The Size, Start and Length parameters are exactly the same as for the simple pulse. In addition there are three parameters controlling the built-in variation. This type of pulse is not available for digital outputs. The behaviour of the built-in variation is described under *Pulses with variations*.

# Varying duration pulse

This specifies a pulse whose amplitude is constant but the pulse length changes as it is used. This is the only pulse with a built-in variation that is available for the digital outputs.

The Size, Start and Length parameters are exactly the same as for the simple pulse. The other parameters control the variation, with the exception of the Push back check box. If this is checked, increases in the pulse duration delay the start of following pulses by the same amount. Decreases in the pulse duration move the following pulses earlier in time. If the check box is clear, changes in the pulse duration do not affect the time of following pulses. The behaviour of the built-in variation is described under *Pulses with variations*

## Square pulse train

This specifies a series of non-varying square pulses. This type of pulse is available for DACs and for the digital outputs.

The Size, Start and Length parameters are all exactly the same as for the simple square pulse. The extra parameters are Pulses, which sets the number of pulses in the train, Interval which sets the pulse spacing - the spacing between the start of two adjacent pulses, which should not be less than Length, and Freq (Hz) which allows you to set the pulse spacing in terms of pulses per second. Changes made to Interval are reflected in Freq and vice-versa.

## Ramp with varying amplitudes

This item specifies a pulse with different start and end amplitudes so that the top of the pulse can be sloping. The variation in amplitude, if this is used, can be applied to either the pulse start or end, or both.

The Start and Length parameters are exactly the same as for the simple pulse, the size parameter is also called Start, but shows the DAC units to avoid confusion. The extra parameters are End, which sets the pulse level at the end of the pulse and a selector for the variation which can be set to Step both, Step start or Step end. This is an extremely versatile form of pulse; for example by setting the start size to zero you can produce a ramp running from one level to another.

## Sine wave

This item specifies a cosine wave output of fixed duration, amplitude and frequency for output on a DAC. For Signal version 3, sine wave output can be generated on all DACs. In earlier versions only DACs 0 and 1 could be used.

The Size parameter sets the amplitude of the cosine wave (the distance from the mid-point to the extreme value, so the total amplitude will be Size*2). The other parameters are Centre, which sets the level about which the cosine oscillates, Start phase, which sets the initial phase in degrees, Cycle (s), which sets the duration of one complete cycle and Freq (Hz) which allows you to set the cycle period in terms of cycles per second. The Centre value is an absolute or relative voltage level as required. Because the output is actually a cosine wave an initial phase of 90 degrees will start the output off at the centre level.

## Digital marker

This item specifies a digital marker generation item with the marker code either set by the pulse item or read from the digital inputs. Digital marker generation items can only be placed onto the control track (as they do not apply to an output) where they are shown by a green tag.

The Start parameter sets the time for the digital marker generation. The Code parameter sets the marker code value unless the Read inputs check box is set, in which case the code value is read from the 1401 digital inputs.

# Arbitrary waveform

This item specifies arrays of data to be output to one or more DACs at a specified rate. Output to each DAC starts simultaneously and consists of the same number of points, so the output also finishes simultaneously. An arbitrary waveform item can only be placed upon the control track, where it is shown by a purple tag, it also shows the waveforms on the relevant DACs.

| | | | | |
|---|---|---|---|---|
| Waveform on DACs 01 at 650 to 902 ms | | | Id | |
| DAC list | 01 | Start (ms) | 650 | |
| Rate (Hz) | 5000 | Points | 1258 | |

Adding a waveform item to the pulse outputs is a two-step process:

1. First you add the arbitrary waveform item to the control track, which creates the item with a waveform that is all zeroes. Once the waveform item has been created you can adjust the DACs used, the waveform output rate and the number of points to get the parameters correct - it's particularly worthwhile to set up the DAC list at this point.

2. Once the basic waveform item is set up, you can then put the waveform that you want to use into it. This is most easily done using the clipboard; if you copy Signal data to the clipboard you can then paste it into your output waveform(s). To do this, select your new waveform item and press Ctrl+V on the keyboard or right-click on the item and select Paste. Signal then provides the Paste waveform dialog which you can use to control what data is pasted and which DAC outputs it is pasted into, along with other aspects of the paste operation. You can also use the script language to copy data from a script array into the output waveform(s).

The DAC list parameter specifies which DACs are used, it is a list of DAC numbers from 0 to 7, this list is always sorted so if you enter "620" it will be redisplayed and used as "026". The Start parameter sets the start time for the waveform output in the usual manner, while the Rate (Hz) parameter sets the output rate for the data points for each DAC and the Points parameter sets the number of data points output for each DAC.

The maximum output rates possible vary according to many factors such as the ADC rate and the pulse output timing resolution. Tests carried out at CED with two channels of ADC data sampled at 10 KHz show that waveform rates of up to 275 KHz can be achieved with a micro1401 and 500 KHz is possible (with difficulty) when using a modern Power1401. The Maximise wave rates check box in the Outputs page of the sampling configuration dialog can be used to increase the waveform output rates achievable - with a small penalty of increased timing jitter in all sequencer operations.

### Waveform output scaling

Waveform data on the clipboard is held as floating-point numbers exactly as seen in the Signal data display. When this data is copied into a waveform output item it is scaled using the individual DAC scale factors as set up in the main sampling configuration Outputs page. This conversion does not pay any attention to the units defined for either the source waveform data or the DAC outputs. So if you copy waveform data with mV units where the data values range from -200 to +200 mV and paste it into a DAC scaled to generate outputs from -5 to +5 volts, Signal will try to generate an output running from -200 to +200 volts, not -0.2 to +0.2 volts. As the DAC scaling only allows values from -5 to +5 volts your waveform will be severely truncated. You can fix this either by adjusting the DAC scale factors, or by rescaling the data before you put it onto the clipboard. The easiest way to rescale the data is to use a virtual channel with an expression such as "ch(1)*2.5" as this allows you to adjust the scaling until it is correct. Once the waveform is as you want it you can copy the virtual channel data onto the clipboard and paste it into the output waveform. When setting output waveforms using the script language an identical scaling process is carried out if you are using an array holding floating-point values, values in an integer array are used directly with 32767 corresponding to the upper full-scale DAC range limit and -32768 corresponding to the lower DAC range limit.

### Multiple waveform items

You can define an unlimited number of arbitrary waveform items but there are restrictions; both the DAC list and the waveform output rate must be the same for all waveform outputs in a given set of outputs (waveform outputs for different states can be different).

# Pasting a waveform

Copying data to the clipboard in Signal also places the visible view data onto the clipboard in a private format which can be used to set the waveforms for outputs. To place suitable data on the clipboard, open a data file and adjust the display so that the required time range and channels are visible, then use the Edit menu Copy command.

You can then paste this data into a waveform item in the pulse outputs configuration dialog by selecting a waveform output item and pressing `Ctrl+V` (the Paste command shortcut) or right-clicking on the dialog and selecting Paste. Signal then provides a dialog to control the paste so that you can ensure that you paste the right data into the waveform and to control other options.

The upper part of the dialog describes the data on the clipboard and provides control of which data to be taken from the clipboard - you specify the first channel whose data will be used (channel numbers start at 1 from the first waveform channel that was pasted; the actual channel numbers for the data that was copied are not used) and the start data point offset for each channel to be taken from the clipboard. An offset of zero takes data starting with the first point on the clipboard, larger offsets cause points at the start of the clipboard data to be skipped. The same offset is used for all pasted channels.
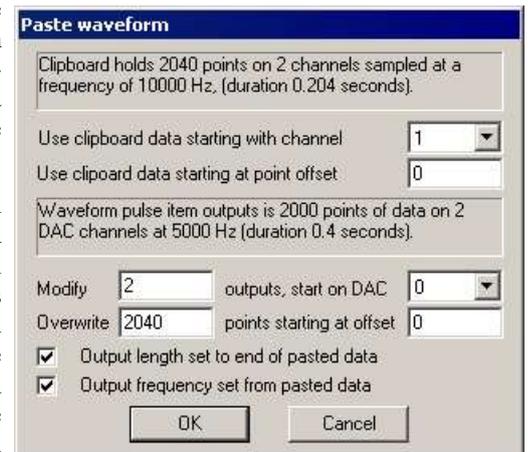
The lower part of the dialog describes the current waveform output parameters, and provides control over where the data taken from the clipboard is put to what extent the waveform output parameters are modified to match the clipboard data. The upper pair of controls, making up the line Modify x outputs, start on DAC n, defines the channels of the waveform output to modified. The DAC selector will only show DACs that are in the DAC list for the waveform output item that you are pasting-into, so it is a good idea to set the DAC list correctly before pasting. If you are changing more than one DAC output, DACs in the DAC list starting with the DAC specified are changed. The lower pair, making up the line Overwrite x points, starting at offset y, sets the amount of data that is pasted and where in the waveform buffer it is put.

The Output length set to end of pasted data and Output frequency set from pasted data check boxes act as their titles imply. If all data offsets are set to zero, the points to overwrite is set to the points on the clipboard and these two check boxes are checked, the paste operation copies all the clipboard data and changes the waveform output parameters to match. Click OK to paste the data, or Cancel to do nothing.

# Pulses with variations

Some types of pulse can be set up so that they vary automatically. The pulse types that support this are initial level, square pulse with varying amplitude, square pulse with varying duration and the ramp pulse. All of these use the same three parameters to control the variation, only the varying aspect depends upon the type of pulse. The pattern of variation used is: the pulse is generated a number of times without variation, then a number of times with one 'step change' added, then two steps and so on. This repeats until the maximum number of changes has been reached, at which point the cycle restarts with the pulse with no step changes.
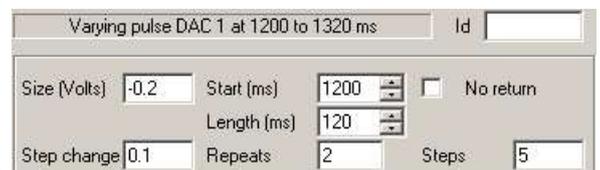
The Step change parameter sets the amount by which the varied aspect (the pulse amplitude for example) changes at a time. The Repeats parameter sets the number of times each step is repeated before moving on to the next step and the Steps parameter sets the maximum number of changes to be added. This arrangement gives a final value of Initial + (Steps * Step change). The total number of pulse forms generated is one more than Steps as the variation includes the basic pulse without any step changes.

In the example shown, seven pulses will be generated with amplitudes of 30, 20, 10, 0, 10, 20 and 30 mV. Each pulse will be generated twice in the order shown; the entire sequence will repeat after 14 pulses. If you wanted a sequence that ran 30, 20, 10, 0, 10, 20,and 30, you would set the pulse amplitude to 30 and the step change to 10.

# Extended and Fixed interval sweep modes

With Basic, Peri-triggered, Fast triggers and Fast fixed interval sweep modes, the length of the pulse outputs is set by the sampled data (Basic, Fast triggers and Fast fixed interval modes have the outputs length the same as the sweep length, Peri-triggered mode has outputs length as the sweep length less any pre-trigger time). With Extended and Fixed interval sweep modes, the outputs length is set independently of the data frame and data

acquisition starts at a preset time within the outputs. Because these values can vary with the state in multiple states sampling, for convenience they have been located in the pulses dialog.

When using Extended mode the length of the outputs and the sweep start time values are shown whenever the control track is selected. The Outputs field sets the length of the outputs, this value must be at least as long as the sampling sweep. The Trigger parameter sets the time, relative to the start of the pulse outputs, of the start of the sampling sweep.
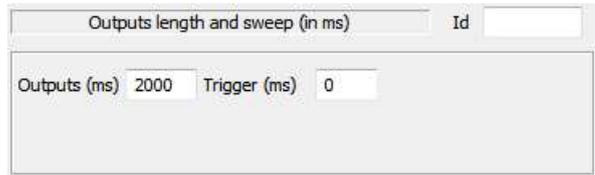
With Fixed interval mode two more control track parameters appear; Interval, and Vary (%) (with Fast fixed int mode the Interval field is available but not Vary). These set the timed interval between output frames (which cannot be less than the pulse output frame length) and the percentage random variation in the interval, from 0 to 100, for the selected state. If the variation is non-zero, the frame interval used while sampling will vary randomly from Interval-Vary% to Interval+Vary%.
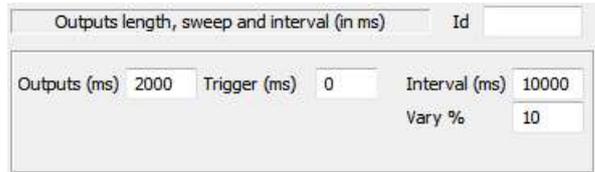
### Variable sweep points

If variable sweep points are enabled (only available with Extended and Fixed interval modes with multiple states in use), extra control track parameters are provided to control the sweep points. The Sweep points item sets the number of points sampled per channel for the selected state, this can be set to any value less than or equal to the overall maximum sweep points value set in the General section of the sampling configuration, and must be an even number. The Length parameter controls the same sweep points value in terms of the sweep duration.
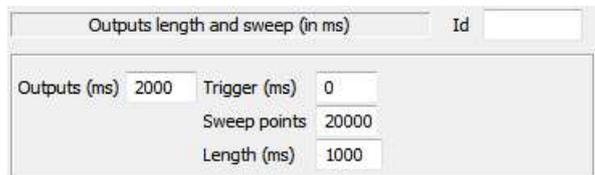
# Controlling pulse outputs during sampling

While sampling is in progress, the pulses configuration dialog can be obtained by using the Sampling menu, by clicking on the relevant toolbar button or by right-clicking on spare application areas to get a popup menu. All three mechanisms provide the pulse configuration dialog in the same form as used offline. The only difference is that the OK button has been renamed Apply and the Cancel button Close and that pressing Apply doesn't cause the dialog to disappear.

You can use the pulse configuration dialog freely during sampling to change the pulses that are output, the only restriction being that you cannot increase the size of any arbitrary waveform items, or add an arbitrary waveform item where one was not present beforehand. All changes made will be copied into the sampling configuration if the sampling completes successfully so that they can be used next time.

### Timing warnings

With Fixed interval or Fast fixed interval mode in use, the use of external sweep triggers is disabled as the interval timer does all of the triggering. If the delay between the end of one pulse frame and the start of the next is too short, the internal trigger may be missed, if this occurs a warning message is generated at the end of data acquisition.

# Sequencer outputs during sampling

The Signal output sequencer is available as an alternative to Pulses for generation of outputs during sampling. It can generate both digital and analogue signals and can be used to respond to external signals and to control data capture.

An output sequence is a list of up to 8191 instructions (2047 for older 1401 types) that are executed by the 1401 during sampling at a constant, user-defined rate to produce the outputs required. Sequences are defined by a sequence file, which is a text file with a `.pls` file extension. The Signal sequencer has the following features:

- It controls digital output bits 15-8 to produce precisely timed digital pulse sequences. In Power1401s and Micro1401s it can also control the 1401 digital output bits 7-0.
- It controls the 1401 DACs (Digital to Analogue Converters) to produce voltage pulses and ramps.
- It can play cosine waves at variable speed and amplitude through the DACs.
- It can test digital input bits 7-0 and branch on the result.
- It supports loops and branches, can randomise delays and stimuli and generate digital marker items.
- It has 256 variables (`V1` to `V256`) that can be read and set by on-line scripts.
- It supports a user-defined table of values for fast information transfer from a script.
- It can read the latest value from a waveform channel and, using this information, provide real-time (fractions of a millisecond) responses to input data changes.
- It can set or get the sweep state code, get the start time of the sweep, wait until a specified time within the sweep or trigger a sweep.
- It can control and monitor arbitrary waveform outputs.

You write sequences with a text sequence editor; each text line generates one instruction.

## Sequencer technical information

The sequencer clock starts within a microsecond of recording time zero, the time at which the Start button is pressed, and is time locked to the 1401 sweep timing and waveform channel recording. Each clock tick books an interrupt to run the next sequencer instruction and updates digital output bits 15-8 if they were changed by the previous instruction.

An interrupt is a request to the 1401 processor to stop what it is doing at the earliest opportunity and do something else, then continue the original task. The time delay between the interrupt request and the instruction running depends on what the 1401 is doing when the clock ticks and the speed of the 1401. This delay is typically a few microseconds, so instructions do not occur precisely at the clock ticks but changes to digital output bits 15-8 do. Changes made by the sequencer to the 1401 DACs and digital output bits 7-0 occur a few microseconds after the clock tick.

The table shows the minimum clock interval, the approximate time per step and the extra time used for cosine and ramp output for each 1401. Notice that the first item is in units of milliseconds (to match Signal) and the remainder are in microseconds.

|  | Power-3 | Power mk II | Power | Micro3 | Micro mk II | micro1401 |
|---|---|---|---|---|---|---|
| Minimum tick interval | 0.004 ms | 0.006 ms | 0.01 ms | 0.01 ms | 0.025 ms | 0.10 ms |
| Time used per tick | <1 µs | <1 µs | <1 µs | ~1 µs | ~1 µs | <8 µs |
| Cosine penalty/tick | 0.2 µs | 0.3 µs | 0.55 µs | ~1 µs | ~1 µs | ~5 µs |
| Ramp penalty/tick | 0.1 µs | 0.25 µs | 0.5 µs | 0.7 µs | 0.7 µs | ~3 µs |
| DIV/RECIP penalty | 0.2 µs | <1 µs | <1 µs | <3 µs | <3 µs | <10 µs |

The Minimum tick interval is the shortest interval we recommend that you set. The Time used per tick is how long it takes to process a typical instruction. The Cosine penalty/tick is the extra time taken per cosine output. The Ramp penalty/tick is the extra time taken per ramped DAC, the DIV/RECIP penalty indicates the extra time taken up by these instructions. Time used by the sequencer is time that is not available for sampling, transferring data back to the host or arbitrary waveform output. To make best use of the capabilities of your 1401 you should set the slowest sequencer step rate that is fast enough for your purposes.

If you overload the system so much that the output sequencer cannot keep up, Signal will warn you of this when sampling finishes. If the 1401 is extremely overloaded by the combination of sampling and sequencer output, sampling will fail with an appropriate error message.

The clock interval that you set is rounded to the nearest 4 microseconds to generate the actual sequencer tick interval. For modern 1401s (Power1401s and Micro1401s mk II and -3), intervals less than 5 milliseconds are rounded to the nearest 0.1 microsecond. All subsequent timing is based upon the rounded interval so that full accuracy is maintained.

### 1401-specific limitations

On the micro1401 and Micro mk II, the limit to the number of sequencer instructions is 2047, all other types allow 8191.

# The sequence editor

The output sequence is stored as a text file with the extension `.PLS`. You can open existing Signal output sequence files or create new ones with File menu New. There are five buttons at the top of the window:

### Compile

This checks the sequence to make sure that it is correct with no labels missing or duplicated and no duplicated key codes. The picture shows a sequence with a simple error (the `LB` in the seventh line should be `LP`). The line in error is marked and an explanatory message is shown at the top of the window.

### Format

This aligns the labels, key codes, instructions and any arguments, output text and comments and removes step numbers. Undefined labels are not flagged but there must be no other errors.

### Format with step numbers

This does the same job as the Format button, and also starts each line with the step number. Step numbers can be useful as they give an indication when you are running out of space and can pinpoint the line where your sequence is not behaving as you expect.

### Current

This compiles the sequence and, if the sequence is correct, saves it and makes this the current sequence for use during sampling. If you were to open a new data file and start sampling, this sequence would be used. You can also set the output sequence file from the Sampling Configuration dialog.

### Help

This is the Help button. It opens a window holding a list of the sequencer topics for which help is available. You can copy and paste text from the help window into your sequence. You can also obtain online help on a topic by clicking on the relevant text and pressing the F1 key.

# Sequencer compiler error messages

When you use the Format or Compile buttons in the sequence editor, Signal displays the result of the compilation or format operation in the message bar at the top of the window. The messages report either successful operation or the cause of the problem.

```
No errors, N lines compiled
```

Your sequence has been checked and is syntactically correct. This means that it will certainly run, but it is up to you to ensure that the sequence of codes produces the desired effect. If your sequence runs off the end of the defined instructions, it will stop as the compiler adds a `HALT` instruction to the end of every sequence. The number of lines compiled is a count of the number of instructions generated, which is one more than the number in the sequence text as Signal adds a `HALT` instruction to the end of the sequence.

### Formatting done

The format operation has completed and all lines of instructions are syntactically correct in themselves. There is no check that a label referenced in an instruction exists.

### Invalid label format, or same as Opcode, VAR, CONST or function

A colon was found, indicating a label, but the characters before the colon are not legal. A label starts with an alphabetic character (`A-Z`) and is followed by alphanumeric characters (`A-Z` and `0-9`) and is terminated by a colon.

Case is not significant in labels. The label can be up to 8 characters long. It must not be the same as an instruction name, a variable name, a constant name or a built-in function.

### Invalid key definition
A quote mark was found, but there was no acceptable character following the quote, or there was more than one character.

### Unknown command used
A group of characters was found in the correct position to be an instruction, but they were not recognised. Change them to a correct instruction mnemonic. This can also be caused by a missing colon at the end of a label.

### The first/second/third/fourth argument is invalid
The instruction argument is either missing, invalid or was too long to be correct. If the argument included a table reference, the offset may be too large or too small.

### Internal error, contact CED
This is caused by a serious system error. If you can reproduce this problem please contact CED for advice.

### Out of memory
Signal has run out of memory while processing the sequence. Close any non-essential windows in the Signal application and try again. If this does not allow compilation, close any other applications and retry. As you are most unlikely to exhaust your memory when compiling it may mean that something has gone badly wrong.

### Label or table size defined twice
A label has been defined on more than one line. Remember that the labels are converted to upper case, so two labels that differ in case only will cause this error. This error is also given if you use TABSZ more than once in a sequence.

### Unexpected characters at end of line
The most common cause of this error is a missing semicolon to introduce a comment.

### Couldn't find branch destination
The label used by the instruction is not defined anywhere in the sequence. Define the label, or correct the label name.

### Invalid numeric value
A numeric argument was expected, but an invalid number or one that was outside the permitted range for the argument was found.

### Probability out of range
The probability field of the BRAND instruction must be set to a number in the range 0 to less than 1.0000000000; correct the number.

### Bad digital i/o pattern
Whenever a digital i/o pattern is expected, exactly 8 characters must be present, one for each bit of the digital i/o. The characters are restricted to "0", "1" and ".", plus "c" for DIGIN and "i" for DIGOUT.

### A label, key or display string but no instruction
Labels, key definitions and display strings may only occur on lines that have an instruction. If you really want an instruction that does nothing, use NOP. Blank lines and lines consisting entirely of comments are allowed and ignored.

### Too many instructions
You have defined more than the maximum allowed number of instructions (8191, or 1023 for the original micro1401), so you must shorten your sequence. You may be able to significantly shorted your sequence by using variables and the SampleSeqVar() script instruction. You can also split a sequence into sections and load these one at a time.

### Input line too long; shorten and try again
Each line of the file must be no more than 100 characters long. Shorten the offending line and try again. This limit in previous versions of Signal was less, so it is a good idea to keep to short lines wherever you can.

*Unexpected end of file encountered*
This shouldn't happen and probably means that something has gone badly wrong.

*Variable name too long, unknown, badly formed or missing*
A variable name used to replace the `V1 to V256` built-in names is incorrectly formed.

*Bad branch code: S=Stopped, G=Going*
The branch code for the `WAVEBR` command was not one of the allowed characters listed, or the branch code was more than one character.

*Variable name already defined as Opcode, label, function or CONST*
Previous versions of Signal allowed a variable name to be the same as a label or an Opcode. This is no longer allowed. Also, the name is not allowed to the same as a built-in function, for example `VAngle` or the same as a `CONST` item. Change the variable name.

*Label already exists as Opcode, variable, function or CONST*
Previous versions of Signal allowed a label to be the same as a variable name or an instruction name. This is no longer allowed. The name may not match a built-in function or a constant. Change the label.

*The DAC value exceeds the DAC output range*
You have entered an expression for a DAC value that exceeds the DAC full-scale range. Check that the Full and Zero values in the Outputs page are correct and that you have not mistyped the value.

*More table entries than set by TABSZ*
You have used the `TABDAT` directive to create table entries. There is either no `TABSZ` directive, or you have generated more table data than you allocated with `TABSZ`.

*Target of branch is too far away*
You should only see this with an original micro1401 if you attempt to `JUMP` or branch to a target instruction that is more than 4096 instructions before or 4095 instructions after the current instruction. Although you can have sequences up to 8191 instructions, there is a limit on the size of a branch. This was a choice between slowing down all branch instructions, or limiting the range of a `JUMP` for these devices.

# Loading sequence files for sampling

The name of the output sequence file to use during sampling is part of the sampling configuration. The file name, including the path to the folder containing it, must be less than 250 characters long. You set the file name either with the Current button in the sequence editor or in the Outputs page of the Sampling Configuration dialog. When you start sampling, Signal searches for the output sequence file named in the sampling configuration and will generate an error message if it cannot be found.

Signal compiles output sequence files whenever you use them. If a file contains errors you are warned and the sequence file is not used.

# Sequencer control panel

While sampling is in progress, you can interact with sequencer execution using the sequencer control panel. The control panel displays the next sequence step and any display string associated with it. You can use display strings to prompt the user for an action or to tell the user what the sequence is doing. If there are any keyboard-controlled entry points in the sequence, these are also shown in the control panel.
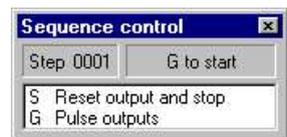
You can show and hide the control panel with the Sequencer Controls option in the Sample menu or by right clicking on any toolbar to activate the context menu. You can also dock it on any edge of the Signal application window. When undocked, the control panel displays sequence entry points as the key that activates them and a descriptive comment. When docked, the keys are displayed as buttons and the comment is hidden to save space. Move the mouse pointer over a key to see the comment as pop-up text. Click the mouse on a key and the sequencer will jump to the instruction associated with the key. The key is also stored as a keyboard marker. This is equivalent to pressing the same key in the time window or using the script language `SampleKey()` routine.

The control panel is always displayed if you start sampling using Sequencer outputs from a Signal menu command or from a dialog. If the sample command comes from the script language, the control panel visible state does not change. The control panel position is saved in the sampling configuration. However, the position is restored only if the control is currently invisible or floating, and the saved position was floating; if the control panel is docked, we assume it is positioned where you want it.

# Marker channel and digital input conflict

If you record a digital marker channel, this uses the same hardware in your 1401 as the sequencer instructions `DIBEQ`, `DIBNE`, `WAIT` and `DIGIN` that read the digital input bits. Reading digital input bits 7-0 clears the hardware flag that indicates that a marker event has occurred. These instructions can cause events to be missed on the marker channel if they read the digital input at exactly the same time as the marker event arrives.

When you sample, Signal will warn you if there is a conflict in the use of these digital input bits. You can choose to hide the warning for the remainder of your Signal session. In October 2002 we released updates to the Micro1401 mk II and Power1401 firmware that allow reads of the digital input that do not clear the hardware flag. The warning is suppressed when the new firmware is installed in your 1401. Newer 1401 designs have this fix in all firmware versions.

# Getting started

The sequencer runs instructions in order unless told to branch, starting with the first instruction. The sequence can either be restarted at the start of each sweep, or it can be set to free run throughout data acquisition. Sequencer execution can be re-routed during data acquisition by associating an instruction with a key on the keyboard. Each time the key is pressed or the associated sequencer control panel button is clicked or the script language `SampleKey()` command is used, the sequencer jumps to the associated instruction. Signal records keys pressed during sampling in the keyboard marker channel, so you can have a record of where in the sampling you switched to a new portion of the sequence.

# Simple sequence restarted each sweep

Here is a simple sequence that pulses DAC 0 high for 10 milliseconds ten times starting at 0.25 seconds into the sweep. It for use with sequencer execution restarting each sweep. You can start and stop the pulses with keys or by clicking buttons. You will find this in `Signal3\Sequence\MyFirst.pls` to save typing it in.

```
0000            MOVI   V1,10       ;Initialise loop counter
0001            DELAY  248         ;Just less than 0.25 s >Waiting
0002 LP:        DAC    0,2         ;Set output high
0003            DELAY  8           ;Wait 9 steps
0004            DAC    0,0         ;Set output low
0005            DELAY  7           ;Wait 8 steps
0006            DBNZ   V1,LP       ;Loop required times
0007            HALT               ;Stop outputs        >Done
```

Open `MyFirst.pls` and click the Check and make current sequence button at the upper right of the window (leave the mouse over each button for a second or so to see the descriptive text). Open the Sampling configuration dialog, and set a configuration with one sampled channel on ADC port 0, a sample rate of at least 10 KHz and a sweep length of a second. On the Outputs page, make sure the sequencer clock rate is set to the default of 1 millisecond, and that the Free run without restarts box is not checked. To record the output pulses you must connect the DAC 0 output to the ADC 0 input. This is easily done with a BNC cable on the 1401 front panel. You do not need to make the connection to follow this description.

Click the Run now button in the sampling configuration, and then click the Start button. The sequencer control panel is now visible. It displays Step 0000 in the top left window and a blank area to the right. If the control panel is not docked there is also an area for comment display, currently clear. If you start sampling, you will see that the control panel displays the text after the > when the relevant steps are being executed, if you are sampling the output you will see the ten pulses being generated.

As this is our first sequence we will explain it in detail. Click the Format and add step numbers button at the top of the sequencer window. All of the lines get a step number starting at zero, this is the step number displayed in the control panel. You can remove step numbers with the Format button.

Step 0 is executed at time zero in each sweep. The `MOVI V1,10` instruction on this line sets the value of variable 1 (`V1`) to 10. This variable is used as a loop counter in the sequence, so this statement sets the number of times that execution will go round the loop. The remainder of the line is a comment.

The next line, step 1, holds the instruction `DELAY 248`. This causes sequencer execution to wait at this step for the specified number of sequencer ticks into the sampling sweep, plus one more for the delay instruction itself (this could be changed, but we have kept it this way to match the behaviour of the Spike2 sequencer, for those who use both programs). The sequencer control panel displays the text to the right of the > for the current step, so while this step is executing, the text "Waiting" is displayed. Note that we wait for 249 ticks (0.249 seconds). This means that the next step will execute at precisely 250 steps into the sweep, or 0.25 seconds. Note also that we could have written `DELAY ms(249)-1` to achieve the same effect and that this would have the advantage of being immune to changes in the sequencer tick interval.

Step 2 is `DAC 0,2`, which sets the output of DAC zero to two volts. This step has a label at the start of the instruction text, `LP:`, so we can branch back here from elsewhere in the sequence.

Step 3 is `DELAY 8`, which causes sequencer execution to wait at this step for nine sequencer clock ticks, thus setting the width of the pulse. The width will include the time taken by the DAC instruction as well, giving ten ticks (0.01 seconds) overall. Note that, as for step 1, we could have written `DELAY ms(9)-1` to achieve the same effect.

Steps 4 and 5 are the same as steps 2 and 3; they set the DAC output low again and wait for 8 sequencer ticks. We only wait for 8 ticks because step 6 adds an extra step into this part of the sequence, so we get 10 ticks overall for the low phase too.

Step 6 is `DBNZ V1,LP`, which causes the sequencer to decrement the value of variable number 1 and, if it has not reached zero, branch to the label specified so that the next instruction executed will be step 2. As we have put the value 10 into variable 1 at the start of the sequence, this will cause the loop that creates a pulse to execute ten times, so we get ten pulses.

Finally step 7 is `HALT`, which causes sequencer execution to stop until the start of the next sweep, where it will be automatically restarted at step 0. This step displays the text "Done" in the sequencer control panel while it is executing.

# Free running sequence

The simple way to use the sequencer is with restarts, so that the sequence restarts at the first step at the start of each sweep. However, this mode of operation may prove insufficiently flexible for some situations. If you check the Free run without restarts box in the Outputs page, you get a different style of sequencer operation. You will find a sequence that operates in this manner in `Signal3\Sequence\MySecond.pls`; this also demonstrates some other aspects of sequencer use:

```
0000 FL:     'F DAC   0,2         ;Start fast pulse    >S for slow
0001           DELAY ms(9)-1       ;Wait 9 ms           >S for slow
0002           DAC   0,0           ;Set output high     >S for slow
0003           DELAY ms(8)-1       ;Wait 8 ms           >S for slow
0004           JUMP  FL            ;Continue            >S for slow
0005 SL:     'S DAC   0,2         ;Set slow pulse      >F for fast
0006           DELAY ms(99)-1      ;Wait 99 ms          >F for fast
0007           DAC   0,0           ;Set output low      >F for fast
0008           DELAY ms(98)-1      ;Wait 98 ms          >F for fast
0009           JUMP  SL            ;Continue            >F for fast
```

Most of this sequence is similar to `MyFirst.pls`. The differences of interest are:

Steps 0 and 5 have a keyboard character defined as well as a label for looping. The sequencer control panel will show buttons corresponding to each character defined; pressing the button or the appropriate keyboard character will cause sequencer execution to jump to the relevant instruction.

A `JUMP` instruction is used instead of `DBNZ` so that the pulse output loops continue forever.

The remainder of this information is organised as a reference manual for the sequencer instructions. You can find more information about the output sequencer in the *Signal Training Course* manual.

# Instructions

## Instruction format

Blank text lines and lines with a semicolon as the first non-blank character, are ignored. Instructions are not case sensitive. Each instruction has the format:

```
num lab:      'key   code   arg1,arg2,…  ;Comment     >display
```

| | |
|---|---|
| `num` | An optional step number in the range 0 to 1022, for information only. |
| `lab:` | An optional label, up to 8 characters long followed by a colon. The first character must be alphabetic (A-Z). Labels are not case sensitive. Labels may not be the same as instruction codes, variable names, constants or expression function names. |
| `'key` | In this optional field, `key` is one alphanumeric (a-z, A-Z, 0-9) character. When this character is recorded as a keyboard marker during data capture, the sequencer jumps to this instruction. Each `key` can occur once. Upper and lower case are distinct. The `key` appears in the sequencer control panel. |
| `code` | This field defines the instruction to be executed. It is not case sensitive. |
| `arg1,…` | Instructions need up to 4 arguments and are separated by commas or spaces. These are described with the instructions. If an argument can be represented in different ways, they are separated by vertical bars (read as "or"), for example: `expr\|Vn\|[Vn+off]`. In this case, the argument can be an expression, a variable or a table reference. |
| `comment` | The text after the semicolon is to remind you of the reason for the instruction. If a `key` is set, this comment also appears in the sequencer control panel. |
| `>display` | When a sequence runs, text following a ">" in a comment is displayed in the sequencer control panel to indicate the current instruction. Signal is notified every few milliseconds of the current instruction and the control panel is updated whenever there is free time, so the display is only a sampling of what is going on. The special combination >" means use the same display text as for the previous instruction. This can save you some typing when a group of instructions in a loop need the same display. If there are several instructions in a row that use this, Signal searches backwards until it finds an instruction with a display string that is not >". The special combination >= means make no change to the display string. This might be used in a code section that is called as a subroutine from several different places. |

## Expressions

Many instructions allow the use of an expression in place of a constant value, indicated by `expr`. An expression is formed from constants, numbers, labels, round brackets `(` and `)`, the operators `+`, `-`, `*` and `/`, and sequencer expression functions. Labels have the value of the instruction number in the sequence; the first instruction is numbered 0.

The operators `*` and `/` (multiply and divide) have higher priority than `+` and `-` (add and subtract). This means that `1+2*3` is interpreted as `1+(2*3)` and not as `(1+2)*3`. Apart from this, evaluation is from left to right unless modified by brackets.

The sequence compiler evaluates expressions as real numbers, so `3/2` has the value `1.5`. If `expr` is used as an integer, for example `DELAY expr`, it is rounded to the nearest integer. Values in the range 3.5 to 4.49999… are treated as 4. Before version 4.06 the result was truncated, so 3.0 to 3.9999… was treated as 3.

### Sequencer expression functions
These functions can be used as part of expressions to give you access to sequencer step timing and to convert between user units and DAC and ADC values and to calculate table indices.

| | |
|---|---|
| `s(expr)`<br>`ms(expr)`<br>`us(expr)` | The number of sequencer steps in `expr` seconds, milliseconds and microseconds. For example, with a step size of 200 milliseconds, `s(1.1)` returns 5.5. This is often used with the `DELAY` instruction. Each instruction uses 1 step, so use `DELAY s(1)-1` for a delay of 1 second. |

| | |
|---|---|
| Hz(expr) | The angle change in degrees per step for a cosine output of expr Hz. For a 2 Hz cosine on DAC n, use RATE n,Hz(2). To slow the current rate down by 0.1 degrees per step use RINC n,Hz(-0.1). Use in RATE, RATEW, RINC and RINCW instructions. |
| VHz(expr) | The same as Hz(), but the result is scaled into the 32-bit integer units used when a variable sets the rate. MOVI V1,VHz(2) followed by RATE n,V1 will set a 2 Hz rate. |
| VAngle(expr) | Converts an angle in degrees into the internal angle format. The 32-bit integer range is 360 degrees. The result is expr * 11930464.71. For use with ANGLE and PHASE. |
| VDAC16(expr) | Converts expr user DAC units so that the full DAC range spans the full range of a 16-bit integer. |
| VDAC32(expr) | Converts expr user DAC units into a 32-bit integer value such that the full DAC range spans the 32-bit integer range. Use this to load variables for use with the DAC and ADDAC instructions. |
| ASz(expr) | Converts expr user DAC units into a value suitable for use with the SZ and SZINC commands. That is, it converts an amplitude or amplitude change in user DAC units to a value in the range 0 to 1. It does this by dividing the value you supply by the (DAC scale factor * 5). |
| VSz(expr) | Converts expr user DAC units into a variable value suitable for use with the SZ and SZINC commands. That is, it converts an amplitude or amplitude change in user DAC units to a value in the range 0 to 32768. It does this by multiplying the value you supply by 32768/ (DAC scale factor * 5). |
| TabPos() | The number of table data items defined at this point in the sequence by TABDAT. This is also the index of the next table data item to use. You can use this to define constants that reference table indices. |
| DRange() | The DAC output range (usually 5.0 for a ±5 Volt system or 10.0 for a ±10 Volt system). When the sequence is built for use during sampling, this value is taken from the DAC output range of the attached 1401, otherwise, this is taken from the Voltage range set in the Sample page of the Preferences dialog. |

Used with care, the built-in functions allow you to write sequences that operate in the same way regardless of the sequencer step time or DAC scaling values.

# Variables

You can use the 256 variables, V1 to V256 in place of fixed values in many instructions. In the sequencer command descriptions, Vn indicates the use of a variable. Where a variable is an alternative to a fixed value expression we use expr|Vn. Variables hold 32-bit integer numbers that you can set and read with the SampleSeqVar() script command.

Some variables have specific uses: Variables V57 through V64 hold the last value written by the sequencer to DACs 0 through 7 and V56 holds the last bit pattern read from the digital inputs with the DIBxx or DIGIN instructions.

There are sequencer instructions that can be used to perform arithmetic and logic operations on variables.

### VAR directive
You can assign each variable a name and an initial value with the VAR directive. Names must be assigned before they are used, usually at the start of the sequence. The syntax is:

```
        VAR      Vn,name=expr            ;comment
```

VAR does not generate any instructions. It makes the symbol name equivalent to variable Vn and sets the initial value when the sequence is loaded. Anywhere in the remainder of the sequence where Vn is acceptable, name can be used. name can be up to 8 characters, must start with an alphabetic character and can contain alphabetic characters and the digits 0 to 9. Variable names are not case sensitive. A variable name must not be the same as an instruction code or a label.

There is no need to specify a name or an initial value. If no initial value is set, a variable is initialised to 0 even if not included in a VAR statement. Signal automatically assigns V56 the name VDigIn and variables V57 through V64 the names VDAC0 through VDAC7. The following are all acceptable examples:

```
        VAR     V1,Wait1=ms(100)    ;Set name and initial value
        VAR     V2,UseMe            ;Set name only, so value is 0
        VAR     V3=200              ;No name, initialise to a value
        VAR     V4                  ;No name, initialised to 0
```

When a variable is used in place of a voltage value in a DAC output instruction, the full 32-bit range of the sequencer variable value corresponds to the full range of voltages that can be generated by the DAC, so -2147483648 corresponds to the lowest possible output voltage (-5 or -10 volts) while 2147483647 corresponds to the highest possible output voltage (almost +5 or +10 volts).

When a variable is used in place of a bit pattern in a digital input or output instruction, bits 15 to 8 and bits 7 to 0 have different uses. In the expressions that describe these operations we write `Vn(7-0)` and `Vn(15-8)` to describe which bits are used. `BAND` means bitwise binary `AND` (if both bits are 1, the output is 1, otherwise 0), `BXOR` means bitwise exclusive `OR` (if both bits are different the output is 1, otherwise 0).

When used in one of the cosine output angle instructions, the 32-bit variable range from -2147483648 to 2147483647 represents -180 up to +180 degrees. The `VarValue` script in the `Scripts` folder calculates variable values for the digital and the cosine instructions. The `SeqLib` script installed with Signal and downloadable from the CED website contains a complete set of functions to convert from user values to correctly scaled sequencer variable values.

### Script access to variables

Scripts can set and read sequencer variable values by using the `SampleSeqVar()` script command. See *The Signal script language* manual for details. You can set initial values from the script as long as you set the values after you create the new data file, but before you start sampling. Values set in this way take precedence over values set by the `VAR` directive.

# Constants

It is often useful to define a numeric value as a named constant. For example, when referencing a table value `[V1+Slope]` is easier to understand than `[V1+23]`. It also helps to maintain code; if you need to change a numeric value that is used often make it a named constant and just change the constant once. The = directive does not generate any instructions, it just makes a name equivalent to a number, and the name can be used anywhere a numeric expression is valid. The syntax is:

```
name = expr              ;comment
```

The `name` must start with an alphabetic character and can contain alphabetic characters and the digits 0 to 9. The name can be up to 8 characters long. Numeric constant names are not case sensitive and must not clash with label, instruction, variable or built-in function names. Examples of use:

```
Wait1  = ms(100)
Wait2  = Wait1*1.3
       DELAY  Wait1-1  ;Use constant in an instruction
```

The expression values are calculated and stored as floating point numbers. If they are used in a context that requires an integer value, the fractional part of the number is ignored.

The = directive was added at version 4.06.

# Table of values

The Signal sequencer supports a table of 32-bit values that can be up to 1000000 items in length.

### Declaring the table

You declare that a table exists with the `TABSZ` directive, which normally occurs at the start of your sequence:

```
        TABSZ  expr
```

Where `expr` is an expression that sets the number of items in the table. The table size must evaluate to a number in the range 1 to 1000000. Each table item is a 32-bit integer and uses 4 bytes of 1401 memory, if you try to use a table size that leaves too little 1401 memory available for other necessary purposes (sequencer instructions, arbitrary waveforms and sampled data) an error will occur when you try to start sampling. The first table item has an index number of 0, the second item is index 1, and so on.

### Setting table data

From the script language you can move data between an integer array and the table with the `SampleSeqTable()` function. You can also preset table data from the sequence with the `TABDAT` directive, which must come after the `TABSZ` directive:

```
        TABDAT expr
        TABDAT expr,expr,expr...
```

Where `expr` is an expression that evaluates to a 32-bit integer. Each `TABDAT` directive adds data to the table, starting at the beginning. The sequencer compiler will flag an error if you define more data that will fit in the table. Table data declared in this way is stored separately from the sequence and is transferred to the 1401 when you create a new data file to sample. If you do not set the table data with the `TABDAT` directive or from a script, the values in the table are undefined.. The `TabPos()` expression has the value of the number of data items that have been defined at the point where it is used. For example:

```
        TABDAT TabPos() ;a table item that holds its own index.
```

### Accessing table data

Although you can move data between one of the variables and the table with the `TABLD` and `TABST` instructions, many instructions access the table directly. It takes more time to use a table than to use a variable.

All references to the table use the contents of one of the variables as an index into the table plus an optional offset as: `[Vn]` or `[Vn+off]` or `[Vn-off]`. The offset `off` is an expression that evaluates to a number in the range –1000000 to 1000000. For example, if `V1` holds 10, `[V1]` refers to the contents of index 10, `[V1-10]` refers to index 0 and `[V1+10]` refers to index 20. Out of range table indices read 0 and are non-destructive.

The `TABINC` instruction makes it easy to increment a variable used as a table index and branch until the increment generates an index outside the table. The following example generates five DAC outputs at 5 different intervals:

```
            SET    1,1,0                 ; 1ms per step, normal scales
            TABSZ  10                    ; table of 10 items
oMs         =      TabPos()              ; offset to milliseconds
            TABDAT ms(1000)-3            ; 1000 ms in sequencer steps-3
oVolt       =      TabPos()              ; offset to Volts
            TABDAT VDac32(1)             ; 1 Volt
oNext       =      TabPos()              ; offset to next set of entries
            TABDAT ms(100)-3,VDac32(2)   ;  100 ms, 2 Volts
            TABDAT ms(50)-3,VDac32(3)    ;   50 ms  3 Volts
            TABDAT ms(500)-3,VDac32(-1)  ;  500 ms -1 Volt
            TABDAT ms(200)-3,VDac32(0)   ;  200 ms  0 Volts

            MOVI   V1,0                  ; use V1 as table index, set 0
TLOOP:      DELAY  [V1+oMs]              ; programmed delay
            DAC    0,[V1+oVolt]          ; set DAC 0 to the value
            TABINC V1,oNext,TLOOP        ; add 2 to V1, branch if in table
```

In this case, we could just as easily have set `oMs` to 0, `oVolt` to 1 and `oNext` to 2, or used the values 0, 1 and 2 in the code. However, by using constants, we make it easy to extend the number of items per step. For example, if we decided to insert a new value in the table, making three items per step, we could do so without having to work through our code to check if each occurrence of 0, 1 or 2 was correct or needed changing.

### Long data sequences

If you have a very long data sequence, you should consider using the table as a buffer. The basic idea is to divide the table into two halves and use a script to transfer new data into the half of the table that the sequence is not using. To find out where the sequence has reached, look at the value of the variable used as an index with `SampleSeqVar()`. Set a large enough table size so that the time taken to use half the table is several seconds.

# Including files

There are times when you will want to reuse definitions or sequence code sections in multiple projects. You can do this by pasting the text into your sequence, but it can be more convenient to use the `#include` command to include files into a sequence. A file that is included can also include further files. We call these nested include files. Only the first `#include` of a file has any effect. Subsequent `#include` commands that refer to the same file are ignored. This prevents problems with output sequence files that include each other and stops multiple definitions

when two files include a common file. A `#include` command must be the first non-white space item on a line. There are two forms of the command:

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

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

1.  Search the folder where the file with the `#include` command lives. If this fails...
2.  Search the folder of the file that included that file until we reach the top of the list of nested include files. If this fails...
3.  Search any `\include` folder inside the `Signal7` folder inside your `My Documents` folder (this location was added at version 5.08). If this fails...
4.  Search the `\include` folder inside the `Signal7Shared` folder inside the `Public Documents` folder (this location was added at version 5.08). If this fails...
5.  Search any `\include` folder in the folder in which Signal is installed. If this fails...
6.  Search the current folder.

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

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

The `#include` command for output sequences was added to Signal at version 4.06 and is not recognised by any version before this. A typical file using `#include` might start with:

```
#include <sysinc.pls>          ; my system specific includes
#include "include/proginc.pls" ; search script relative folder
  .....                        ; start of my code...
```

### Opening included files

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

### Errors in included files

If an error is detected during the compilation of an included file, an error message is displayed at the top of the original window indicating which included file has a problem, and the included file is opened (if it can be found) and the offending line is highlighted.

# Sequencer instruction reference

Each instruction below is followed by an example. The examples show the preferred instruction format, however the system is flexible. For example, a comma should separate arguments, but a space is also accepted. The patterns used for digital ports should be enclosed by square brackets, however you may omit the brackets if you wish.

Many of these instructions allow you to use a variable or a table entry in place of an argument. In this case, the alternatives are separated by a vertical bar, for example:

```
        DELAY    expr|Vn|[Vn+off],OptLB
```

This means that the first argument can be an expression, a variable or a table entry. There is no explicit documentation for the use of the table, except in `TABLD` and `TABST`. Where table use is allowed it is written as `[Vn+off]`. If you use a table value in an instruction, the effect is exactly the same as using a variable with the same value as the table entry.

OptLab in instructions is an optional label that sets the next instruction to run. If it is omitted, the next sequential instruction runs.

# Digital I/O

These instructions give you control over the digital output bits and allow you to read and test the state of digital input bits 7-0.

## DIGOUT

The DIGOUT instruction changes the state of digital output bits 15-8 (see under *Sampling data* for the connections). The output changes occur at the next tick of the output sequencer clock, so you need to use this instruction one tick early!

```
          DIGOUT   [pattern]|Vn|[Vn±off],OptLab
```

pattern    This determines the new output state. You can set, reset or invert each output bit, or leave a bit in the previous state. The pattern is 8 characters long, one for each bit, with bit 15 at the left and bit 8 at the right. The characters can be "0", "1", "i" or "." standing for *clear*, *set*, *invert* or *leave alone*. You may omit the square brackets, however the Format command will insert them.

```
          DIGOUT   [....001i]  ;clear bits 3 and 2, set 1, invert 0
          DIGOUT   [.......i]  ;invert 0 again to produce a pulse
          DIGOUT   V10         ;use variable V10 to set the pattern
```

Vn    With a variable the new output is: (old output BAND Vn(7-0)) BXOR Vn(15-8). The variable equivalent of [....001i] is 241+256*3, and of [.......i] is 255+256*1. If you use a table value, set the same value in the table that you would use for a variable. You can use the VarValue script in the Scripts folder to calculate variable or table values.

OptLB    If this optional label is present it sets the next instruction to run.

This example produces ten 1 millisecond pulses 100 milliseconds apart.

```
          MOVI     V1,10        ;V1 holds the number of pulses
LOOP:     DIGOUT   [.......1]   ;bit 0 high           >Pulsing
          DIGOUT   [.......0]   ;bit 0 low            >Pulsing
          DELAY    ms(100)-4    ;4 inst in the loop >Pulsing
          DBNZ     V1,LOOP      ;count down           >Pulsing
          HALT                  ;finished             >Done
```

**See also:**
Output connections, Variables, DIGLOW

## DIGLOW

The DIGLOW instruction changes the state of digital output bits 7-0 of Power1401s and Micro1401s (see under *Sampling data* for the connections). Unlike DIGOUT, the output changes occur immediately, they do not wait for the next sequencer clock tick. You can take advantage of this to change all 16 digital outputs almost simultaneously (within a few microseconds) by using DIGOUT followed by DIGLOW.

```
          DIGLOW   [pattern]|Vn|[Vn+off],OptLB
```

pattern    This determines the new output state. The pattern is 8 characters long, one for each bit, with bit 7 at the left and bit 0 at the right. The characters can be "0", "1", "i" or "." standing for *clear*, *set*, *invert* or *leave alone*. You may omit the square brackets, however the Format command will insert them.

```
          DIGLOW   [....001i]  ;clear bits 3 and 2, set 1, invert 0
          DIGLOW   [.......i]  ;invert 0 again to produce a pulse
          DIGLOW   V10         ;use variable V10 to set the pattern
```

Vn    With a variable the new output is: (old output BAND Vn(7-0)) BXOR Vn(15-8). The variable equivalent of [....001i] is 241+256*3, and of [.......i] is 255+256*1. If you use a table

value, set the same value in the table that you would use for a variable. You can use the `VarValue` script in the `Scripts` folder to calculate variable or table values.

OptLB    If this optional label is present it sets the next instruction to run.

This example produces ten 1 millisecond pulses 100 milliseconds apart.

**See also:**
Output connections, Variables, DIGOUT

# DIBEQ DIBNE

These instructions test digital input bits 7-0 against a pattern (see under *Sampling data* for connections). These are the same inputs that will be used for digital markers in the future. DIBEQ branches on a match. DIBNE branches on a non-match. Both instructions copy digital input bits 7-0 to V56 (VDigIn), for use by DISBEQ and DISBNE.

```
        DIBNE    [pattern]|Vn|[Vn+off],LB
        DIBEQ    [pattern]|Vn|[Vn+off],LB
```

pattern   This is 8 characters, one for each input bit. The characters can be "0", "1" and "." meaning match 0 (TTL low), match 1 (TTL high) or match anything. The bit order in the pattern is [76543210]. You may omit the square brackets, however the Format command inserts them.

Vn        With a variable the result is: (input BAND Vn(7-0)) BXOR Vn(15-8). A result of 0 is a match, not zero is not a match.

LB        The destination of the branch if the input was a match (DIBEQ) or not a match (DIBNE). This label must exist in the sequence.

This example waits for a pulse sequence in which the falling edges of two consecutive pulses are less than $2*V1+2$ sequencer clock ticks apart. It waits for a falling edge, waits for a rising edge with a timeout and then waits for the next falling edge with a timeout. If timed out, we start again. If the input signal has high states less than three ticks wide, or low states less than 2 ticks wide, this example may miss them.

```
WHI:    DIBNE    [.......1],WHI  ;wait until high        >Wait high
SETTO:  MOVI     V1,24           ;set 50 step timeout    >Wait low
WLO:    DIBNE    [.......0],WLO  ;wait for falling       >Wait low
TOHI:   DIBEQ    [.......1],TOLO ;wait for high          >Wait high
        DBNZ     V1,TOHI         ;loop if not timed out  >Wait high
        JUMP     WHI             ;timed out, restart     >Restart
TOLO:   DIBEQ    [.......0],GOTIT;jump if found events   >Wait low
        DBNZ     V1,TOLO         ;loop if not timed out  >Wait low
        JUMP     SETTO           ;timed out, restart     >Restart
GOTIT:  ...                      ;here for 2 close pulses
```

**See also:**
Connections, Variables, DISBEQ,DISBNE

# DISBEQ DISBNE

These instructions test digital input bits 7-0 read by the last DIBEQ, DIBNE or WAIT against a pattern. DISBEQ branches on a match. DISBNE branches if it does not match.

```
        DISBNE   [pattern]|Vn|[Vn+off],LB
        DISBEQ   [pattern]|Vn|[Vn+off],LB
```

pattern   This is 8 characters, one for each input bit. The characters can be "0", "1" and "." meaning match 0 (TTL low), match 1 (TTL high) or match anything. The bit order in the pattern is [76543210]. You may omit the square brackets, however the Format command inserts them.

Vn        With a variable the result is: (input BAND Vn(7-0)) BXOR Vn(15-8). A result of 0 is a match, not zero is not a match.

LB        The destination of the branch if the input was a match (DISBEQ) or not a match (DISBNE). This label must exist in the sequence.

This example shows a typical use of this instruction. We want to run a different part of the sequence for three trial types signalled by external equipment that writes the trial type to digital input bits 1 and 0; 00 means no trial, 01, 10 and 11 select trial types 1, 2 and 3.

```
TRWAIT:'W DIBEQ   [......00],TRWAIT ;Wait for trial >Wait...
          DISBEQ  [......01],TRIAL1
          DISBEQ  [......10],TRIAL2
          DISBEQ  [......11],TRIAL3
```

**See also:**
Variables, DIBEQ,DIBNE

# DAC outputs

The output sequencer supports up to 8 DAC (Digital to Analogue Converter) outputs. The Power1401s have four DACs (expandable to 8) and Micro1401s have two. The last value written to DACs 0-7 is stored in variables V57-V64 (which you can also refer to as VDAC0-VDAC7). The values are stored as 32-bit numbers with the full 32-bit range corresponding to the full range of the DAC. This high resolution allows us to ramp the DACs smoothly.

Values written to the DACs are expressed in units of your choice. The DAC scaling set in the sampling configuration Outputs page determines the conversion between the numbers you supply and the DAC outputs. All DACs are scaled identically. The standard settings for a system with ±5 volts DACs is to set the DAC outputs in volts.

If you write to a DAC that does not exist, the variable associated with the DAC is set as if the DAC were present, but has no other effect.

The Power1401 DAC 2 and 3 outputs are on pins 36 and 37 of the rear panel 37-way Cannon D type Analogue Expansion connector. Suitable grounds are on the adjacent pins 18 and 19. With a Power1401 top-box with additional front panel DACs, the rear panel DAC outputs are mapped to the two highest numbered DACs. For example, with a 2709 Spike2 top box, DACs 2 and 3 are available as BNC connections on the front panel, and the rear panel DACs become DAC 4 on pin 36 and DAC 5 on pin 37.

# DAC ADDAC

The DAC instruction writes a value to any of the 8 possible DAC outputs. ADDAC adds a value to the DAC output. The output value changes immediately unless the DAC is in use by the arbitrary waveform output, in which case the result is undefined.

```
          DAC     n,expr|Vn|[Vn+off],OptLB
          ADDAC   n,expr|Vn|[Vn+off],OptLb
```

n      The DAC number, in the range 0-7. Variable 57+n is set to the new DAC value such that the full DAC range spans the full range of the 32-bit variable.

expr      The value to write to the DAC or the change in the DAC value. The units of this value are as set in the outputs page of the sampling configuration dialog. It is an error to give a value that exceeds the DAC output range.

Vn      When a variable is used, the full range of the 32-bit variable corresponds to the full range of the DAC, so -2147483648 corresponds to the lowest possible output voltage (-5 or -10 volts) while 2147483647 corresponds to the highest possible output voltage (almost +5 or +10 volts). You can use the VDAC32() function to load a variable using user-defined DAC units, the SeqLib script library contains a function that mimics the behaviour of VDAC32 to calculate variable values within a script.

OptLB      If this optional label is present it sets the next instruction to run.

This example ramps DAC 2 from 0 to 4.99 volts in 1 second in steps of 0.01 volts using values and then using variables. You can also use the RAMP instruction to ramp a DAC.

```
         'R DAC    2,0              ;Ramp 0 to 5        >Ramping
            MOVI   V1,499           ;499 steps          >Ramping
RAMP1:      ADDAC  2,0.01           ;0.01V increment  >Ramping
            DBNZ   V1,RAMP1         ;count increments >Ramping
            HALT                    ;task finished      >Done
         'V MOVI   V3,VDAC32(0)     ;Use variables      >Ramping
            MOVI   V2,VDAC32(0.01)  ;increment in V2   >Ramping
            MOVI   V1,499           ;499 steps          >Ramping
            DAC    2,V3             ;set initial value>Ramping
RAMP2:      ADDAC  2,V2             ;add increment      >Ramping
            DBNZ   V1,RAMP2         ;count increments >Ramping
            HALT                    ;task finished    >Done
```

It is a property of signed integers that adding 1 to the maximum positive number yields the minimum negative number. If you use ADDAC repeatedly with the same value, eventually you will run off the end of the DAC range and come back in at the other end.

DAC units run from -32768 to +32767. In a ±5 volt system with 16-bit DACs, this is -5.0000 to +4.99985 volts. The DAC unit value for +5 volts is +32768, but this number does not exist in 16-bit signed integers and wraps around to -32878. Users often want to set the DAC to full scale, so for the DAC command used with expr (not with Vn), we change requests to set +32768 units to set +32767 units. When a sequencer variable is written out to a DAC the upper half of the variable (the top 16 bits) provide the data that is written.

Unlike the digital outputs, the DAC output changes when the instruction runs, not at the next sequencer clock tick; the changes may have a time jitter of a few microseconds.

**See also:**
General DAC information, Expressions, Variables

# RAMP

This command starts a DAC ramping with updates every sequencer step. If the DAC was generating a cosine, the cosine output stops. The DAC ramps from the current value until it reaches a target value, when the DAC cycle flag sets. You can use WAITC to test for the end of the ramp. The RATE instruction stops a ramp before it reaches the target value.

```
         RAMP      n,target|Vn,slope|Vs|[Vs+off]
```

n         DAC number in the range 0-7 (available DACs depend on the 1401 type).

target    This is the DAC value at which to end the ramp. The units of the DAC values are those set in the outputs page of the sampling control dialog.

Vn        When a variable is used for the target, the full range of the 32-bit variable corresponds to the full range of the DAC. You can use the VDAC32() function to load a variable using user-defined DAC units.

slope     This expression sets the DAC increment per sequencer step. The sign of the value you set here is ignored as the sequencer works out if it must ramp upwards or downwards to achieve the desired target value. If your DAC is calibrated in volts, for a slope of 1 volt per second, use 1.0/s(1.0).

Vs        You can also set the slope from a variable or by reading it from the table. In this case, the full range of the 32-bit value represents the full range of the DAC. The DAC changes by the absolute value of this 32-bit value on each step. For a slope of 1 user unit per second, use VDAC32(1.0)/s(1.0).

This example ramps DAC 1 from its current level to 1 volt in 3 seconds, waits 1 second, then ramps it to 0 volts in 5 seconds. See the OFFSET command for another example.

```
         RAMP    1,1.0,1.0/S(3) ;start with zero size
         ...                    ;other instructions during ramp
WT1:     WAITC   1,WT1          ;wait for ramp to end >Ramp to 1
         DELAY   S(1)-1         ;wait for a second    >Wait 1 sec
         RAMP    1,0,1.0/S(5)   ;ramp down
WT2:     WAITC   1,WT2          ;wait for ramp         >Ramp to 0
```

**See also:**
General DAC information, Expressions, Variables, WAITC, RATE, OFFSET

# Cosine output control instructions

The sequencer can output cosine waveforms of variable amplitude and frequency through Micro1401 DACs 0 and 1 and Power1401 DACs 0 to 7. If you attempt to set cosine output for an unsupported DAC, the instruction is treated as a NOP. When enabled, the cosine value is computed and output every step. The time penalty per step per DAC is around 4 us for the micro1401 and about 1 us for the Power1401, Power1401 mk II, Power1401-3, Micro1401 mk II and Micro1401-3. The output is:

output in volts = 5 A $Cos(\theta+\phi)$ + offset

where  A       is an amplitude scaling factor in the range 0 to 1
  $\theta$       an angle in the range 0° to 360° that changes each step (set by ANGLE)
  $\phi$       is a fixed phase angle in the range -360° to 360° (set by PHASE)
  offset   A voltage offset defined by the OFFSET command

$\theta$ changes every step by $d\theta$. A cycle of the cosine takes $360/d\theta$ steps. You can change the angle increment immediately, or you can delay the change until the next time Theta passes through 0°. You can set $d\theta$ in the range 0° up to 360° to an accuracy of about 0.0000001°. With the sequencer running at 1 kHz, you can output frequencies up to 500 Hz with a frequency resolution of around 0.00012 Hz. Ideally the output would be passed through a low pass filter with a corner frequency at one half of the sequencer step rate to smooth out the steps in the cosine wave.

By adjusting $\phi$ you control the output cosine phase where $\theta$ passes through zero. Unless you set the value (PHASE), it is zero and the zero crossing occurs at the peak of the sinusoid. To have the output rising through 0, set the phase to –90.

Each time $\theta$ passes through zero a *new cycle* flag sets. The RAMP, RATEW, RINCW, WAITC and CLRC instructions clear the flag.

**Output as a function of θ+ϕ**
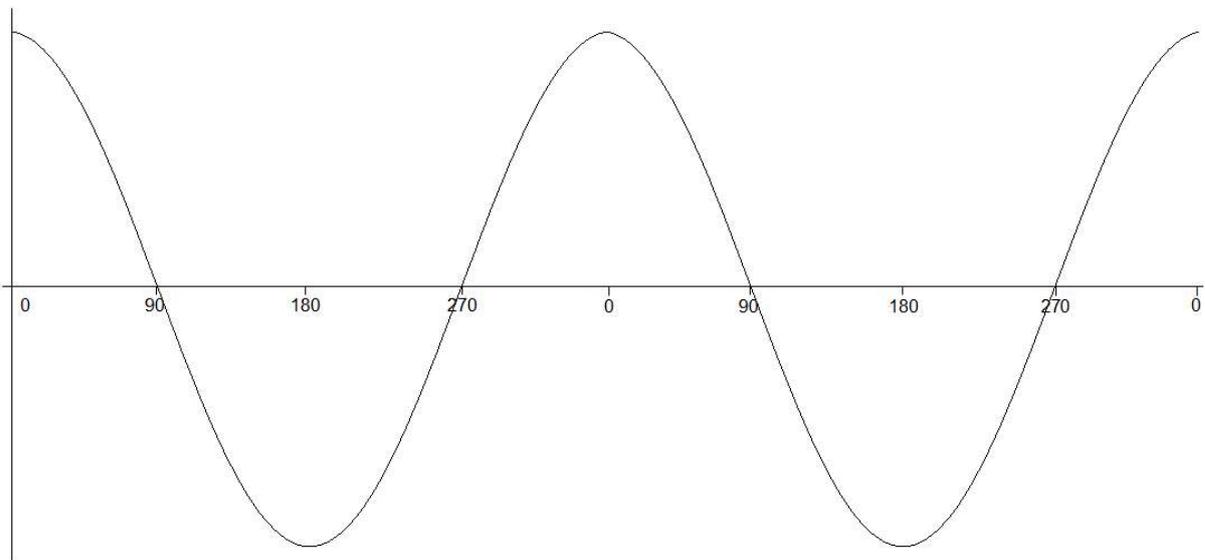


## SZ

This instruction sets the waveform amplitude. If a wave is playing, the amplitude changes at the next sequencer step. The amplitude is set to 1.0 when sampling starts.

```
        SZ        n,expr|Vn|[Vn+off],OptLB      ;DAC n
```

n       DAC number in the range 0-7 (available DACs depend on the 1401 type).

expr    The cosine amplitude in the range 0 to 1. A cosine with amplitude 1.0 uses the full DAC range.

Vn      Variable values 0 to 32768 correspond to amplitudes of 0.0 to 1.0; values outside the range 0 to 32768 cause undefined results.

OptLB    If this optional label is present it sets the next instruction to run.

**See also:**
Cosine output, SZINC amplitude change, OFFSET set cosine offset, General DAC information, Expressions, Variables

## SZINC

This instruction changes the waveform amplitude. The change is added to the current amplitude. If the result exceeds 1.0, it is set to 1.0. If it is less than 0, the result is 0.

```
        SZINC  n,expr|Vn|[Vn+off],OptLB      ;DAC n
```

n        DAC number in the range 0-7 (available DACs depend on the 1401 type).

expr     The change in the waveform scale in the range -1 to 1.

Vn       A variable value of 32768 is a scale change of 1.0, -16384 is -0.5 and so on.

OptLB    If this optional label is present it sets the next instruction to run.

You can gradually increase or decrease the wave amplitude. For example, the following increases the amplitude from zero to full scale (we assume that the waveform is playing):

```
        SZ     0,0.0      ;start with zero size
        MOVI   V1,100     ;proceed in 1% increments
loop:   SZINC  0,0.01     ;a 1% increase
        DELAY  ms(100)-2  ;show some of the waveform at this size
        DBNZ   V1,loop    ;loop 100 times
```

**See also:**
Cosine output, SZ cosine amplitude, OFFSET set cosine offset, General DAC information, Expressions, Variables

## RATE

This sets the angle increment in degrees per step, which sets the cosine frequency. If the nominated DAC was ramping, this cancels the ramp. You can stop the cosine output with a rate of 0. Any non-zero value starts the cosine output.

```
        RATE    n,expr|Vn|[Vn+off],OptLB      ;DAC n
```

n        DAC number in the range 0-7 (available DACs depend on the 1401 type).

expr     The angle increment per step in the range 0.000 up to 180 degrees. The Hz() function calculates the increment required for a frequency.

Vn       For a variable, the value 11930465 is an increment of 1 degree. The VHz() function can be used to set a variable value equivalent to an angle in degrees.

OptLB    If this optional label is present it sets the next instruction to run.

This example starts cosine output at 10 Hz, runs for 10 seconds, and then stops it. This is then repeated using a variable to produce the same effect:

```
     'C RATE   0,Hz(10)   ;start output at 10 Hz
        DELAY  S(10)-1    ;delay for 10 seconds >Sine wave
X:   'S RATE   0,0        ;stop output
        HALT                                   >Stopped
     'V MOVI   V1,VHz(10) ;set V1 equivalent of 10 Hz
        RATE   0,V1       ;start at 10 Hz
        DELAY  S(10)-1,X  ;delay then goto exit >Sine wave
```

**See also:**
Cosine output, RINC(W) change frequency, RATEW set frequency change at phase 0, ANGLE change phase, WAITC wait for phase 0, PHASE define phase zero, General DAC information, Expressions, Variables

# RATEW

This instruction performs the same function as RATE, except that the change is postponed until the next time Theta passes through 0 degrees. RATEW cannot start output; a sinusoid must already be running to pass phase 0. It can stop output, but does not remove the overhead for using cosine output.  This instruction clears the new cycle flag (see WAITC).

```
        RATEW     n,expr|Vn|[Vn+off]         ;DAC n
```

n       DAC number in the range 0-7 (available DACs depend on the 1401 type).

expr    The angle increment in the range 0.000 to 180 degrees. The Hz() built-in function calculates the increment required for a frequency.

Vn      For a variable, the value 11930465 is an increment of 1 degree. The VHz() function can be used to set a variable value equivalent to an angle in degrees.

OptLB   If this optional label is present it sets the next instruction to run.

This example starts cosine output at 10 Hz, runs for 1 cycle, changes to 11 Hz for one cycle, then stops:

```
        ANGLE   0,0         ;make sure we are at phase 0
        RATE    0,Hz(10)    ;start output at 10 Hz
        RATEW   0,Hz(11)    ;request 11 Hz next time around
CYCLE10: WAITC   0,CYCLE10   ;wait for the cycle>10Hz
CYCLE11: WAITC   0,CYCLE11   ;wait for the cycle>11Hz
        RATE    0,0         ;stop output
```

**See also:**
Cosine output, RATE set frequency change, RINC(W) change frequency, ANGLE change phase, WAITC wait for phase 0, CLRC clear new cycle flag, PHASE define phase zero, General DAC information, Expressions, Variables

# ANGLE

This changes the cosine angular position. It takes effect on the next instruction when the angle increment is added to the value set by this instruction and the result is output.

```
        ANGLE   n,expr|Vn|[Vn+off],OptLB     ;DAC n
```

n       DAC number in the range 0-7 (available DACs depend on the 1401 type).

expr    The phase angle to set in the range -360 up to +360.

Vn      For a variable, the value 11930465 is a phase of 1 degree (to be precise, 4294967296/360 is a phase of 1 degree). You can use the VAngle() function to convert degrees into a suitable value for a variable.

OptLB   If this optional label is present it sets the next instruction to run.

This example sets the phase angle to –90 degrees directly, and by using a variable. There is no need to use the VAngle() function; we could have set V1 to –1073741824. However, VAngle(-90) is much easier to understand.

```
        ANGLE   1,-90       ;set the DAC 1 cosine angle directly
        MOVI    V1,VAngle(-90)
        ANGLE   1,V1        ;set using a variable
```

**See also:**
Cosine output, RATE set frequency, RINC(W) change frequency, RATEW set frequency change at phase 0, PHASE define phase zero, General DAC information, Expressions, Variables

# PHASE

This changes the relative phase of the cosine output for the next cosine output. A common use is to change the output from a cosine (maximum value at phase zero) to sine (rising through zero at phase zero).

```
        PHASE   n,expr|Vn|[Vn+off],OptLB     ;DAC n
```

n       DAC number in the range 0-7 (available DACs depend on the 1401 type).

expr      The relative phase angle to set in the range -360 up to +360. The relative phase is set to 0 when sampling starts. Set –90 for sinusoidal output.

Vn      For a variable, the value 11930465 is a phase of 1 degree (to be precise, 4294967296/360 is a phase of 1 degree). You can use the `VAngle()` function to convert degrees into a suitable value for a variable.

OptLB      If this optional label is present it sets the next instruction to run.

This example plays a 1 Hz sinusoidal output (assuming that the output is not running).

```
PHASE   2,-90     ;set the DAC 2 phase angle directly
ANGLE   2,0       ;prepare to start as a sine wave
RATE    2,Hz(1)   ;start the sinusoid
```

**See also:**
Cosine output, RATE set frequency, RINC(W) change frequency, RATEW set frequency change at phase 0, General DAC information, Expressions, Variables

# OFFSET

This changes the cosine output voltage offset for the next cosine output.

```
OFFSET  n,expr|Vn|[Vn+off],OptLB     ;DAC n
```

n      DAC number in the range 0-7 (available DACs depend on the 1401 type).

expr      The offset value for sinusoidal output. The units of this value are as set in the outputs page of the sampling configuration dialog. It is an error to give a value that exceeds the DAC output range.

Vn      When a variable is used, the full range of the 32-bit variable corresponds to the full range of the DAC. You can use the `VDAC32()` function to load a variable using user-defined DAC units.

OptLB      If this optional label is present it sets the next instruction to run.

This example ramps DAC 0 from 0 to 1 volt, the runs 5 cycles of a sine wave at 1 Hz, and finally ramps the data back to 0 volts.

```
          DAC    0,0              ;use DAC 0 for all output
          OFFSET 0,1.0            ;set DAC 0 offset
          SZ     0,0.2            ;1 V sinusoid
          PHASE  0,-90            ;Prepare sinusoid
          ANGLE  0,0              ;set start point
          RAMP   0,1.0,1.0/s(1)   ;ramp to 1 volt in 1 sec
RAMPUP:   WAITC  0,RAMPUP         ;wait for ramp       >Ramp up
          RATE   0,HZ(1)          ;start sinusoid
          DELAY  S(4.9)           ;Sinusoid            >Sine
          RATEW  0,0              ;stop at cycle end
END:      WAITC  0,END            ;wait for end        >Wait end
          RATE   0,0              ;stop now
          RAMP   0,0.0,1.0/S(1)   ;ramp to 0 volt in 1 sec
RAMPDN:   WAITC  0,RAMPDN         ;wait                >Ramp down
          HALT
```

**See also:**
Cosine output, SZ cosine amplitude, SZINC amplitude change, General DAC information, Expressions, Variables

# WAITC

Each time the phase angle of a cosine passes through 0°, a new cycle flag sets. This flag is also set when a ramp terminates. There is a separate flag for each DAC. This flag is cleared by `CLRC`, `RATEW`, `RINCW` and when tested by `WAITC`.

```
WAITC   n,LB                          ;DAC n
```

n      DAC number in the range 0-7 (available DACs depend on the 1401 type).

LB      A label to branch to if the new cycle flag is not set. If the flag is set, the sequencer clears the flag and does not branch.

This instruction can produce a pulse at the start (or at least a known time after) the start of each waveform cycle. The following sequence outputs 4 cycles of waveform at different rates on DAC 1, and changes the digital outputs for each cycle.

```
        SZ      1,1.0       ;make sure full size
        ANGLE   1,0.0       ;make sure we start at phase 0
        RATE    1,1.0       ;1 degree per step to start with
        DIGOUT  [00000001]  ;so outside world knows
        RATEW   1,1.2       ;next cycle faster, clear cycle flag
w1:     WAITC   1,w1        ;wait for cycle >1 degree cycle
        DIGOUT  [00000010]  ;announce another cycle
        RATEW   1,1.4       ;next cycle a bit faster
w2:     WAITC   1,w2        ;wait for cycle >1.2 degree cycle
        DIGOUT  [00000011]  ;yet another one
        RATEW   1,1.6       ;last cycle a bit faster
w3:     WAITC   1,w3        ;wait for cycle >1.4 degree cycle
        DIGOUT  [00000100]  ;last cycle number
w4:     WAITC   1,w4        ;wait for end   >1.6 degree cycle
        RATE    1,0.0       ;stop waveform
```

**See also:**

Cosine output, RATE set frequency, RINC(W) change frequency, RATEW set frequency change at phase 0, ANGLE change phase, PHASE define phase zero, CLRC clear new cycle flag, General DAC information

# RINC RINCW

These instructions behave like `RATE` and `RATEW` except that they *change* the output rate (angle increment per step) by their argument rather than set it. `RINCW` clears the new cycle flag.

```
        RINC    n,expr|Vn|[Vn+off],OptLB    ;DAC n
        RINCW   n,expr|Vn|[Vn+off],OptLB    ;DAC n
```

n       DAC number in the range 0-7 (available DACs depend on the 1401 type).

expr    The change in the angle increment per step. You can use the built-in `Hz()` function to express the change as a frequency.

Vn      For a variable, the value 11930465 is a change of 1 degree. You can use the `VarValue` script in the `Scripts` folder to calculate variable values.

OptLB   If this optional label is present it sets the next instruction to run.

This example starts cosine output at 10 Hz and lets you adjust it from the keyboard.

```
        RATE    1,Hz(10)  ;start output at 10 Hz
wt:     JUMP    wt        ;HALT stops all output>P=+1Hz, M=-1Hz
     'P RINC    1,Hz(1),wt;1 Hz faster
     'M RINC    1,Hz(-1),wt;1 Hz slower
```

These instructions can be used to produce waveforms that change gradually in frequency. The following code generates a linear speed increase every two steps on DAC 1:

```
        SZ      1,1.0     ;make sure full size
        ANGLE   1,0.0     ;make sure we start at phase 0
        RATE    1,1.0     ;1 degree per step to start with
        MOVI    V1, 900   ;in 900 steps of...
loop:   CRINC   0.01      ;...1/100 degrees to...
        DBNZ    V1, loop  ;...10 degrees per step
```

The next example produces 90 cycles, each increasing by 0.1 degrees per step per cycle.

```
        SZ      1,1.0     ;make sure full size
        ANGLE   1,0.0     ;make sure we start at phase 0
        RATE    1,1.0     ;1 degree per step to start with
        MOVI    V1, 90    ;in 90 steps of...
loop:   RINCW   1,0.1     ;...1/10 degrees to...
wait:   WAITC   1,wait    ;...(wait for next cycle)...
        DBNZ    V1, loop  ;...10 degrees per step
```

**See also:**

Cosine output, RATE set frequency, RATEW set frequency change at phase 0, WAITC wait for phase 0, CLRC clear new cycle flag, PHASE define phase zero, General DAC information, Expressions, Variables

# CLRC

This instruction clears the cosine output new cycle flag. If you have been running for several cycles and you want to stop the next time phase 0 is crossed use this instruction immediately before using WAITC.

```
         CLRC    n,OptLB                          ;DAC n
```

n        DAC number in the range 0-7 (available DACs depend on the 1401 type).

OptLB    If this optional label is present it sets the next instruction to run.

This example starts a sinusoid and stops it at the next phase 0 crossing after a user stop requests. Because the sinusoid passes phase 0 in the WAITC instruction and does another step in the RATE 1,0 instruction, we offset the phase by 2 steps. However, this would cause the start of the sinusoid to be 2 steps wrong, so we change the start angle to match.

```
     'G PHASE   1,-2*Hz(2)    ; compenstate for ending
        ANGLE   1,2*Hz(2)     ; so we start in correct place
        RATE    1,Hz(2)       ; 2 Hz output
HERE:   JUMP    HERE          ; output is running >Running
     'S CLRC    1             ; Stop output
WT:     WAITC   1,WT          ; wait for cycle end>Waiting
        RATE    1,0,HERE      ; stop and then idle
```

**See also:**
Cosine output, RATE set frequency, RINC(W) change frequency, RATEW set frequency change at phase 0, WAITC wait for phase 0, PHASE define phase zero, General DAC information, Expressions, Variables

# General control

These instructions do not change any outputs or read data from any inputs. They provide the framework of loops, branches and delays used by the other instructions.

# DELAY

The DELAY instruction occupies one clock tick plus the number of extra ticks set by the argument. It produces simple delays of 1 to more than 4,000,000,000 sequencer steps.

```
         DELAY   expr|Vn|[Vn+off],OptLB
```

expr     The extra sequencer clock ticks to delay in the range 0 to 4294967295. The s(), ms() and us() built-in functions convert a delay in seconds, milliseconds or microseconds into sequencer steps.

Vn       Variable or table index from which to read the number of extra clock ticks.

OptLB    If this optional label is present it sets the next instruction to run.

This example uses display messages to tell the user what the sequence is doing.

```
         DELAY   2999      ;wait 2999+1 ticks        >3 second delay for 1ms ticks
         DELAY   s(3)-1    ;3 seconds -1 tick delay >3 second delay
         DELAY   V1,LB     ;wait V1+1 ms, branch     >variable delay
         DELAY   [V1+9]    ;V1+9 is table index      >table delay
```

**See also:**
Expressions, Variables, TABSZ and TABDAT directives, DBNZ count and branch for loops, CALL,CALLV,RETURN, JUMP to label, HALT the sequence, NOP do nothing

# DBNZ

DBNZ (Decrement and Branch if Not Zero) subtracts 1 from a variable and branches to a label unless the variable is zero. It is used for building loops.

```
        DBNZ    Vn,LB
```

Vn          The variable to decrement and test for zero.

LB          Instruction to go to next if the result of the decrement is not zero.

DBNZ is often used with MOVI to set up loops, for example:

```
        MOVI    V2,1000     ;set times to loop
WT:     DIGOUT  [00000000] ;set all digital outputs low
        DIGOUT  [11111111] ;set them all high
        DBNZ    V2,WT       ;loop 1000 times
```

**See also:**
Variables, DELAY wait for a number of steps, CALL,CALLV,RETURN, JUMP to label, HALT the sequence, NOP do nothing

# Compare variables

These instructions compare two variables or a variable and a 32-bit expression and branch on the result of the comparison. All comparisons are as signed 32-bit integers.

```
        Bxx     Vn,Vm,LB        ;compare with a variable
        Bxx     Vn,expr,LB      ;compare with a constant
        Bxx     Vn,[Vm+off],LB ;compare with a table entry
```

xx          This is the branch condition. The xx stands for: GT=Greater Than, GE=Greater or Equal, EQ=Equal, LE=Less than or Equal, LT=Less Than, NE=Not Equal.

Vn          The variable to compare with the next argument.

Vm          A variable to compare Vn with or table index variable.

expr        A 32-bit integer constant to compare Vn with.

This example collects the latest data value from channel 1 (assumed to be a waveform), waits for it to be in a set range for 1 second, then outputs a pulse to a digital output bit.

```
START:  CHAN    V1,1            ; get channel 1 data
        BGT     V1,4000,START   ; if above upper limit, wait
        BLT     V1,0,START      ; if too low, wait
IN:     MOVI    V2,S(1)/4       ; timeout, 4 instructions/loop
INLOOP: CHAN    V1,1            ; to check if still inside
        BGT     V1,4000,START   ; if above upper limit, wait
        BLT     V1,0,START      ; if too low, wait
        DBNZ    V2,INLOOP       ; see if done yet
REWARD: DIGOUT  [.......1]      ; Task done OK
        DELAY   S(1)            ; leave bit set for 1 second
        DIGOUT  [.......0]      ; clear done bit
        ...                     ; next task...
```

We want the data to be in range for one second. There are 4 instructions in the loop that tests this, so we set to the loop to run for the number of steps in a second divided by 4. For this to work correctly, the sequencer must be running fast enough so that 4 steps are no longer than the sample interval for the waveform channel.

**See also:**
Expressions, Variables, TABSZ and TABDAT directives

# CALL CALLV RETURN

These instructions run a labelled part of a sequence and return. CALL and CALLV save the next step number to a *return* stack and jump to the labelled instruction. The RETURN instruction removes the top step number from the *return* stack and jumps to it. CALLV also sets a variable to a constant.

```
        CALL   LB             ; use LB as a subroutine
        CALLV  LB,Vn,expr     ; Vn = expr, then call LB
        RETURN                ; return to step after last CALL
```

LB      The next instruction to run. The CALLed section should end with a RETURN.

Vn      CALLV copies the value of expr to this variable. CALL1 sets V33, CALL2 sets V34 CALL3 sets V35 and CALL4 sets V36.

expr    A 32-bit integer constant that is copied to a variable.

You can use CALL inside a CALLed subroutine. This is known as a *nested* CALL. If you call a subroutine from inside itself, this is known as a *recursive* CALL. The return stack has room for 64 return addresses. If you use more than this, the oldest return address is overwritten, so your sequence will not behave as you expect.

This example generates different pulse widths from DAC 0. The sequence is written to be independent of the sequencer rate, However, it must be high enough so that the widths are possible. In this case a sequencer **Step period** of 1 millisecond (set in the **Outputs** section of the sampling configuration) would be fine. The example sets DAC 0 to zero, then pulses for 20 milliseconds twice, once using CALL and once using CALLV. Then after a delay, there is a 50 millisecond pulse.

```
        DAC    0,0            ; make sure DAC0 is zero
        MOVI   V3,ms(20)-2    ; these two instructions...
        CALL   PUL            ; ...have the same effect as...
        CALLV  PUL,V3,ms(20)-2; ...this one. 20 ms pulse
        DELAY  s(1)-1         ; wait 1 second, then...
        CALLV  PUL,V3,ms(50)-2; ...a 50 ms pulse
        HALT                  ; So we don't fall into PUL routine
PUL:    DAC    0,1            ; set DAC value
        DELAY  V3             ; wait for time set
        DAC    0,0            ; set DAC back to zero
        RETURN                ; back to the caller
```

CALL/CALLV and RETURN let you reuse a block of instructions. This can make sequences much easier to understand and maintain. The disadvantage is the additional steps for the CALL and RETURN. If you need to set a variable, use CALLV and there is only the overhead of the RETURN instruction.

**See also:**
Expressions, Variables, DELAY wait for a number of steps, DBNZ count and branch for loops, JUMP to label, HALT the sequence, NOP do nothing

# JUMP

The JUMP instruction transfers control unconditionally to the instruction at the label. Many instructions allow the use of an optional label to set the next instruction, so you can often avoid the need for this instruction. You can also jump using the contents of a register as the destination, or relative to a label (LB):

```
        JUMP   LB             ; Jump to label
        JUMP   (Vn),OptLB     ; Jump to instruction Vn
        JUMP   LB(Vn),OptLB   ; Jump to instruction LB+Vn
```

(Vn)    The value of variable Vn sets the instruction number to jump to.

LB(Vn)  Jump to the instruction given by label LB plus the contents of Vn.

OptLB   An optional label to jump to if (Vn) or LB(Vn) is not an instruction number. The first instruction is 0, the last depends on the size of the sequence.

**Multiple states**

You can use the JUMP instruction with a variable holding an offset from a label to design a sequence that does different things according to the current sweep state. As an example:

```
        VAR    V1,CurSta              ; A variable to hold the sweep state number

        WSWP   0                      ; Wait until a sweep is in progress - not needed if t
        STATE  CurSta                 ; Get the state code (0 to n) in a variable
        JUMP   JTAB(CurSta),DONE       ; Jump into the table or to DONE if the state code is
DONE:   JUMP   DONE                   ; Finish off here - does nothing
```

```
; State 1 generates one digital pulse
S1:         WSWP   s(0.1)-1                 ; Generate a trigger pulse at 0.1 seconds
            DIGOUT [......1]
            DELAY  10                       ; We could be more precise about timing if we wanted...
            DIGOUT [......0],DONE

; State 2 generates two digital pulses
S1:         WSWP   s(0.1)-1                 ; Generate a trigger pulse at 0.1 seconds
            DIGOUT [......1]
            DELAY  10                       ; We could be more precise about timing if we wanted...
            DIGOUT [......0]
            WSWP   s(0.2)-1                 ; Generate a second trigger at 0.2 seconds
            DIGOUT [......1]
            DELAY  10
            DIGOUT [......0],DONE

; This is the table of jumps, each one of which goes to the code to handle a separate sweep stat
; JUMP above jumps into the table, then it jumps out to the relevant code. We put the table at t
; so that if the state code is too high for the table we automatically jump to DONE.
JTAB:       JUMP   DONE                     ; State 0, does nothing
            JUMP   S1                       ; Jump to state 1 code
            JUMP   S2                       ;  and so on
....                                        ; The table should have an entry for each state in use
```

### State machine

You can also use the JUMP instruction to implement a state machine. A state machine is characterised by activities in each state, and transitions between states. State machines are a convenient way to tidy up a complicated sequence of operations involving conditions such as the level of an input signal. For example:

| State | Activity | Transition |
|---|---|---|
| 0 | Wait for digital input 0 to be low | To state 1 |
| 1 | Wait for digital input 0 to go high | To state 2 on high |
| 2 | Wait for digital input 0 to go low | To state 3 on low,  digital output high on change |
| 3 | Wait for 10 seconds | To state 0, digital output low on change |

This could be coded as:

```
            VAR    V1,StaV=State0  ;initial state
            VAR    V2,Until=0      ;Used to time state 3
            VAR    V3,Now

IDLE:       ...                    ;background actions >=
            JUMP   (StaV)          ;run state machine  >=

STATE0:     DIBNE  [.......0],IDLE ;wait for low       >State 0
            MOVI   StaV,State1,IDLE ;move on           >"
STATE1:     DIBNE  [.......1],IDLE ;wait for high      >State 1
            MOVI   StaV,State2,IDLE ;                  >"
STATE2:     DIBNE  [.......0],IDLE ;wait for low again >State 2
            DIGOUT [.......1]      ;Digital out high   >"
            TICKS  Until,s(1)      ;1 second ahead     >"
            MOVI   StaV,State3,IDLE ;                  >"
STATE3:     TICKS  Now,0          ;get current time    >State 3
            BLT    Now,Until,IDLE  ;wait for 1 second  >"
            DIGOUT [.......0]      ;Digital out low    >"
            MOVI   StaV,State0,IDLE
```

The ... at label IDLE stands for instructions that you want to run in the background while the state machine runs - probably handling on-going sampling. Of course, the more background instructions you include, the less frequently the state machine checks the state of the inputs.

### See also:

DELAY wait for a number of steps, DBNZ count and branch for loops, CALL,CALLV,RETURN, HALT the sequence, NOP do nothing

# HALT

The `HALT` instruction stops the output sequence and removes all overhead associated with it. It does not stop the sequencer clock, which continues to run. Any cosine output will stop, but will restart when the sequence restarts. To restart the sequencer, press a key associated with a sequence step or click a key in the sequencer control panel. If you associate a display string with this instruction, it appears in the sequencer control panel.

```
        HALT                    >Press X when ready
```

**See also:**
DELAY wait for a number of steps, DBNZ count and branch for loops, CALL,CALLV,RETURN, JUMP to label, NOP do nothing

# NOP

The `NOP` instruction (`No OP`eration) does nothing except use up one sequencer clock tick. It can be thought of as the equivalent of `DELAY 0`.

**See also:**
DELAY wait for a number of steps, DBNZ count and branch for loops, CALL,CALLV,RETURN, JUMP to label, HALT the sequence

# Variable arithmetic

These instructions perform basic mathematical functions while a sequence runs. You can also compare variables and branch on the result

# MOVI

This instruction moves an integer constant into a variable. The syntax is:

```
        MOVI    Vn,expr,OptLB     ; Vn = expr
```

Vn      A variable to hold the value of `expr`.

expr    An expression that is evaluated as a 32-bit integer.

OptLB   If this optional label is present it sets the next instruction to run.

`MOVI` is not the same as the `VAR` directive. The `VAR` directive sets the value of a variable when the sequence is copied to the 1401 and does not occupy a step. The `MOVI` instruction is part of the sequence and set the value of the variable each time the instruction is used.

**See also:**
Expressions, Variables, Set variable to another variable

# MOV ABS NEG

The `MOV` instruction sets a variable to the value of another with the option of adding a 32-bit number and dividing by a power of two). The `NEG` instruction is identical to `MOV` except that the source variable is negated first. `ABS` is also the same, except that the absolute value (negative values become positive) of the variable is taken first. `ABS` was added at version 4.06. The syntax is:

```
        MOV     Va,Vb,expr,shift   ; Va = (Vb + expr) >> shift
        NEG     Va,Vb,expr,shift   ; Va = (-Vb + expr) >> shift
        ABS     Va,Vb,expr,shift   ; Va = (|Vb| + expr) >> shift
```

Va      A variable to hold the result. It can be the same as `Vb`.

Vb      A variable used to calculate the result. It is not changed unless it is the same variable as `Va`.

expr    An optional expression that is evaluated as a 32-bit integer. If this argument is omitted, it is treated as 0.

shift      An optional argument in the range 0 to 31, set to 0 if omitted, that sets the number of times to divide the result by 2.

The following examples assume that `V3` holds 1000:

```
        VAR     V6,Result
        MOV     V1,V3                ; set V1 to 1000
        NEG     V1,V3                ; set V1 to -1000
        MOV     V1,V3,-8             ; set V1 to 992
        NEG     Result,V3,0,4        ; set V6 to -63
        ABS     Result,Result        ; set V6 to 63
        MOV     Result,V3,4,1        ; set V6 to 502
```

**See also:**
Expressions, Variables, Set variable to constant

# ADDI

This instruction adds a 32-bit integer constant to a variable. There is no `SUBI` as you can add a negative number. The syntax is:

```
        ADDI    Vn,expr,OptLB        ; Vn = Vn + expr
```

Vn      A variable to hold the result of `Vn + expr`.

expr      An expression that is evaluated as a 32-bit integer.

OptLB      If this optional label is present it sets the next instruction to run.

The following examples assume that `V1` holds -1000:

```
        VAR     V1,Result=-1000
        ADDI    Result,1000          ; set V1 to 0
        ADDI    V1,-4000             ; set V1 to -5000
```

**See also:**
Expressions, Variables, Add variable to variable

# ADD SUB

The `ADD` instruction adds one variable to another. The `SUB` instruction subtracts one variable from another. In both cases you can optionally add a 32-bit integer constant and optionally divide the result by a power of two. The syntax is:

```
        ADD     Va,Vb,expr,shift ; Va = (Va + Vb + expr) >> shift
        SUB     Va,Vb,expr,shift ; Va = (Va - Vb + expr) >> shift
```

Va      A variable to hold the result. It can be the same as `Vb`.

Vb      A variable to add or subtract.

expr      An optional expression evaluated as a 32-bit integer. If omitted, 0 is used.

shift      An optional argument in the range 0 to 31, set to 0 if omitted, that sets the number of times to divide the result by 2.

The following examples assume that `V1` holds -1000, `V3` holds 1000, `V6` holds 100:

```
        VAR     V6,Result=100
        ADD     V1,V3                ; V1 = 0 (-1000 + 1000 + 0)
        SUB     V1,V3                ; V1 = -1000 (0 - 1000 + 0)
        ADD     V1,V3,-8             ; V1 = -8 (-1000 + 1000 - 8)
        SUB     Result,V3,0,2        ; V6 = -225 (100 - 1000 + 0)/4
        ADD     Result,V3,4,1        ; V6 = 389 (-225 + 1000 + 4)/2
```

**See also:**
Expressions, Variables, Add constant to variable

## DIV RECIP

`DIV` and `RECIP` divide variables. These are relatively slow instructions; they take around .4 microseconds in a Power1401 mk II, a little less in a Power1401-3, around 1 microsecond in the Power1401 mk I, 3 microseconds in a micro1401 mk II or Micro1401-3 and 10 microseconds in a micro1401 mk 1.

```
        DIV     Va,Vb               ; Va = Va / Vb
        RECIP   Va,expr             ; Va = expr / Vb
```

If the numerator is 0, the result is 0. If the denominator is 0, the result is 2147483647 if the numerator is greater than 0 and –2147483648 if it is negative. All modern 1401s truncate towards 0. So, 7/3 or –7/–3 is 2 in all 1401s, but –7/3 or 7/–3 is –2.

## MUL MULI

`MUL` multiplies a variable by another variable, then optionally adds a 32-bit integer constant and divides the result by a power of two. `MULI` multiplies a variable by a 32-bit integer constant and divides the result by a power of 2.

```
        MUL     Va,Vb,expr,shift ; Va = ((Va*Vb)+expr) >> shift
        MULI    Va,expr,shift    ; Va = (Va*expr) >> shift
```

`Va`      A variable to hold the result. It can be the same as `Vb`.

`Vb`      A variable used to calculate the result.

`expr`    An expression that is evaluated as a 32-bit integer. It is optional for `MUL` and required for `MULI`. If this argument is omitted, it is treated as 0.

`shift`   An optional argument in the range 0 to 31, set to 0 if omitted, that sets the number of times to divide the result by 2.

The following examples assume that `V1` holds –10 and `V3` holds 10:

```
        MULI    V1,10               ; V1 = -100 (-10 * 10)
        MUL     V1,V3,-8            ; V1 = -992(-100 * 10 -8)
```

**See also:**
Expressions, Variables

## Variable logic

These instructions perform basic bitwise logical functions between two variables or a variable and a constant while a sequence runs.

## AND, ANDI

These instructions bitwise `AND` a variable with a variable or a 32-bit expression. A bitwise `AND` means that each bit of the 32-bit result is 1 if both corresponding source bits are 1, otherwise the result bit is 0. For example, 3 `AND` 1 is 1, 0x55 `AND` 0xAA is 0.

```
        AND     Va,Vb        ; Va = Va AND Vb
        ANDI    Va,Vb,expr   ; Va = Vb AND expr
```

`Va`      The variable to hold the result.

`Vb`      A variable to `AND` with `Va` or with the expression.

`expr`    A 32-bit integer constant to `AND` with `Va`.

This example waits for the digital input to have bit 4 set, then branches based on the digital input value (placed in `VDigIn` or `V56` by `DIBNE`).

```
WT:       DIBNE    [...1....],WT   ; wait for bit 4 set >Wait for Bit 4
          ANDI     V1,VDigIn,0x0f  ; isolate bits 0..3 (value 0-15)
          JUMP     ACTION(V1)      ; branch based on the result
ACTION: JUMP       ACT0            ; action for value 0
          JUMP     ACT1            ; action for value 1
          ...
          JUMP     ACT15           ; action for value 15
```

**See also:**
Expressions, Variables, Constants

# OR, ORI

These instructions bitwise OR a variable with a variable or a 32-bit expression.  A bitwise OR means that each bit of the 32-bit result is 1 if either corresponding source bit is 1, otherwise the result bit is 0. For example, 3 OR 1 is 3, 0x55 OR 0xAA is 0xff.

```
          OR       Va,Vb           ; Va = Va OR Vb
          ORI      Va,Vb,expr      ; Va = Vb OR expr
```

Va       The variable to hold the result.

Vb       A variable to OR with Va or with the expression.

expr     A 32-bit integer constant to OR with Va.

**See also:**
Expressions, Variables, Constants

# XOR, XORI

These instructions bitwise exclusive OR a variable with a variable or a 32-bit expression.  A bitwise exclusive OR means that each bit of the 32-bit result is 1 if the corresponding source bits differ, otherwise the result bit is 0. For example, 3 XOR 1 is 2, 0x55 XOR 0xAA is 0xff.

```
          XOR      Va,Vb           ; Va = Va XOR Vb
          XORI     Va,Vb,expr      ; Va = Vb XOR expr
```

Va       The variable to hold the result.

Vb       A variable to XOR with Va or with the expression.

expr     A 32-bit integer constant to XOR with Va.

**See also:**
Expressions, Variables, Constants

# Table access

Tables are declared with the TABSZ directive and can be populated with data using the TABDAT directive. Most access to tables is through the [Vn+off] method, but there are also instructions for loading and storing a register in a table, for adding and subtracting table values from a variable, and for incrementing or decrementing a register used as a pointer into the table.

# TABLD TABST

These two instructions load a variable from the table and store a variable into the table. Many instructions can load arguments from the table, so TABLD is not often required.

```
          TABLD    Vm,[Vn+off],OptLB  ; load Vm from the table
          TABST    Vm,[Vn+off],OptLB  ; store Vm into the table
```

Vm       The variable to load from the table or store into the table.

| | |
|---|---|
| `+off` | An optional expression that evaluates to an integer in the range –1000000 to 1000000. If omitted, the value 0 is used. |
| `Vn` | Any offset in the `off` expression is added to the contents of this variable and the result is used as an index into the table. If the index lies in the table, `Vm` is loaded from the table or stored in the table at the index. If the index is outside the table, `TABLD` copies 0 to `Vm` and `TABST` does nothing. |
| `OptLB` | If this optional label is present it sets the next instruction to run. |

**See also:**
TABSZ and TABDAT directives, Variables, TABINC

# TABADD TABSUB

These two instructions add a table value to a variable or subtract a table value from a variable. These instructions were added at version 4.06.

```
        TABADD   Vm,[Vn+off],OptLB  ; add table value to Vm
        TABSUB   Vm,[Vn+off],OptLB  ; subtract value from Vm
```

| | |
|---|---|
| `Vm` | The variable to add data to or subtract it from. |
| `+off` | An optional expression that evaluates to an integer in the range –1000000 to 1000000. If omitted, 0 is used. |
| `Vn` | The variable value plus the offset is used as a table index. If the index lies in the table, `Vm` is changed. If the index is not in the table, the instruction does nothing. |
| `OptLB` | If this optional label is present it sets the next instruction to run, otherwise the next sequential instruction runs. |

**See also:**
TABSZ and TABDAT directives, Variables, TABINC, TABLD, TABST

# TABINC

This instruction adds a constant to a variable and detects if the result is a valid table index. If it is a valid index, the instruction branches. If it is not, the result is reduced by the table size if it is positive and is increased by the table size if it is negative, and the instruction does not branch. This gives you an efficient way to work through the table.

```
        TABINC   Vn,expr,OptLB
```

| | |
|---|---|
| `Vn` | This variable is assumed to hold a valid table index. |
| `expr` | This expression evaluates to a positive or negative number that is added to `Vn`. |
| `OptLB` | If this optional label is present it sets the next instruction to run. |

For example, the following codes plays pulses through DAC 0 based on data in the table. The table data holds groups of three items, holding the time for the DAC to stay at 0, the DAC amplitude and the time to stay at the amplitude. Some example table data is given, but the data could also be set with the `SampleSeqTable()` script command.

```
        TABSZ   12               ;4 sets of 3 items
        TABDAT  ms(50)-2,VDAC32(1),ms(50)-3
        TABDAT  ms(100)-2,VDAC32(1.3),ms(70)-3
        TABDAT  ms(200)-2,VDAC32(1.5),ms(90)-3
        TABDAT  ms(400)-2,VDAC32(1.9),ms(110)-3
     'G MOVI    V1,0             ;use V1 as the table pointer
LOOP:   DAC     0,0              ;strt with the DAC low
        DELAY   [V1]             ;wait for first period>Low
        DAC     0,[V1+1]         ;get the DAC value
        DELAY   [V1+2]           ;wait for second period>High
        TABINC  V1,3,LOOP
        DAC     0,0              ;tidy up the dac
```

**See also:**

TABSZ and TABDAT directives, Variables, TABLD,TABST

# Access to data capture

Most activities in the sequencer are independent of the sampling process. However, there are times when you need to know the value of a channel to decide what to do next or you need to trigger a sweep.

# CHAN

This instruction gives the output sequencer access to sampled data on a waveform channel. You can also use this command to get the most recent value written to the DAC outputs. The variable value is set to 0 if the channel is not being sampled.

```
          CHAN    Vn,chn            ; Vn = ChanData(chn)
```

chn     The channel number is 1 to 80 for sampled channels or 0 to -7 for the last value on DACs 0 to 7. The result is the most recent data available.

Waveform and DAC data are treated as 16-bit signed values from –32768 to 32767. You also have access to DAC values as 32-bit data in variables V57 to V64 (VDAC0 to VDAC7) without the need to use the CHAN instruction. This instruction takes rather longer to execute than other sequencer instructions (it incurs the DIV/RECIP time penalty), and may cause timing problems if used in circumstances when the 1401 is heavily loaded.

This example waits for a sampled signal to cross 0.05 volts and produces a pulse.

```
          VAR     V1,level=VDAC16(0.05) ;level to cross
          VAR     V2,data           ;to hold the last data
          VAR     V3,low=0          ;some sort of hysteresis level
          DIGOUT  [00000000]        ;set all dig outs low
BELOW:    CHAN    data,1            ;read latest data    >wait below
          BGT     data,low,below    ;wait for below      >wait below
ABOVE:    CHAN    data,1            ;read latest data    >wait above
          BLE     data,level,above  ;wait for above      >wait above
          DIGOUT  [.......1]        ;pulse output...
          DIGOUT  [.......0],below  ;...wait for below
```

**See also:**
Variables

# DCON

This instruction is used to turn a dynamic clamp model on or off.

```
          DCON    num,on,OptLb          ; num = model, on = on/off switch
```

num     An expression that sets the model number, from 1 to the maximum available. Models are numbered in the order they are shown in the main dynamic clamp setup dialog, with the first model being number 1.

on      An expression that turns the model on or off, set this to 1 to enable a model, 0 to disable it.

OptLb   If present, the instruction branches to this label.

This instruction can be used in conjunction with the Signal dynamic clamping mechanism to turn models on and off at particular times within the sampling sweep. Note that this command only operates upon the model information currently in the 1401 and that this information is updated by Signal whenever the user updates model data and whenever the sampling sweep state changes in multiple states sampling, and that this updating process will override changes made by this instruction with the enable or disable flags associated with the main model data.

**See also:**
Variables, Dynamic clamping

# POINTS

This instruction sets a variable to the current number of points sampled in the current sweep. The variable value is set to 0 if the sampling sweep has not started.

```
        POINTS   Vn,OptLB        ; Vn = Sweep points
```

Vn          This variable is updated with the sweep point count.

OptLB       If present, the instruction branches to this label.

This can be used instead of the `WSWP` command to wait until a specific point in the sweep, but based on points rather than time. This instruction takes rather longer to execute than other sequencer instructions (it incurs the DIV/RECIP time penalty), and may cause timing problems if used in circumstances when the 1401 is heavily loaded.

**See also:**
Variables, POINTS get sampled pointsVariables

# STATE

This instruction sets a variable to the current sweep state code.

```
        STATE    Vn,OptLb        ; Vn = Sweep state code
```

Vn          This variable is updated with the current sweep state code.

OptLb       If present, the instruction branches to this label.

This can be used in conjunction with the Signal multiple states mechanism (in Static or Dynamic outputs mode only) to produce a sequence that behaves in a different fashion according to the sweep state.

**See also:**
Variables

# SETS

This instruction sets the state code for the current sweep.

```
        SETS    expr|Vn|[Vn+off],OptLB    ; Sweep state = expr
```

expr        This is the state code, from 0 to 255 normally.

Vn          When a variable or table entry is used to set the state, the value sets the sweep state.

OptLb       If present, the instruction branches to this label.

This can be used instead of the normal Signal multiple states mechanisms to generate data files with separate state code numbers for each data sweep. This instruction should not be used with a sampling configuration which uses multiple states as the two mechanisms will be in conflict; use it in a sampling configuration with multiple states turned off. To match the built-in multiple states mechanism you should restrict yourself to state codes from 0 to 255, however you can use other state code values if you wish.

To find the state number for the current sweep when using the built-in multiple states mechanism, use the STATE instruction.

**See also:**
Expressions, Variables

# SWEEP

This sets a variable to the start time of the current sweep in sequencer clock ticks plus an optional expression.

```
        SWEEP    Vn,expr         ; Vn = Sweep start time + expr
```

Vn          This variable is updated with the start time of the current sweep.

expr        This optional expression will be added to the sweep start time as expressed in sequencer clock ticks.

This can be used to find the start time of the current sweep, or a time within the current sweep. If there is no sweep in progress, the start time of the previous sweep is used.

**See also:**
Expressions, Variables

# WSWP

This instruction waits until the specified time (in sequencer ticks) within the sampling sweep is reached (or has been passed).

```
          WSWP    expr|Vn|[Vn+off],OptLB   ; Wait till expr in sweep
```

expr        This is time within the sweep, in sequencer ticks. Values of `expr` from 1 to the sweep duration specify a time within a sweep, if you specify a time greater than the sweep duration the sequencer will wait forever. The following values of `expr` have special meanings:

    0   Wait until sampling of sweep data is in progress
   -1   Wait until a sampling of sweep data is not in progress
   -2   Wait until the sampling sweep is armed; the 1401 is ready to accept a trigger but has not been triggered yet

Vn          When a variable or table entry is used, the value is the time within the sweep in sequencer ticks or one of the special values above.

OptLb       If present, the instruction branches to this label when the wait is completed.

This can be used to pause sequencer execution until a required sweep time is reached; it is the easiest way of synchronising the sequencer with sampling. If you want to perform some activity until a certain point in the sweep is reached rather than just waiting, you can use the `TICKS` sequencer instruction to do this.

**See also:**
Expressions, Variables

# TRIG

This instruction causes a trigger to start a sampling sweep. If the **Free run without restarts** box is checked, you can use this in **Basic**, **Extended** and **Fast trigger sweep modes**. If the box is clear, you can use it in **Extended** mode only. Using this instruction in other circumstances may cause sampling problems.

```
          TRIG                 ; Trigger a sweep
```

In **Extended** sampling mode, you <u>must</u> use `TRIG` to trigger sweeps as external triggers are disabled.

# TICKS

This instruction sets a variable to the current sampling time in sequencer clock ticks and adds an expression or 0 if `expr` is omitted. Note that this time is the time since sampling started, not the time since the sampling sweep started. The `s()`, `ms()` and `us()` expression functions can be used to make the sequence independent of the clock rate.

```
          TICKS   Vn,expr          ; Vn = Signal time + expr
```

Vn          This variable is updated with the current time.

expr        This optional expression will be added to the time.

This can be used with the `CHAN` command and variable related branches to check the timing of external pulses. The sequencer runs under interrupt, and competes for time with other interrupt driven processes in the 1401 interface. This causes some "jitter" in the timing. The jitter for a Micro1401 or Power1401 is typically only a few microseconds, often less than 1 microsecond for fast modern units.

It is often necessary to perform some activity such as testing a voltage against a threshold until a given point in the sampling sweep is reached. This can be done using the TICKS instruction along with SWEEP as in this example fragment.

```
        VAR     V1,level=VDAC16(0.05) ;level to cross
        VAR     V2,data         ;to hold the last data
        VAR     V3,limit        ;when to stop looking
        VAR     V4,now          ;to hold current time

        MOV     limit,start     ;get the sweep start time
        ADD     limit,ms(100)   ;wait until 100 ms into sweep

WATCH:  TICKS   now             ;current time in variable
        BGE     now,limit,DONE  ;give up if too late
        CHAN    data,1          ;get newest channel 1 data
        ABS     data,data       ;convert to absolute magnitude
        BGT     data,level,TRIG ;above level, go and do something
        JUMP    WATCH           ;keep going round
```

**See also:**
Expressions, Variables

# REPORT MARK

The REPORT instruction records a digital marker (if the digital marker channel is enabled) as if there was an external pulse on the "data available" input (see under *Sampling data* for the input to use). The MARK instruction does the same, except it takes the argument as the value to record. REPORT has no arguments.

```
        REPORT  OptLB
        MARK    expr|Vn|[Vn+off],OptLB
```

expr    The argument should have a value in the range 0 to 255. If a variable or table is used, the bottom 8 bits of the value are used.

OptLB   If this optional label is present it sets the next instruction to run, otherwise the next sequential instruction runs.

```
LB:     DIBNE   [.......1],LB >Waiting for bit 0
        REPORT                  ;save a marker when this is set
        MARK    12              ;set code 12 as a digital marker
```

**See also:**
Output connections, Expressions, Variables, TABSZ and TABDAT directives

# Randomisation

These functions use a pseudo-random number generator. The generator is seeded by a number that is based on the length of time that the 1401 has been switched on.

# BRAND

BRAND branches with a probability set by the argument or by a variable. This could be used when several different stimuli are required, but in a random sequence.

```
        BRAND   LB,expr|Vn|[Vn+off]
```

LB      Where to go if the branch is taken

expr    This is the probability of branching in the range 0 up to (but not including) 1.

Vn      When a variable or table entry is used for a branch, the value is treated as a 32-bit unsigned number; 0 means a probability of zero and 4294967295 (the largest 32-bit unsigned number) means a probability of 0.9999999998.

```
        BRAND   LB,0.5    ;branch with 50% probability
```

To produce a multiple way random branch you use more BRAND instructions. A three way equal probability branch to LA, LB and LC can be coded:

```
        BRAND   LA,0.33333  ;Split the first route with p=1/3
        BRAND   LB,0.5      ;0.6667 to here * 0.5 is 0.3334 (1/3)
LC:     ...                 ;If neither of the above, comes here
```

The following shows the sequence for a five-way branch with equal probabilities:

```
        BRAND   LA,0.2  ;5 way, LA probability is 0.2 (1/5)
        BRAND   FX,0.5  ;Probability to here=0.8, so to FX=0.4
        BRAND   LB,0.5  ;Probability to here=0.4, so to LB=0.2
LC:     ...             ;Probability to here=0.2
FX:     BRAND   LD,0.5  ;Probability to here=0.4, so to LD=0.2
LE:     ...             ;Probability to here=0.2
```

The best technique is to reduce the branches to a power of two as soon as possible. Case 1 of the five-way branch is split off (probability of 0.2), leaving 4 ways. The 4 ways are split with a probability of 0.5 (0.4 for each division) then the last two routes are split, again with a probability of 0.5 (0.2 for each division).

### Poisson process

In a Poisson process, the probability of something happening per time interval is constant. You can generate a delay with a Poisson statistic by:

```
POISSON: BRAND   POISSON,prob ; poisson delay
```

The probability is given by prob = 1.0 - 1.0/(mDelay*S(1)), where mDelay is the mean delay required in seconds and S(1) is the built in function that tells us how many steps there are per second. If you would rather express this in terms of a rate, then prob = 1.0 - rate/S(1), where rate is the expected rate in Hz.

```
TENHZ:   BRAND   TENHZ,1.0-10/S(1) ;10 Hz mean rate
         DIGOUT  [.......1]        ;set output high
         DIGOUT  [.......0],TENHZ  ;set output low, goto TENHZ
```

This example generates a digital output that pulses to produce an approximation to a Poisson distributed pulse train with a mean frequency of 10 Hz. The approximation improves the shorter the step time. The mean interval between pulses is 100 milliseconds plus the time for 2 steps and the shortest gap between pulses is 3 sequencer steps.

### Scripts and variables

From a script you can set sequencer variables as 32-bit signed integers. For the range 2147483648 to 4294967295 we must use negative numbers. This script example shows you how to convert a probability into a variable value and pass it to the sequencer:

```
Proc SetBrandVar(prob, v%) 'prob is probability, v% is variable
prob *= 4294967296.0;       'range 0-4294967296 is 0 to 1.0
if (prob > 4294967295.0 ) then prob := 4294967295.0 endif;
if prob > 2147483647 then prob -= 4294967296.0 endif;
if prob < -2147483647 then prob := -2147483647 endif;
SampleSeqVar(v%, prob);
end;
```

### See also:

Expressions, Variables, TABSZ and TABDAT directives, MOVRND set a variable to a random number

# MOVRND

This instruction generates a random number in the range 0 to a power of 2 minus 1, then adds an integer constant to it and stores the result in a variable.

```
        MOVRND  Vn,bits,expr
```

Vn      The variable to hold the result.

bits    The number of random bits to generate in the range 1 to 32. The generated random bits fill the variable starting at the least significant bit. Bits above the highest numbered generated bit are set to 0.

expr    An optional expression that evaluates to a 32-bit integer number, that is added to the random bits. If this is omitted, nothing is added.

Expressed in terms of the script language, the random number is one of the numbers in the range `expr` to `expr+Pow(2,bits)-1`. For example, `MOVRND V33,8,1` generates a random number in the range 1 to 256.

The following code fragment implements a random delay of between 1 and 2.024 seconds (assuming a 1 millisecond clock).

```
        MOVRND  V1,10,998 ;load V1 0 with (0 to 1023) + 998
        DELAY   V1        ;this uses 999 to 2023 steps
```

**See also:**
Expressions, Variables

# Arbitrary waveform output

In addition to generating voltage pulses, ramps and cosine waves through the DACs, Signal can play arbitrary waveforms. The sequencer can start waveform output from one of the available areas and also test the state of waveform output and branch on the result. The sampling configuration sets the size of the areas used for waveform storage and the number of these areas, not all of a waveform area need be used. The waveforms are stored in 1401 memory and can be updated just before and during sampling using the `SampleSeqWave()` script command.

# WAVE

The `WAVE` instruction triggers the start of arbitrary waveform output from a waveform area.

```
        WAVE    area|Va,len|Vl,OptLb                 ; Start arbitrary waveform output
```

`area`   If present, an expression that sets the waveform area to use, from 1 to the maximum available. If this is omitted area 1 is used.

`Va`   If present, a sequencer variable that holds the area number as above.

`len`   If present, an expression that sets the number of waveform points to be output, set to zero or omit this argument to output all of the points in the area. Note that this value is points per DAC, not total points.

`Vl`   If present, a sequencer variable that holds the output points as above.

`OptLb`   If present, the instruction branches to this label.

Note that, in versions of Signal before 5.00, there was only a single waveform output area and the WAVE instruction had no arguments. For compatibility with existing sequences, all of the arguments are optional and, if omitted, it will output all of the points from waveform area number 1. The waveform output areas used by the sequencer can only be loaded up with data by using the `SampleSeqWave()` script language function, waveform areas can be reloaded with different data during sampling as required.

# WAVEBR

The `WAVEBR` instruction tests the state of the waveform output and branches on the result. No branch occurs if there is no output running or requested.

```
        WAVEBR  LB,flag
```

`LB`   Label to branch to if the condition set by the flag is met.

`flag`   An optional single character flag to specify the branch condition:

    `S`     branch if arbitrary waveform output is stopped.
    `G`     branch if arbitrary waveform output is going.

The following (fairly trivial) sequence plays the output waveform 5 times.

```
        MOVI    V1,5        ; load variable 1 with 5
WL:     WAVE                ; start output
WT:     WAVEBR  WT,G        ; wait here while output is going
        DBNZ    V1,WL       ; do this 5 times  >Waiting for cycle
```

The `WAVE` command starts output, it then waits for the output to stop at the `WT` label. Next the sequence repeats this process 5 times.

**See also:**
Control from a script, WAVE start waveform output

# Sampling with multiple states

Multiple states sampling is one of the most powerful features in Signal. Multiple states can be used to do a lot of things, by far the most common (and straightforward) is to allow a sampling configuration to have a number of different sets of output pulses available (one per state) and to switch between these outputs during sampling. So, for example, you could have one state that generates a single stimulation pulse on a DAC, another state that has two stimulation pulses separated by 20 milliseconds and another that has two pulses separated by 40 milliseconds, and Signal could then switch between them randomly or in a preset order during sampling. This form of multiple states is known as Dynamic outputs, we will concentrate on this type of multiple states usage because it is the most useful and most general-purpose.
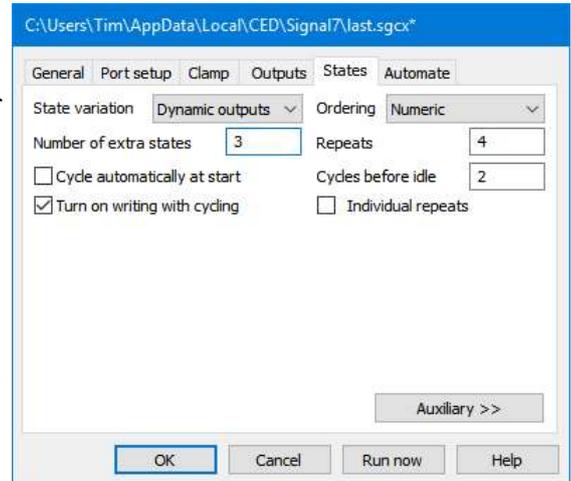


### State numbers, idling and state 0

With multiple states sampling disabled all sampling uses the only state available, which is number zero. With multiple states in use you gain a number of extra states, which are numbered from one upwards. The design of Signal expects (though it does not require) that state zero will be reserved for passive or idle outputs rather than for outputs that will form part of the main experimental data – so for example state zero might generate no stimulus at all or maybe a sub-threshold stimulus that allows the health of the preparation to be checked but does not generate any useful data.

This design choice is most visible in the 'idling' behavior of Signal states sequencing. Idling means that Signal will stop states sequencing, switch to state zero, and turn off writing to disk. Controls in the states configuration allow you to specify when states sequencing will automatically idle; there is also an Idle button in the states control bar. The built-in states sequencing within Signal also expects this arrangement - for example if you use numeric sequencing with three extra states Signal will run through states 1, 2 and 3 but will not use state zero until the sequencing has finished.

### State labels

Where Signal needs to refer to a state, for example in a button in the states control bar, it generates a simple descriptive name; either 'Basic 0' for state zero or 'State n' for the other states. This reflects the special nature of state zero as described above. It is often useful to be able to replace these labels with more descriptive ones such as '2 Pulses' or '10 ms int', this can be done for Dynamic outputs multiple states using the Label field in the Pulses dialog. For other forms of multiple states you have to temporarily select Dynamic outputs mode, use the Pulses dialog to edit the state labels, and then revert the multiple states mode to the form that is wanted.

### Auxiliary states hardware

In addition to controlling and modifying the 1401 outputs, Signal multiple states can be used with other 'auxiliary states' hardware (for example a Magstim transcranial magnetic stimulator) this hardware can be set up differently for each state. The Auxiliary >> button is used to select and configure the auxiliary states hardware that is use with this sampling configuration. The details of the various auxiliary states devices that are supported are documented separately under *Auxiliary states devices*.

### What else can multiple states do

In addition to Dynamic outputs there are two other more specialised forms of multiple states: Static outputs and External digital.

Static outputs mode replaces multiple sets of pulse outputs with multiple sets of unchanging outputs, one for each state. It is not much different from what one could achieve using Dynamic outputs without any output pulses defined so you could only set the initial levels of the outputs. The key extra feature of Static outputs states is that the outputs are set up before the sampling sweep begins, at the time the sweep is enabled (soon after the previous sweep finishes). So Static outputs can be useful if you want to feed controlling information to external equipment such as a stimulator to get it ready to deliver different stimulations – setting the outputs earlier allows the stimulator time to read the outputs and become ready before triggering the sampling sweep.

**External digital** mode is very different. In this style of operation the 1401 digital inputs are read by Signal at the end of each sampling sweep to retrieve data generated by external stimulation equipment such as a visual stimulator. This digital input data is used to generate the state code value for the frame just sampled. So this form of multiple states is suitable with external equipment which determines the stimulation to be used independently of Signal and outputs digital data to indicate what it has done.

### Enabling multiple states

The General page in the sampling configuration dialog contains a check box labelled Multiple frame states that enables multiple states in data acquisition. With this check box clear, sampling does not use multiple states and all sampled data frames are set to state zero, with it checked, sampling will use multiple states and set the data frames to the appropriate state value. The check box is not available with fast triggers, fast fixed interval or gap-free sweep modes as these do not allow for adjustment of the 1401 behaviour between sweeps.

When multiple frame states are enabled, the sampling configuration dialog gains another page labelled States which holds controls used to configure multiple states. All configuration of multiple states is carried out using this page in the sampling configuration, the actual outputs generated by the different dynamic output states are defined using the standard pulses configuration dialog..

## Dynamic outputs states

The State variation selector at the top left of the States page in the sampling configuration selects the type of multiple states to use from External digital, Static outputs and Dynamic outputs. The controls shown in the dialog change dramatically with the type of multiple states that is selected. Unless you are sure that you want to use another form of multiple states you should select Dynamic outputs.

With Dynamic outputs each state uses a different set of output pulses. The actual digital and DAC outputs for each state (along with some other information such as the state label) are set up using the Pulses configuration dialog available from the Outputs section of the sampling configuration. The states page is purely concerned with defining how many states there are and how they are sequenced – how and when Signal will switch from state to state during sampling.
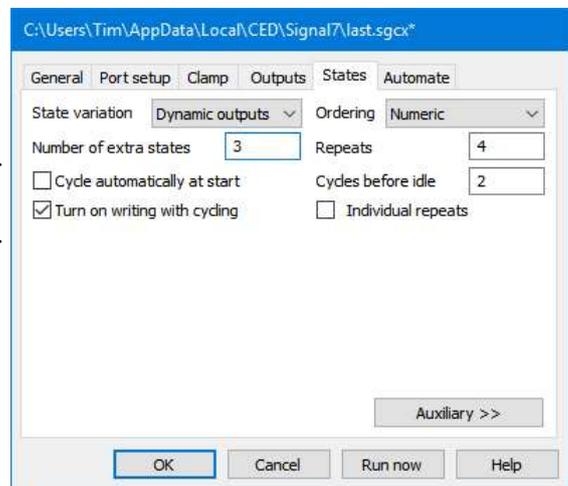


The Number of extra states item can be set to any value from 1 to 256, note that this sets the number of states <u>in addition</u> to state zero. Thus in the example shown there are 4 states possible, with codes running from 0 to 3. In many circumstances Signal will only make use of the extra states and reserves state zero as a background or idle state. This item also controls the states available in the pulses configuration dialog; you should set the number of extra states that you want here and set up the pulse outputs afterwards.

## States sequencing

Because Signal controls the states when using Dynamic outputs mode, we need to be able to specify which state is used at what point during sampling. The simplest way to do this is to control the state manually using the states control bar (see here), but we will often want some form of automatic sequencing – it's easier, faster and less error-prone. Signal provides two basic forms of states sequencing – numeric or protocol. The four numeric modes provide simple numeric or randomised states sequencing, while using protocol mode allows more complex sequences of states to be generated.
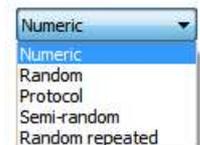
# Numeric non protocol sequencing modes

In the numeric states sequencing modes the user specifies how many of each state are to be used, how many times they are to be used overall, how and whether the ordering is to be randomised, and what happens when state sequencing begins. All of these options are controlled using other items on the states configuration page, the items and their operation are the same for all state sequencing modes apart from Protocol mode.

*Repeats*
This item sets how many of each state are to be used to make up one cycle of the states sequence. It can be set to any number from 1 to 1000. For example with 3 extra states and 4 Repeats, each state from 1 to 3 will be used 4 times in one cycle of the states sequencing. It is possible to set a different repeat count for each state by using individual repeats.

*Cycles before idle*
This item sets how many cycles (a cycle being states*repeats as described above) of sequencing are wanted before the sequencing stops and Signal switches automatically to idling in state 0. Set this item to the number of cycles you want or to 0 if you want states sequencing to repeat forever until stopped manually.

*Cycle automatically at start*
This check box, if checked, causes states sequencing to begin automatically when sampling starts (normally sampling starts with Signal idling in state zero, though writing to disk can be turned on).

*Turn on writing with cycling*
This check box, if checked, causes writing to disk to be enabled whenever states sequencing is started – this can be quite useful as otherwise it is hard to do this neatly and all too easy to forget to do at all and often we do not want to write data collected while idling.
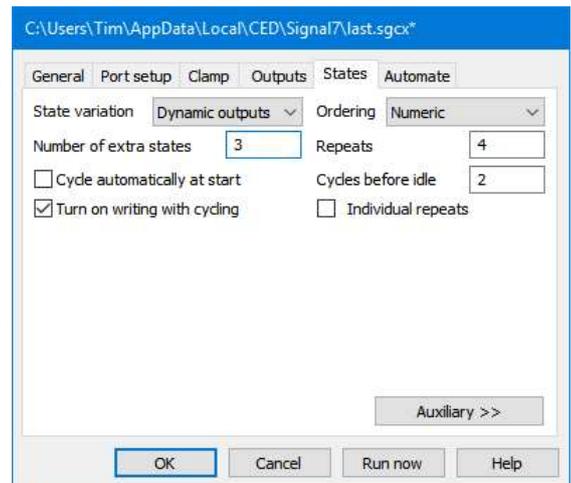
**Ordering**
This selector sets the states sequencing mode to be used, it can be set to any of:

Numeric
In this mode the states are used in numerical order with each state being used the number of times that is set by Repeats. First state 1 is used the specified number of times, then state 2 and so forth. Once the last state has been done, one sequencing cycle has been completed. For the example with 3 states and 4 repeats, Numeric sequencing would give:

| 1 | 1 | 1 | 1 | 2 | 2 | 2 | 2 | 3 | 3 | 3 | 3 |

in one cycle of sequencing.

Random
In this mode, one cycle of the sequencing again uses each state the number of times specified by Repeats, but the order of the states within a cycle is randomised. So, again for the example, Random sequencing might give:

| 2 | 3 | 2 | 1 | 3 | 3 | 1 | 2 | 1 | 3 | 1 | 2 |

in one cycle of sequencing (but of course what you actually get will vary). When the sequencing starts another cycle, the states order is re-randomised.

Semi-random
This is an alternative method of randomisation where the states are not all randomised across a cycle but instead randomised within one set of states. For the example settings the first 3 frames will always include one of each state (in random order), as will the next 3 and so forth, but one cycle of sequencing still consists of (states * repeats) frames. So you might get:

| 2 | 1 | 3 | 3 | 1 | 2 | 3 | 2 | 1 | 1 | 2 | 3 |

in a sequencing cycle for the example. As you will have realised, this mode achieves exactly the same as setting the number of repeats to 1 in Random mode, but a sequencing cycle still consists of each state being used the number of times specified by Repeats.

Protocol This is a non-numeric mode and is described separately. When Protocol mode is selected, all of the check boxes and controls for repeats and cycles are hidden and replaced by a Protocols… button.

Random repeated This is an alternative method of randomisation where the order of the states is randomised, but each state repeats the number of times that is set by Repeats. For the example settings the first 4 frames will always be the same state, as will the next 4, but the state numbers for each set of four will be random. So you might get:

2 2 2 2 3 3 3 3 1 1 1 1

in a sequencing cycle for the example shown.

# Individual repeats

The simple descriptions of state sequencing all assume that each state is used Repeats times in one sequencing cycle. The Individual repeats check box allows you to set a different repeat count for each state. If it is clear, the Repeats item is used to set how many times each state is repeated. If the check box is set, different controls are used to set the repeats wanted for each state separately.

When Individual repeats are selected, the overall Repeats item for all states is hidden and the dialog instead shows a state selector and repeat count so that you can set a repeat count for each state. Individual repeat counts are used in much the same way as the overall repeat count. In Numeric ordering each state is repeated in turn the specified number of times, while with Randomised ordering each state is repeated the set number of times but the order within each sequencing cycle is randomised. Semi-random ordering does not work very well with individual repeats enabled as towards the end of a cycle the states with a lower repeat count will be left out, but maybe this behaviour will be useful in some circumstances. With Random repeated ordering the order of the states is randomised as usual, but the number of repeats varies as set for the state in question.
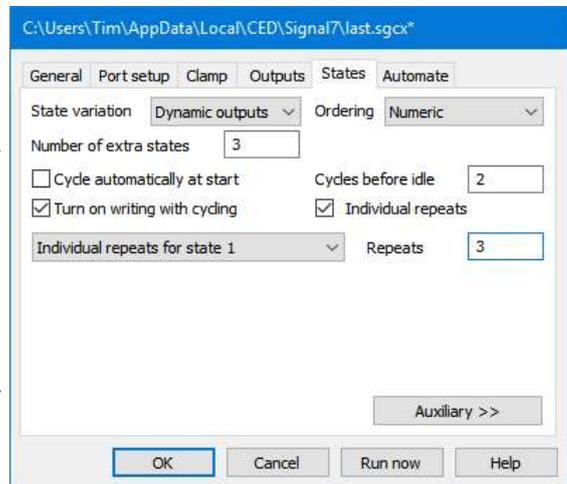


# State sequencing using protocols

When Protocol states ordering is selected all of the states page controls apart from the number of extra states are hidden and a Protocol… button is provided that allows you to create, view and edit the protocols.

A protocol consists of a list of steps; each step defines a state that will be used, the number of times it will be used and the step to go to next. There are also controls defining what happens when a protocol starts, and when it finishes. Each protocol can use up to ten steps, protocols can loop and be chained together to produce longer sequences of states. Up to 50 protocols can be defined in a single sampling configuration.

Protocols are defined using the protocols dialog which is provided when you press the Protocol… button on the states page. The dialog has a selector at the top that is used to select a protocol for viewing and editing. To create a new protocol press the Add Protocol button, while the currently selected protocol can be deleted using the Delete

button. The protocol name can be changed by directly editing it in the protocol selector. Blank protocol names are not allowed and will be rejected.

## General and start-of-protocol options

The check boxes below the Add Protocol and Delete buttons set general options and define what happens at the start of protocol execution – some of them perform similar functions to check boxes available for numeric sequencing modes:

### *Create toolbar button for protocol*

This item provides a separate button for this protocol in the states control bar when it is checked – see the section *Controlling multiple states online*.

### *Run protocol automatically at start*

This item does what it says; when checked it sets this protocol to be run when sampling starts. A protocol that is to be run automatically at the start is indicated by having a '*' character appended to the protocol name.

### *Cycle protocol state only after write*

This controls the sequencing of protocol steps; if it is checked then the protocol sequence does not advance unless data file frames are written to disk, if the data is not written or a sweep is rejected then the same state is repeated.

### *Use per-step write flags*

This item enables control of writing to disk on a step-by-step basis if checked. The actual control of writing is then done using the check boxes by the right-hand edge of the protocol table section of the dialog.

### *Turn on writing at protocol start*

This item causes writing of sampled sweeps to disk to be turned on when the protocol starts execution if it is checked.

### *Reset pulse steps at protocol start*

This item, if checked, causes any varying pulses that are defined in the output pulses to be reset to their initial state when the protocol starts.

## Table of protocol states

The table below the check boxes defines the actual protocol steps. There are ten steps in a protocol, each one with a State, a Repeats count and a Next step. These specify the state to be used, the number of times to repeat this state and the step to go to (from 1 to 10) when this step is done. If you set the next step to zero then this step is the end of the protocol. If per-step write flags option has been selected the Write check boxes are enabled and will control the turning on and off of writing data to disk at the beginning of each step of the protocol.

When protocol execution begins it starts with step 1. The state set by the State field in step 1 is set and is used the number of times set in the Repeats field. Following this the protocol switches to the step that is set by the Next field. This process continues until it is ended by encountering a Next item of zero. In the example shown, step 1 repeats state 1 four times and then goes to step 2. This repeats state 3 eight times, after which the protocol ends. If the Next item for step 2 were set to 1, the protocol would loop forever or it could be set to 3 for a more complex sequence.

## End-of-protocol options

The controls at the bottom of the dialog control what happens when the sequence of states defined in the protocol table reaches its end:

### *Repeats for entire protocol*

This item, immediately below the protocol table, controls the number of times that the protocol repeats before it ends. Set this to 1 for a protocol that goes through the steps only once (as in the example above) or set it to a larger number for a protocol that repeats a set number of times before ending. For example, if you set the repeat count to

three in the example above, the protocol would run for 36 frames before ending. If you set this item to 0 the protocol will repeat forever until stopped manually.

### At end

This item controls what happens once the specified number of protocol repeats has been executed. It can either be set to Finish or to a protocol that will be 'chained-to', chaining to a protocol allows more complex sequences than is possible with just ten steps. If the selector is set to Finish protocol execution finishes, otherwise execution switches to the protocol selected and carries on. When a protocol is chained-to, it starts off completely normally, so the Turn on writing and Reset pulse steps check boxes take effect. These check boxes do not take effect if the last step in the protocol has a Next item of step 1 or when an entire protocol repeats, so it is meaningful to allow a protocol to chain to itself.

The check boxes below control what happens when a protocol actually finishes; they are disabled if the protocol chains to another protocol.

### State zero when protocol finishes

This item causes automatic switching to state zero when the protocol finishes if checked, if it is clear then the last state used by the protocol will continue to be used.

### Turn off writing when protocol finishes

This item controls disabling of writing sampled data to disc, if checked then automatic writing of data to disk will be turned off when the protocol finishes.

If both of these check boxes are checked they give the standard idling behaviour when the protocol finishes. When the test protocol shown above is executed writing to disk would first be turned on, then the following sequence of states would be generated:

```
1  1  1  1  3  3  3  3  3  3  3  3
```

and finally Signal would idle in state zero, with writing to disk turned off.

# Controlling multiple states online

When sampling using multiple states Signal provides a states control bar to allow you to control the sampling state and states sequencing. The control bar can be docked at the edges of the Signal application or be left floating anywhere within the application. The states control bar will be initially visible when sampling starts, it can be hidden or shown by using the sample menu or a popup menu that is displayed if you right-click on an unused part of the Signal application window.

The layout of controls in the states control bar is depends upon whether a numeric or protocol sequencing style is being used; numeric modes have extra buttons for controlling states sequencing whilst protocol mode has extra items to select and run a protocol and optional buttons to run individual protocols.

# Non protocol ordering

The states control bar contains a number of buttons and controls:



| Reset | Press this button to reset any states sequencing in operation (so that the state sequence restarts at the beginning) and also to reset any varying pulses to their initial state. |
|---|---|
| Idle | Press this button to force states sequencing to idle. It switches to manual control of states, sets state zero and turns off writing to disk. |
| Pause | Press this button to pause any automatic state sequencing that is in progress. This option does not pause the sampling itself which continues to run using the current state (use the sampling control panel if you want to pause the sampling). While states sequencing is paused this button is shown depressed; press it again to resume sequencing. |

Manual Press this button to terminate any automatic sequencing in progress and to switch to manual control of states. With manual control, the frame state is controlled interactively by the user, sampling begins with manual control selected unless the Cycle automatically at start option is used.

On write Press this button for automatic states sequencing with the states changing only if the previous data was written to disk. The states sequencer will have control of the frame states and will move on to the next stage after a sampling sweep only if the sweep data was saved to disk. This allows for artefact rejection and interactive sweep accept/rejection without missing out states from the sequence.

Cycle Press this button to start automatic states sequencing with the states always changing. The state sequencer will have control of the frame states and the sequencer will always move on to the next stage after a sampling sweep regardless of whether the sweep data is written to disk or not.

Basic 0 … Press these buttons when in manual mode to switch to a state. Buttons for unused states are hidden, as are buttons for states numbers greater than 9. If you have created labels for the states in the pulses configuration dialog these labels will be used in the buttons instead of the standard state names. During automatic states sequencing the state buttons are disabled; Signal depresses them automatically to show the state currently in use.

State n To the right of the individual state buttons is a state selector and spinner that can be used when in manual mode to choose any state from those available. This selector is most useful when there are more than 9 states, the limit for individual state buttons. As for the buttons, if you have set a label for a state this is shown instead of the simple state name. Selecting a state uses it immediately. During automatic states sequencing the current state is shown.

# Protocol ordering

When multiple states are used with protocol ordering some controls in the states control bar are hidden and others are displayed instead. Those that are retained behave in the same way as described above (the On write and Cycle buttons are hidden because the individual protocols select between these two styles of operation). If the number of states is less than ten, so that they can all be represented by the individual state buttons, the main state selector is also hidden to save space. The extra items provided are the protocol selector, the run protocol button and the individual protocol buttons:



Protocol To the right of the state buttons (or the state selector if it is visible) is a protocol selector and spinner that can be used to select a protocol from those available. Unlike the state selector, selecting a protocol does not execute the protocol. During execution of a protocol the current protocol in use is shown so you can see what is going on if you use chained protocols. If all of the protocols have individual buttons then the protocol selector is not shown.

Run This button is shown below the protocol selector. Pressing it causes Signal to start executing the selected protocol. If all of the protocols have individual buttons then this is also not shown.

Buttons These buttons are created for the individual protocols as required, they are arranged to the right of (or below) the protocol selector and are labelled with the protocol name. A protocol button will be created if the Create toolbar button for protocol check box in the protocol definition is set and there are not too many protocol buttons – the limit is 20. Pressing one of these buttons causes the relevant protocol to be executed immediately.

## Starting a protocol

Execution of a protocol can be begun by the user pressing the Run button with that protocol selected (or the button for an individual protocol), or by another protocol chaining to it, or by means of the script language.

## Stopping a protocol

Execution of a protocol is stopped when the protocol finishes, by the user pressing the Idle or Manual buttons in the states control bar or by another protocol being started by whatever means.

# Static outputs states

Static outputs states are very similar to Dynamic outputs mode, the difference is that instead of having different sets of output pulses it generates unchanging outputs that are present throughout the sampling sweep. When Signal is getting ready for a sampling sweep, it writes values to the 1401 DACs and digital outputs using data for the current state of the experiment. These outputs could be used to select the stimulus to be generated by external equipment or have other effects.

The States box to the bottom left of the dialog is used to select a state and to define the values to be written to the digital outputs for each sweep. The digital output bits available are set in the Outputs page of the sampling configuration, set a check box to set that output high for the relevant state or leave it clear for a low output. The actual digital outputs and connector pins used are the same as for pulse outputs (see *Pulse outputs during sampling*).
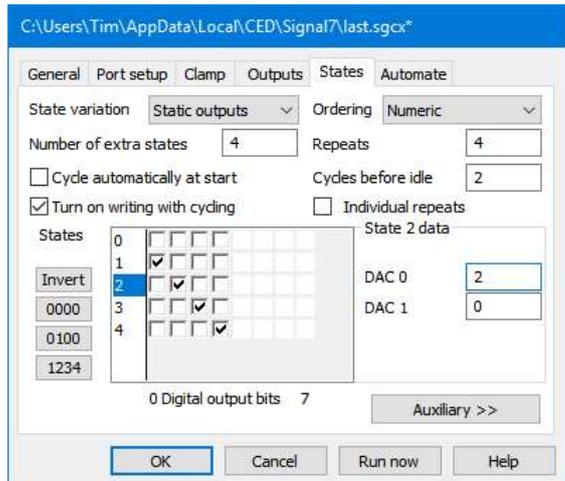
The buttons to the left of the States box set or modify the digital bit patterns in various useful ways, they are intended to allow simple patterns to be set up quickly and to help to produce more complex ones:

Invert    This inverts all of the bits for all states. This is useful for converting all zeroes to all ones and a walking one into a walking zero.

0000    This sets all of the bits for all states to zero.

0100    This sets most bits to zero with a walking 1. This leaves state zero all clear, sets bit 0 for state 1, bit 1 for state 2 and so forth. The pattern is not adjusted to skip disabled digital outputs.

1234    This sets the bit values to count the states using binary code. Thus state 1 has just bit 0 set, state 2 has bit 1 only and state 3 has both bits 0 and 1. Once again the pattern is not adjusted to skip disabled digital outputs.

The State data box to the right of the States box shows and sets the DAC outputs for the state that is selected in the States box - you can select a state by clicking on the digital outputs line for that state - in this picture state 2 is selected. The DAC outputs used (0 to 3 only are available) are enabled and disabled in the Outputs page of the configuration. If individual repeats are enabled then the repeat count for a state is set here too.

Static output states can only be used with the outputs type set to None in the Outputs page, as otherwise the state values would conflict with the other outputs that are generated. When sampling using Static output states, controls for the state and states sequencing are provided in exactly the same manner as for Dynamic outputs.

# External digital states

External digital states are very different from the other forms of multiple states sampling. When using this mode, external equipment generates up to 8 bits of digital code corresponding to the current experiment state. Signal reads this code from the 1401 digital inputs at the end of each sampling sweep and uses this data to set the state for each sampled data frame. There is no internal control over the states (as it is set by the external equipment) so all of the state sequencing controls are hidden.

The States box on the bottom left of the dialog is used to define the input bit patterns that correspond to individual states. Input bits 0 to 7 are shown, with 0 on the left, the Digital inputs enable check boxes below control which inputs are used; disabled inputs are ignored. An unchecked bit corresponds to a zero bit (low or 0 volts) while a checked bit selects a one bit (high or 5 volts). The buttons to the left of the States box modify the bits in various useful ways as documented for Static outputs mode. The bits for state zero are disabled as they are ignored, don't forget that any unconnected digital inputs will read as ones, not zeroes!
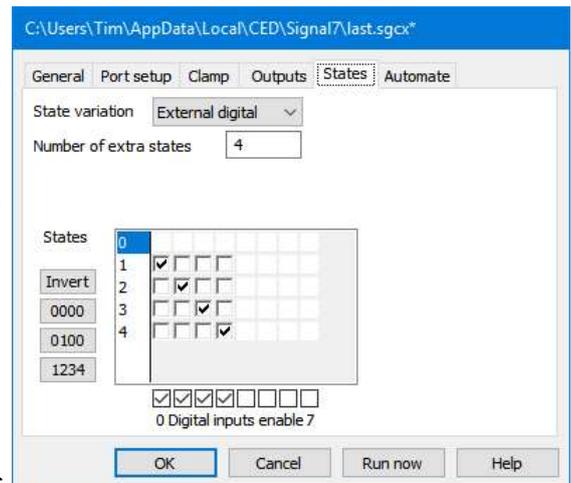
During sampling Signal will read the digital inputs at the end of each sampling sweep. The inputs are read as soon as possible after Signal recognises that the sampling sweep has ended, but for complete reliability you should aim to ensure that the inputs remain valid for at least 25 milliseconds after the sweep end. The bits that are read are then checked against the bits defined for each of sampling state starting with state 1, and the state for the new frame is set to the first one that matches. If no match is found then the frame state is set to zero.

External digital states uses digital input bits 8 to 15; see *Sampling data* for details of the digital input connections. On modern 1401s, the lowest two of these digital inputs (bits 8 and 9) are not connected to the digital input connector on the rear of the 1401; instead they are taken from the Event input BNC connectors on the front of the 1401. Please also note that if a digital input is not connected it will read as high (reads as a 1); it is thus important to disable any unconnected inputs or set up the states bits so that a high input works.

When sampling using External digital states, Signal behaves much as it does with multiple states disabled, and no states control bar is shown. The only difference is that the state code value for sampled data frames varies according to the digital inputs.

# Auxiliary states hardware

In addition to the standard multiple states facilities that control or use 1401 outputs Signal can make use of one or more auxiliary states devices. These are external devices (most often a type of stimulator) that are set up in a different way for each state. Signal already supports a number of different auxiliary states devices, if you need support for other equipment, please contact CED.

The Auxiliary >> button in the States page is used to select which auxiliary devices are to be used during sampling and to configure them. When you click on the Auxiliary >> button a menu is generated by Signal, this starts with Add Device and if any devices are already in use this is followed by Remove Device and below that the devices currently in use are shown in a list as Device name - Label.
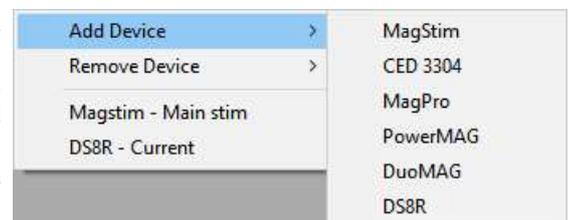
### Add Device

This command is used to add a new device to be used. If you move the mouse pointer over this menu item a sub-menu is generated holding a list of the available auxiliary states device types. Select the device of your choice and it will be added to the list of devices in use and the configuration dialog for that device displayed so that you can set a helpful device label and configure the device as required.

### Remove Device

This command is used to remove a device from those in use. If you move the mouse pointer over this menu item a sub-menu is generated holding a list of the devices in use ( shown as Device name - Label to avoid confusion). Select a device and it will be removed from the list of those devices in use.

### Individual devices

These menu entries, showing both the device name and label, are used to edit the configuration for a device in use. Select a device and its configuration dialog will be displayed so that you can view and edit it.

### Viewing and editing auxiliary states device settings during sampling

While sampling is in progress, you can open the settings dialog for any auxiliary states device that is in use using the Auxiliary states devices item in the main Signal Sampling menu. In



addition, you can optionally display the auxiliary states control bar which contains a button which will give you quick access to the device settings. When these settings dialogs are used online the overall device settings cannot be altered, but the settings for the individual states can be freely adjusted. For some devices, the auxiliary states control bar includes a control which can be used to quickly edit a device parameter. For TMS devices, the parameter provided for quick access is the resting motor threshold (RMT) value.

The details of the various supported auxiliary states devices are documented separately under *Auxiliary states devices*, including the details of their configuration dialogs.

In addition to manual configuration, the `SampleAuxStateNew()` and `SampleAuxStateRemove()` script language functions can be used to control the list of devices and the `SampleAuxStateParX()`, `SampleAuxStatePar$()` and `SampleAuxStateValX()` functions can be used to read and write auxiliary device settings.

# Sampling with clamp support

Signal incorporates a number of specialised features directly supporting whole-cell and single-channel clamping experiments - online holding potential control, resistance measurements and membrane analyses, dynamic clamping, leak subtraction and single-channel analysis. In order to avoid confusing users who are not interested in this type of experiment, all of these features can be hidden using a check box in the Edit menu Preferences dialog (Clamp page), the initial state of this option is set up by Signal when starting if it detects that it has not already been set. The specific features controlled by this option are the online clamping support, leak subtraction analysis and the generation and analysis of idealised single-channel traces.

Unlike many applications used for clamping experiments, Signal is a very general-purpose program that can be used for many other types of experiment. This can make it harder for new users setting up clamping experiments to find the aspects of Signal that they need. In addition to the clamp-specific features mentioned above (and general familiarity with Signal), clamping researchers should start by reading the Getting started with clamping experiments discussion or, if they are using one of the specialised amplifiers supported as auxiliary telegraph devices, the similar discussion here.

For proper familiarity, researchers should look through all of the sampling configuration dialog, at the amplifier telegraph system (*Amplifier telegraphs*), the pulses configuration dialog (*Pulse outputs while sampling*) and dynamic outputs multiple states (*Sampling with multiple states*). In addition to clamping-specific analysis mechanisms, the general *Analysis menu* documentation should be useful as all of these analyses can be used on clamp data. The trend plot, measurement generation and curve fitting sections of the analysis menu may be of particular interest. Specialised data acquisition and analysis can be carried out using the Signal script language which has access to all aspects of Signal data.

## Online clamp support features

The online clamp support within Signal consists of a specialised page within the sampling configuration which holds clamp set information (input channels and control DACs that are linked to a clamping amplifier) mechanisms for online analysis and experiment manipulation that make use of this information to provide automatic resistance measurements, holding-potential (or current) controls and a membrane analysis display plus generation of dynamic clamping currents. Note that the dynamic clamping support is independent of the clamp sets - you can use either of these facilities without the other as you prefer, or both together.

There are a number of limitations to the types of experiments that can be used with the clamping support which are described under Other sampling configuration considerations.

### Getting started with clamping experiments

Signal is a much more general-purpose program that many of the other applications used for voltage- and current-clamping experiments. While this does have advantages - for example Signal can carry out an extremely wide range of analyses - it does make getting started with these types of experiment a bit harder. If you are using one of the specialised amplifiers supported as auxiliary telegraph devices such as the MultiClamp 700 you should refer to the similar discussion here which covers both special aspects of using these amplifiers and the general aspects discussed here.

### Defining your clamping sets

You should start defining a sampling configuration for clamping experiments by determining what signals you are going to be sampling and how the amplifier that you are going to used is controlled. For example, in voltage-clamp experiments the membrane potential imposed by the amplifier will be controlled by a signal generated by one of the 1401 DACs. You will then be able to see how you connect the outputs from the amplifier to 1401 ADC inputs and connect a DAC to the amplifier external control input. The Ports page of the sampling configuration dialog can then be used to set the calibration and units for the ports connected to the amplifier outputs, and the Outputs page used to set the calibration and units for the external control DAC. It is very important that you get these calibrations correct otherwise your sampled data will not be correctly shown and the membrane analysis and holding potential controls will not operate correctly - see the user manual for your amplifier for details on this.

Once you have done that, you can define your clamping sets. These are defined in detail here here, simply put a clamping set is a pair of channels of data relating to the same cell/electrode, one showing the membrane current

and one the membrane potential, plus a DAC which controls whichever one of these is being clamped. Signal uses the clamping set information for all aspects of running a clamping experiment.

If there is anything about the clamping set information that you have entered, for example bad ADC port units, it will show that information in red in the clamping set information. You will need to correct this before using the sampling configuration.

### Getting your sampling configurations correct

Once the ADC and DAC calibrations and units are correct, and the clamping set information is entered and satisfactory, you are getting there! Other bits of the Signal sampling configuration information will need to be set up according to what you want to do. You will need to read the relevant pages of this help file, the Sampling data, Sampling with clamp support, Pulse outputs while sampling and Sampling with multiple states chapters will probably all be very useful. At the very least you will need to:

- Use the General page to define the sampling rate that you want to use, the ADC ports you want to sample-from and the duration of each frame of data.

- Use the Outputs page to select pulse outputs. Sequencer-controlled outputs can be used in clamping experiments but hopefully you will be able to start of with the much simpler pulses mode.

- Use (very probably) the General page to enable multiple states sampling and set the multiple states mode to Dynamic outputs.

- Generate sets of suitable stimuli using the DACs connected to the amplifier external control inputs. You do this using the pulses configuration dialog available from the Outputs page of the sampling configuration. Multiple states sampling in dynamic outputs mode is very useful here as it allows you to have up to 256 separate sets of output stimuli in one sampling configuration and switch between them in various ways during the experiment.

### Using voltage clamp together with current clamp

It is often the case that users want to use both voltage clamping and current clamping on the same cell. While Signal can easily do this, it cannot do this within the same data file, the amplifier settings for the two modes are too different. So to switch from one mode to another it is necessary to stop sampling and save the new data file, change the amplifier mode and sampling configuration, and start sampling again to a different data file. Note that the Automation section of the sampling configuration provides automatic file name generation and data file saving facilities that help to make this quick.

It is necessary here to have separate sampling configurations for VC and IC experiments, as the ADC port and DAC settings needed will be very different. So to run a dual-mode experiment you would probably do something like this:

- Set the amp into VC mode
- Load in the VC sampling configuration file
- Run it and collect data
- Stop sampling and save your data
- Set the amp in IC mode
- Load in the IC sampling configuration file
- Run it and collect data

Remember that any changes you make to the amplifier gains or external control sensitivities must be accounted-for by altering the relevant ADC port or DAC calibrations, so your signals and stimuli are correctly calibrated. It is probably a good idea to set up the amplifier with suitable gains and sensitivities, get the calibrations correct for these, and then never change the amplifier settings.

Note also that there is a sample tool-bar in Signal which you can set up with individual buttons for your key sampling configurations. Using the sample bar allows loading in a new sampling configuration file to be a one-click operation, very useful when you are busy trying to collect data quickly.

### Issues with 1401 signal limits

As normally shipped, the 1401 data acquisition interface ADCs and DACs only cover a ± 5 volt range. If you increase the amplifier gain too much, you may well find that they exceed this limit and are clipped, similarly with some external command sensitivities you may find that the 5 volt DAC output range does not allow the stimuli you

need. In some circumstances and with some amplifiers these problems can be avoided by using low amplifier gains and high external command sensitivities, but it is normally simplest, with modern 1401s, to change the supported voltage range to ±10 volts. This can be done using the Try1401 program that is installed with Signal, use the File menu 1401 Options command.

# Clamping configuration

The clamp page in the sampling configuration is used to configure Signal online clamping support. This page allows up to eight clamping sets to be defined along with a state (set of pulse outputs) to be used for membrane analysis and also provides a button used to configure dynamic clamping and another button used to retrieve information from MC700 (or other) amplifiers.

### What is a clamping set?
A clamping set is a pair of interlinked data file channels that hold the current and voltage data for a cell (more specifically across a cell membrane) plus a control DAC that is used to control the membrane potential (for voltage clamp experiments) or current (for current clamp). It is necessary for the clamping sets to be defined in the sampling configuration so that the stimulus and response channels are known and the membrane analysis can find the correct data, this also allows Signal to check the units for these channels so that current and voltage values can be scaled correctly to amps and volts. Defining the DAC used to control the membrane current or voltage allows Signal to manipulate the holding potential and stimulus pulses and again to check that the DAC units are correct for the clamping mode.

When using the auxiliary telegraph support for the MultiClamp 700 amplifier or other supported amplifiers, Signal is able to automatically set up the clamping sets using information read back from the amplifiers. This ensures that all of the clamping information plus the calibrations of relevant ports are always correct - a great boon for the hurried researcher. A short 'getting started' guide to using Signal with the MultiClamp 700 support is available here, nearly all of the information given is also applicable to use with other amplifiers.

### Defining the clamping configuration
The upper part of the clamp page shows a list of all eight clamp sets and allows you to click on a set to select it while the lower area allows you to view and edit the parameters of the selected set.

The Clamp type for a set can be Not clamping to disable use of that clamp set or set to Whole-cell or Single channel for either voltage clamp or current clamp. If all of the sets are Not clamping then the online clamping support will be disabled for this sampling configuration. When the clamping information is saved it is sorted so that all of the clamp sets that are in use are first in the list and all of the unused sets are placed at the end, but you can edit any of the sets that you want.

The Stimulus channel and Response channel items set the data file channels that will hold the corresponding data. For voltage-clamp experiments the stimulus channel is voltage and the response channel is current, for current-clamp it is the other way around. These items can be set to any valid channel number from 1 to 80. If the channel specified does not exist in the sampling configuration then the text for the incorrect item will be shown in red. If the channel does exist then the channel calibration will be checked for valid units. The units for a voltage channel must contain the character V and start with either V or one of M, U, N, P or F (lower-case characters are also allowed) implying milli-, micro-, nano-, pico- and femto-volts respectively. Plausible legal voltage units would thus include 'V', 'mVolts' or 'uV'. The initial character is used to scale the measured values to actual volts during online analysis. Similarly the units for a current channel must contain the character A and start with either A or one of M, U, N, P or F, 'nA' and 'pAmp' are obvious examples that would be acceptable. If you hover the mouse pointer over a prompt that is red then a tooltip will be generated that gives more information about what is wrong with your selection.

The Control DAC item sets the DAC number that is used to control the stimulus, 0 to 7. This DAC must be in use in the pulse outputs and the units set for the DAC must be appropriate for voltage or current (according to the clamp type) or the DAC prompt text will be shown in red.
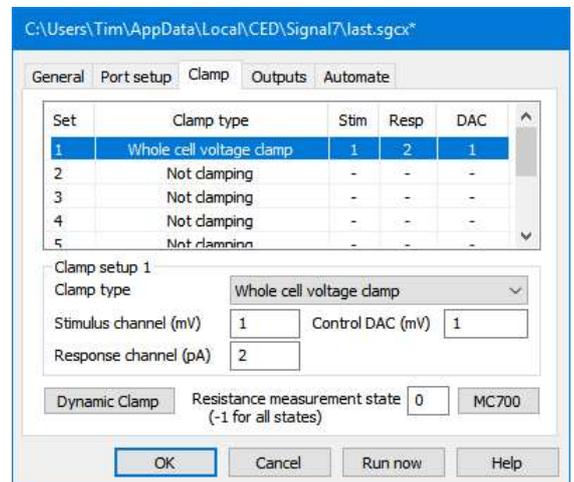
The Resistance measurement state item specifies a state (a set of pulse outputs) to be used for measurement of resistance and other membrane analyses. This value is not checked within the dialog, but it must either be -1 or zero if multiple states are not being used or from -1 to the maximum state number in use if you are using multiple states. Whenever this state is used during sampling Signal will automatically recalculate and display the membrane resistance, if this value is set to -1 then the membrane analysis is always carried out, regardless of the state. In addition, this state will be automatically selected (if it is not -1) when the membrane analysis dialog is used. So, with state 4 set as the state for resistance, whenever state 4 is used during sampling the membrane resistance will be recalculated, displayed and saved in the data file. Signal states are quite a complex topic; for more information on them, see under *Sampling with multiple states*. The clamp support expects that you will be using Dynamic outputs multiple states so you can ignore the extra complexity of the other multiple states modes.

A suitable stimulus pulse for resistance measurements should be defined for the control DAC in the specified state (or all states), this pulse can be added to the pulse outputs or modified during sampling should this prove necessary. If there is more than one pulse for this state the pulse to be used for measurements can be labelled by setting the pulse ID to 'RM', otherwise the first suitable pulse found in the outputs will be used. Currently, a square pulse (constant or varying amplitude) or a square pulse train can be analysed. Varying duration square pulses, sine waves, ramps or arbitrary waveforms are not currently supported.

### Dynamic clamping

The button labelled Dynamic Clamp at the bottom of the clamp configuration page is used to gain access to the main dynamic clamping setup dialog. Dynamic clamping is a very complex topic which is described below, note that there is no requirement to have clamp sets defined (which provide information for holding potential control and membrane analysis) in order to use dynamic clamping – the two systems are independent sections of Signal.

### Multiclamp 700, AxoClamp 900A and EPC 800 integration

If you have installed support for MultiClamp 700, AxoClamp 900A or EPC 800 amplifier telegraphs, a button will be available at the bottom of the clamp page that provides options to read the current amplifier settings and use these to set up clamping information. This option can obtain the ADC and DAC ports set within MC700/900A/EPC800 telegraph configuration dialogs and use them plus the amplifier settings to set the clamping sets and channels used thereof, the clamping modes and the relevant ADC and DAC port calibrations - the only things you have to set yourself are the MC700 or 900A or EPC800 settings available from the ports configuration page - so it should be much easier to get everything set up correctly. These options are only available when the amplifier control software is also running, as Signal needs to be able to read back the amplifier settings from this software.

The options available from the MC700/AC900/EPC800 button are `Read channel sets for clamping`, which initialises the clamping sets with stimulus and response channels plus control DACS which were derived from the current amplifier settings, `Read clamping modes for sets` which retrieves the clamping mode used in the amplifier for each clamping set, `Read channel names, scaling and units` which retrieves all calibration information for all waveform channels generated from the relevant ADC ports (as defined in the MC700, AC900 or EPC800 telegraph settings) and `Read DAC scaling and units` which sets up the DAC specified for each clamp set using the external command sensitivity read back from the amplifier. An extra option, `Read All – chans, mode, ADC & DAC`, reads all of this information at once.

These options match the reading & use of amplifier settings which may be automatically carried out when sampling starts depending upon various amplifier configuration options. As well as allowing quick & reliably correct setup of Signal, they can be used to check what the current amplifier settings are and, because the DAC scalings can thus be quickly set up, makes it easier to set up the output pulses that you want to use.

See the settings dialog information for the relevant amplifier for details of how a clamping set is recognised using the amplifier settings.

# Other sampling configuration considerations

As mentioned in the introduction, because Signal is a very general-purpose program it is possible to produce a sampling configuration that does not allow the clamping features to operate correctly, though we have tried hard to make the software as flexible as possible. A checklist of the aspects of the sampling configuration that we believe you need to ensure are correct is:

- Any sweep mode can be used but as the two fast sweep modes and gap-free mode do not allow multiple states or changes to the pulse outputs they are very limiting and will not work at well. Gap-free sampling can be used to

avoid lost data, particularly for single channel experiments, but again only one fixed set of pulse outputs are available and the clamping control bar and membrane analysis are not available. Currently Signal does not take account of gap free sampling in single channel analysis but we hope to allow this in the future.

- External convert sampling cannot be used as this prevents Signal from determining the time-course of events.

- The stimulus and response channels specified in clamping sets must exist (be generated by the configuration) and the ports used to log the channels must use appropriate units as described in the clamping configuration details. The channel calibration scaling factors must be correctly set up for the amplifier gains, and any telegraph settings must be correct, or data analysis will give incorrect results.

- Pulses outputs must be used with absolute levels turned off if you want the holding potential controls to work as expected. Sequencer output can be used for specialised experiments but it will not work with the membrane analysis or holding potential controls.

- All the DACs specified for stimulus control must be enabled in the outputs and calibrated using suitable units. The DAC scaling factors must be correctly set for the amplifier control inputs or the stimuli generated will be incorrect.

- If multiple states are used (which is almost always necessary), these should use dynamic outputs variation.

- The state specified for resistance measurements must exist. State zero always exists, even if multiple states are not used.

Signal checks the sampling configuration for incompatibilities with the requirements of clamping at the start of sampling and will generate an error message if problems are detected.

# Running a clamping experiment

When a sampling configuration with at least one enabled clamping setup is used, the clamping control bar is created in addition to the other Signal toolbars that may be available during sampling. The clamping bar contains a separate set of controls for each clamping setup in use and can either be floating or docked to any edge of the Signal window. For each clamping set in use, the bar displays the same items; a selector for the resistance values to be displayed, items for control of the membrane holding potential, a text display area and a button marked Memb used to provide the membrane analysis dialog.
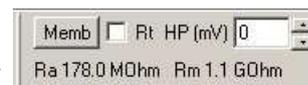
The Rt check box is used to toggle between showing the total electrode resistance only (useful when attaching to a cell) or access and membrane resistance (more useful once the cell is attached and a seal established). If the check box is set then only the total resistance will be displayed.

The holding potential (or holding current, for current clamp) controls are used to adjust the baseline level of the stimulus DAC (as set in the sampling configuration) used by the clamping set. The initial value of the holding potential is taken from the initial (baseline) level of the appropriate DAC in state zero. Whenever the holding potential is changed, the baseline level for the DAC is set to the new value in all states, the holding potential is also initialised for all states at the start of sampling. Note that, when editing the holding potential directly, you have to accept the new value by pressing enter, tab or otherwise moving away from the edit field for the change to take effect.

You can step the holding potential up and down by clicking on the spinner to the right of the edit field, by using the keyboard up- and down-arrow keys when the text cursor is in the edit field or by rotating the mouse wheel when the text cursor is in the edit field. All of these changes take place immediately. For all of these methods, the step size is 1 millivolt normally, 5 millivolts if the Shift key is held down and 20 millivolts if the Ctrl key is down (0.5, 2.5 and 10 pA for current clamp).

At the bottom of the toolbar area for a clamping set is a text field where the total resistance or electrode access and membrane resistances are displayed, according to the current state of the Rt check box. This field is automatically updated whenever the state specified for resistance measurements in the sampling configuration is used, you can select the state manually or it can be used as part of normal states cycling or protocol execution. Resistance measurements are carried out regardless of whether sampled data is being written to disk.

All of the controls in the clamping toolbar are disabled when the sampling mode is fast triggers, fast fixed interval or gap-free as these modes do not allow the necessary interaction with sampling.

# Membrane analysis

The Membrane button in the clamping control bar is used to provide the membrane analysis dialog; this dialog uses the state for resistance measurements specified in the sampling configuration to carry out a more complete analysis of membrane properties. When it is used, sampling is automatically switched to the state specified for resistance measurements and writing to disk is turned off. The membrane analysis dialog can also be opened using the `SampleMembAna()` script function.



The membrane analysis measures and displays the total resistance, access and membrane resistance, the time constant for the decay of the capacitance transient and the calculated membrane capacitance (see here for details of the analysis).

At the top of the dialog is the waveform display area. This shows a slice of waveform from the response channel at the time of the stimulus pulse. If the analysis has been successful the fitted curve corresponding to the capacitance decay will be drawn on top of the waveform data. The two buttons labelled >< and <> (below the waveform display area) can be used to increase and decrease the time range shown.
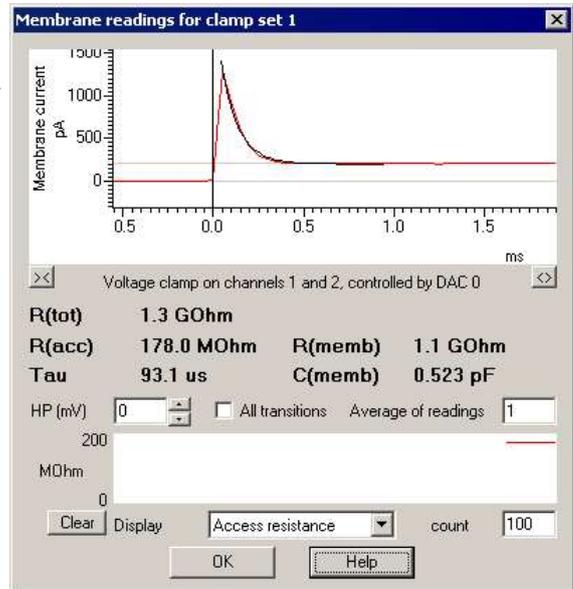
Immediately below the waveform display area is an area used to show the results of the most recent analysis. These results are updated after every frame, the results can optionally be either the average of all stimulus transitions or the result of just one transition and can also be averaged over a number of sweeps if desired.

The holding potential control can be used to adjust the membrane holding potential (or current) in exactly the same way as the equivalent control in the clamping control toolbar.

The All transitions check box, when checked, causes the response to all of the edges of the selected stimulus pulse to be analysed rather than just the first edge (the one shown in the waveform display). If more than one edge is analysed, the results shown in the text area are based upon the averaged results for all edges. This control also affects the results shown in the graphical display of a measurement over time.

In addition to averaging across all stimulus transitions, you can average the analysis results that are displayed over a number of sweeps by using the Average of readings control. This can be set to any value from 1 to 1000. Note that if you set a large value here there will be a significant delay before any changes become clear.

The area below these controls provides a display of a selected measurement over time. The Display selector selects the measurement to be displayed from those available, while the `count` control sets the number of measurements shown over the width of the display area, from 10 to 1000. The values displayed in this graph are updated with data from every sweep (the Average of readings control has no effect, but the All transitions check box operates). The Clear button deletes all saved measurements and restarts the display – this can be particularly useful if an aberrant value causes the automatic Y scaling to show too large a data range.

When the membrane analysis dialog is closed the sampling state sequencing system and writing to disk are both restored to their previous state at the time that the dialog was displayed.

# Changing pulses

The pulses dialog can be used to edit the output pulses used (including the pulse for resistance measurements) at any point while the experiment is in progress. Any changes made to the pulses while the membrane analysis dialog is in use will not cause the membrane analysis to fail in a destructive fashion, but deleting the pulse for resistance measurements or changing it to a type that is not usable for resistance measurements may prevent the membrane analysis from being carried out.

# Resistance data

The total, access and membrane resistance values, along with the membrane capacitance, are stored in the new data file as frame variables. The names of the variables are 'RTotn', 'RAccn', 'RMembn' and 'CMembn' respectively, where n is 1 for the first clamp set, 2 for the second and so forth. The values stored will be zero for data where the relevant clamp set is not being used, 1.0 when a resistance measurement has not yet been taken or the relevant value (in MOhms or pF units) generated by the last membrane analysis.

These values can be viewed in the File information dialog, retrieved using the script language or used as measurement values in trend plot generation.

# Analysis methods

Resistance and other membrane measurements are generated by analysis of a transition; a step change in stimulation (clamped voltage or current) level with the new level being maintained for a reasonable period. For a simple square pulse there are two transitions; one at the start of the pulse and one at the end, for a pulse train there are a sequence of transitions. Analysis is carried out on the first suitable pulse found in the outputs for the specified state; you can force a particular pulse to be used by setting the pulse ID to 'RM'.

To measure total resistance, the post-transition level is measured during the final quarter of the stimulus pulse (for transitions at the end of a pulse the previous pulse width is assumed). The delay of ¾ of the pulse width is intended to allow the capacitive transient to settle and can be adjusted by altering the stimulus pulse width. The baseline (pre-transition) level is measured over the same amount of time just before the transition. Measurements are taken from both the stimulus and response channels allowing the resistance to be easily calculated by using Ohms law.

After the stable levels before and after the transition have been measured, voltage clamp data can be further analysed to generate membrane capacitance, access resistance and membrane resistance measurements. The decay phase of the overshoot is analysed by fitting a double exponential curve to the current signal. The data used for the fitting is half the total stimulus pulse length in duration, starting at the point of maximum overshoot. If the double-exponential fit is successful the shorter time constant found (corresponding to the neuron body) is used, otherwise the time constant from a single-exponential fit is used. Having got a time constant for the decay, the area underneath the overshoot is measured to give the total charge. This analysis is more complex than the simple calculation of the total resistance and can fail for a number of reasons - the possible causes of failure are too few points for the curve fitting (either due to sampling too slowly or using too short a pulse for the measurements), insufficient overshoot in the pulse response to carry out the measurement or a failure in the exponential curve fitting (probably due to excessively noisy data or not enough data points for a good fit).

Having measured the total charge and the amplitude of the voltage step, the membrane capacitance can be calculated as charge/delta volts. We are then able to estimate the access resistance by Rs = time constant/capacitance, which allows us to separate out the membrane and access components of the total resistance.

For current clamp experiments, only the overall resistance is measured.

When all transitions are being analysed, the results of analysis of each transition are averaged to give a single set of measurements for the sweep.

# Dynamic clamping

Dynamic clamp is a specialised experimental technique in which current is injected into a cell in order to simulate, or cancel out, a natural transmembrane conductance of some sort. This is done by repeatedly sampling the voltage differential across membranes, calculating (using equations appropriate to the conductance being modelled, which can be relatively simple or extremely complex) the current that will be generated and updating a DAC output that drives a current clamp amplifier to inject the current. To be useful for research, this process has to be done quickly relative to the speed of the biological processes being modelled, so the mechanisms used to evaluate the equations need to be fast. The main areas of research that dynamic clamping is used for are:

- Isolating the ionic conductance of a particular ion type in order to study its specific effects on the cell.
- Connecting two cells in a brain slice with an artificial synapse in order to study a cell's response to synaptic input or generate hybrid networks. The pre-synaptic cell can also be replaced with an artificially generated action potential.

Signal includes a high-performance, easily configurable, dynamic clamping option which is fully integrated into its standard sampling mechanisms. It makes use of the high performance available from the latest 1401 design to

evaluate the dynamic clamping model equations using highly optimised mechanisms, the equations are evaluated at the same rate as the ADC inputs are sampled.

Using the 1401 allows dynamic clamping to be tightly coupled to the ADC sampling, ensures that the timing is fast & unaffected by the non real-time aspects of the Windows operating system and ensures that, if the equation evaluation and DAC updating are slow relative to the ADC sampling, this can be reliably detected.

### Hardware requirements

Because of the high processing speeds required, a Power1401 mk II, Power1401-3 or Micro1401-4 data acquisition system is required in order to use dynamic clamping. We have also found that a high-performance clamp amplifier is required for effective dynamic clamping. The MultiClamp 700A and 700B amplifiers from Molecular Devices (previously Axon), the 2400 from A-M systems, the BA-03X and many other amplifiers (BA-01X, BA-01M, ELC-01X, ELC01M, ELC03XS, SEC03M, SEC05X, SEC01X) from NPI Electronics and the EPC 800 from HEKA are all known to work well with current clamping and are therefore suitable for dynamic clamping, but of course this is not to say that other amplifiers are unsuitable. The 700A and 700B amplifiers from Molecular Devices, the 2400 from A-M systems, the EPC 800 from Heka and the BA-03X from NPI have been tested at CED and seen to work correctly with Signal dynamic clamping.

### Models, ADCs and DACs

The basic organisational element within dynamic clamping is the model. A model can define something as simple as an ohmic leak conductance or as complex as a population of Hodgkin-Huxley ion channels or a synapse. The basic job of a model is to describe how the dynamic clamp current that is to be injected will be calculated. Models use one or two sampled waveform channels as inputs, have a set of equations that control how the inputs control the conductance plus defined parameter values such as the maximum conductance that are used within equation evaluation and drive one or two DAC outputs with the output of the equations to control the current injected by the amplifier. If the sampling configuration is using multiple states sampling the model information is extended to provide multiple sets of model parameters which can differ between states, but the basic model settings such as the model type, input channels and DAC outputs are the same for all states.

Signal allows you to use up to 15 dynamic clamp models of any type simultaneously within one sampling configuration, models can share both waveform input channels and output DACs. When the models are evaluated all outputs that drive the same DAC are summed and the final DAC output is the sum of all the dynamic clamp outputs and any output sent to that DAC by the standard Signal pulse or sequencer output systems.

#### *Waveform (ADC) inputs*

The starting point for any dynamic clamp calculations are measurements of membrane potentials – the potential within the cell measured relative to the surrounding medium. Signal dynamic clamping uses measurements taken from sampled data file channels to find the membrane potential, this means that the channels in question must be created by sampling from an ADC port as set in the sampling configuration and that the data must be suitably calibrated in millivolts. Other calibrations would be possible – all that is necessary is that Signal 'understands' the units so it can derive correct potentials – but measurements in millivolts appear to be always used by researchers. It is also important that the channel calibration factors – the conversion factors between volts at the 1401 input and millivolts measured by the clamp amplifier electrode - set in the ports configuration are correctly set to match the amplifier behaviour so that the membrane potential values seen by Signal are correct.

#### *DAC outputs*

It is equally important that the DACs used to control current injection by the clamping amplifier are correctly set up so that Signal can correctly control simulated currents. Because of this, any DACs used as control outputs by models must be calibrated in either picoamps or nanoamps, and the output calibration factors (the values used to define what current a given DAC output level causes) as set in the Outputs page of the sampling configuration is correctly set up to reflect the behaviour of the amplifier current control input, otherwise the currents generated by dynamic clamping will not be correct. When Pulse outputs are used the individual DACs can be separately calibrated, for Sequencer outputs one set of DAC calibration values are used for all DACs. Note that while sequencer outputs are compatible with dynamic clamping, the simple clamp support using defined clamping sets requires Pulse outputs.

*Digital inputs*

Some dynamic clamping models may be triggered by a 1401 digital input. When a model is configured to use digital triggers you can specify the digital input bit number to use from 0 to 7. These bits are in the upper 1401 digital input byte and so are actually bits 8-15, they are referred to as bits 0 to 7 for simplicity. See the documentation on digital inputs for details of the digital input connections.

**Starting & stopping**

Dynamic clamping uses the DACs on the 1401 to control the injection of current into a cell. Excessive current injection (sourcing or sinking) will quickly cause the cell membrane potential to exceed reasonable limits and probably kill the cell so mechanisms within Signal are used to try to avoid transients or other unwanted effects. When getting ready to sample, Signal makes sure that all of the DACs used to control dynamic clamping currents are set to zero and as the sampling starts any dynamic clamp models are preset using the current ADC input voltages to get the models ready for use and to preset the DACs to viable but stable values. As soon as sampling stops all of the DACs used to control dynamic clamping currents are set to zero once again. In addition, the models are zeroed and preset every time the parameters are changed to increase stability.

As Signal is an episodic data capture system, in all sampling modes apart from Gap-free there is an interval between sweeps during which sampling is not happening. This interval can be fairly small (probably <10 milliseconds for untriggered Basic mode, <100 microseconds for untriggered Fast triggers) but can also be very long, for example in Fixed interval or Extended modes. As the dynamic clamp equations are driven by ADC sampling this means that during this interval the dynamic clamp equations are not running and current injection is fixed at whatever level it was at at the end of the sweep. This can give problems if the delay is long so untriggered Basic or Fast triggers/Gap-free modes will work best. In addition a control is available to force the current injection to zero between sweeps, this is appropriate if the required resting current injection is close to zero and you wish to avoid the possibility of transient currents lingering.

**Acknowledgements**

The work done in adding dynamic clamp to Signal was inspired by the description of the `StdpC` program, described *in Journal of Neuroscience Methods 158 (2006) 287-299*. We are very grateful to Thomas Nowotny (one of the authors of this paper) for his advice and the time he spent explaining dynamic clamp to us.

In addition, we are grateful to Stephen Brickley, Stephanie Ratte, Carlos Paladini and Farzan Nadim and their co-workers and students for the work they put into testing the dynamic clamp support within Signal and for the many useful suggestions they made for improvements to the software.

# Dynamic clamp setup

Access to the Signal dynamic clamp system is provided through the Clamp section of the Sampling configuration dialog. If this section is not visible you will need to go to the Preferences dialog and select the Clamp page where you can then enable the clamping options provided by Signal.

The Clamp section of the Sampling configuration dialog is mostly occupied by up to eight clamping set definitions for voltage and current clamp or single channel patch clamp experiments. These are not part of the dynamic clamp system and can be used alongside dynamic clamping or not as you prefer. To set up dynamic clamping in a sampling configuration you should click on the Dynamic Clamp button provided at the bottom of the Clamp page. This provides the main Dynamic clamp setup dialog which is used to define the dynamic clamp models that you will be using.You can also use the Dynamic clamp setup dialog online to enable or disable models and to view, edit and update model parameters while sampling is in progress.
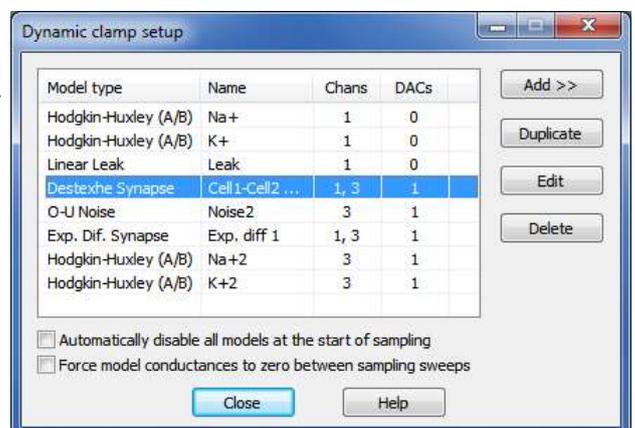


The Dynamic clamp setup dialog holds of a list of the models in use and buttons to add, edit or delete models. Models which have been disabled will be shown in grey, while modes with problems with their input channel or DAC numbers will have those numbers shown in red.

To add a new model, press the Add >> button which provides a menu allowing you to select the model you want. To create a duplicate of an existing model, press the Duplicate button. To edit an existing model, select the model by clicking on it in the list and then click on the Edit button or simply double click on the model in the list. To delete a model, select it and click the Delete button. It is not possible to add or delete models during sampling though you can disable them or change model parameters. The models available are:

- Hodgkin-Huxley (Alpha/Beta), which simulates a population of voltage-dependent ion channels using a formulation based upon multiple rate equations.

- Hodgkin-Huxley (Tau), which simulates a population of voltage-dependent ion-channels.

- Alpha synapse, which simulates a synapse between two cells using the Alpha function.

- CPG synapse, which simulates a central pattern generator synaptic connection between two cells.

- Destexhe synapse, which simulates a synapse between two cells using the Destexhe model.

- Electrical synapse, which simulates a simple electrical connection (a gap junction) between two cells.

- Exponential synapse, which simulates a synapse between two cells using a single exponential.

- Exponential difference synapse, which simulates a synapse between two cells using a difference between two exponentials.

- User defined synapse, which simulates a synapse between two cells using a model that replays a user defined waveform read from a text file.

- Leak models, which simulates various types of trans-membrane current leakage which do not have time-dependent behaviour.

- Noise, which simulates synaptic or leak noise using an Ornstein-Uhlenbeck process.

### Overall options
The Automatically disable all models at the start of sampling check box at the bottom of the dialog can be used to ensure that all models are initially disabled, should this be desired.

The Force model conductances to zero between sampling sweeps check box below can be used to ensure that all current control DACs are set to zero while Signal is between sampling sweeps, should this prove advisable (as discussed here). Generally, it is probably better if you leave this unchecked. This option does not have any effect when sampling using gap-free mode.

## Hodgkin-Huxley (Alpha/Beta) model

This model simulates the membrane current generated by a population of ion channels that obey the Hodgkin-Huxley equations using a formulation based upon multiple rate equations. The alpha/beta model is described using the equations:

$$I = G_{max}\, m^p\, h^q\, u^r\, (E_r - V)$$

$$\frac{dm}{dt} = \alpha_m (1 - m) - \beta_m m$$

$$\alpha_m = k_{\alpha,m} F_{\alpha,m} \left( \frac{V - V_{\alpha,m}}{S_{\alpha,m}} \right)$$

$$\beta_m = k_{\beta,m} F_{\beta,m} \left( \frac{V - V_{\beta,m}}{S_{\beta,m}} \right)$$

where $p$, $q$ and $r$ are integers (1 to 4), $E_r$ is the reversal potential of the current, $V$ is membrane potential, $m$, $h$ and $u$ are activation and inactivation variables and $G_{max}$ is the maximum conductance in nanosiemens. The calculation of $E_r - V$ is done this way around (rather than the more obvious $V$ - $E_r$) so that the system generates the correct polarity of output current. The first equation is, of course, the same as that in the Tau Hodgkin-Huxley model with a third component $u$ added. The equations deriving $h$ and $u$ are identical to those given above for $m$, the $\alpha_m$ and $\beta_m$ values (plus $\alpha_h$ and $\beta_h$, $\alpha_u$ and $\beta_u$) are functions of $V$ and represent changes in activation and inactivation. There are three choices for the activation and inactivation functions $F_{a,b}(x)$:

$$F_1(x) = \frac{x}{e^x - 1}, \quad F_2(x) = e^x, \quad F_3(x) = \frac{1}{1 + e^x}$$

in addition to these selections any of the four functions can be replaced by information read from a user-generated lookup table so that any alternative function you might require can be used. The values from these tables replaces the calculation of the internal rate variables $\alpha_m$, $\beta_m$, $\alpha_h$, $\beta_h$, $\alpha_u$ and $\beta_u$ as shown in the above equations with values read from a user-supplied table.

To implement this, we observe that for the activation variable m (the analysis for $h$ and $u$ is of course identical):

$$\delta m = \left(\alpha_m(1-m) - \beta_m m\right)\delta t = \alpha_m \delta t - m\left(\alpha_m + \beta_m\right)\delta t$$

So, given an $m$, we can calculate the change in $m$, and thus the value of $m$ for the next step. The remaining problem is to calculate the initial value of $m$. In the case that $\delta m$ is zero, we get:

$$m = \frac{\alpha_m \delta t}{\left(\alpha_m + \beta_m\right)\delta t}$$

Implementing this would require a long calculation, taking an appreciable amount of time. Because of this we simply set $m$, $h$ and $u$ to zero, as they will settle to their correct values after just a few sample intervals.

### Hodgkin-Huxley (Alpha/Beta) configuration

Adding or editing a Hodgkin-Huxley (Alpha/Beta) model brings up a dialog allowing the parameters of the model to be defined.

The **Model Name** field is used to enter a name for the model to help you remember which model is which, it can be up to 19 characters long. **Input Channel** is the waveform channel number (as displayed in the file view) for the membrane potential. It must have the units of **mV** in order to be recognised as a valid channel. The **Control DAC** field sets the DAC used to control injection of the current, the DAC selected must be calibrated in either **pA** or **nA** and have zero calibration offset – these values are set in the DAC scaling items in the Outputs page in the sampling configuration.

The $G_{max}$ field sets the maximum conductance of the ion channels being simulated in nanosiemens, while the $E_r$ field sets the reversal potential for the current. The polarity of the injected current can be changed by changing the sign of the conductance, a positive conductance will behave as though you were adding an ion channel population while a negative conductance acts in such a way as to effectively remove ion channels.

Below these parameters we have three sets of very similar parameters, one of which sets parameters that control the activation component of the Hodgkin-Huxley model, the second controls the inactivation component while the last controls component 3. It is possible to turn off the components independently using the Include check boxes; not including a component forces the corresponding *m*, *h* or *u* value to 1.

The rest of the parameters select the equations that are used for the alpha and beta rate variables of each component and set values that are used within these rate equations when they are evaluated. As an alternative to selecting an equation you can select User Defined, which replaces data generated by evaluating an equation with data read from a user generated table. See below for more details on the user defined function.

The Disable model check box at the bottom of the dialog disables and enables the entire model either across the board or for the currently selected state as appropriate. Disabling the model has the same effect as setting the $G_{max}$ conductance to 0.

### User-defined function

A user defined function replaces a selected function which is used to calculate one of the internal rate variables $\alpha_m$, $\beta_m$, $\alpha_h$, $\beta_h$, $\alpha_u$ or $\beta_u$ with a lookup table read from a text file on disk. The table defines how the relevant internal variable value depends upon the membrane potential. With a user-defined function the function parameters are replaced with a field holding a text file name. There are many ways that such a text file could be created; one of the easiest is to use a Signal script. The example GenAB.sgs script that is installed with Signal is provided as an example of how to do this.

The text file name can be a fully qualified file path and name such as "C:\Signal7\MyAB\XAB_am.txt" or it can be a simple file name such as "XAB_am.txt" in which case the directory where the sampling configuration file is stored is searched for the file. If a partial path and file name such as "MyAB\XAB_am.txt" is used then the directory path to the file is generated by starting with the sampling configuration directory and appending the text from the model file name. If no file extension is provided then .txt is appended. The table below illustrates this:

| File name in model | Configuration directory | Model data read from |
|---|---|---|
| C:\MyAB\XAB_am.txt | C:\Signal7 | C:\MyAB\XAB_am.txt |
| XAB_am.txt | C:\Signal7 | C:\Signal7\XAB_am.txt |
| MyAB\XAB_am.txt | C:\Signal7 | C:\Signal7\MyAB\XAB_am.txt |
| XAB_am | C:\Signal7 | C:\Signal7\XAB_am.txt |

Model functions that vary with the sweep state are generated by using separate file names in the model settings for each state. You can get Signal to automatically generate text file names that are derived from the sampling state by including the text "%ST%" in the text file name, this will be converted by Signal to the current sampling state as a two digit number (00 to 99) or three digits (100 to 256) as appropriate.

The data format in the text file is a pair of numbers on each line separated by spaces or a single tab character. Lines beginning with a single quote, " ' " are regarded as comments and will be ignored. The first number on the line is a membrane potential in millivolts as measured by the model input channel, the second is the rate variable value - a positive number from 0 upwards. The data in the text file should be sorted in order of membrane potential values with the lowest (most negative) value first and the highest (most positive) value last.

*Table usage and requirements*

When the text file data is used Signal reads all of the table values from the text file and converts them to a second lookup table which is 257 points long and exactly covers the ADC range. This conversion is done using cubic splining which will generally produce a very accurate result, but it does impose requirements upon the text file data. Firstly, the range of membrane potentials for which rate variable values are given should, in addition to covering the expected range of membrane potentials, exceed by a few points (5 at each end is ample) the entire range of potentials that could possibly be sampled from the input channel given the ADC port calibrations in use. This is necessary in order for the cubic splining to produce accurate results close to the table extremes. If the table values provided do not cover the entire ADC range then the generated table will be truncated at the first or last text file value as appropriate and a warning will be generated.

Secondly, the membrane potential values should be sufficiently close together to accurately represent the function behaviour. 100 values is probably the bare minimum that would be satisfactory and 250 to 500 is recommended. Evenly spaced potential values will probably be the easiest to produce but this is not a requirement - if your model displays quickly-changing behaviour at some potentials then you should design your text file to have more closely-

spaced values at these points while fewer points in areas with essentially linear behaviour will save space and time & not cause inaccuracy.

### Resource file read failure

At Signal version 7.03, we changed the library used to read and create XML files to use a faster, more stable library. This new library also adheres more closely to the XML standard, with a lot more checking generally. Sadly, this has exposed an XML design error (mine, to be precise) in the way that the information for this model was saved in Signal sampling configuration files, which means that Signal 7.03 and later versions will refuse to load sampling configuration files from Signal versions before 7.03 which contain this model (because the XML is seen as corrupt). However, it is possible to (carefully) edit these old sampling configuration files to correct them so that Signal can read them again. To do this:

1. Open the sampling configuration file using a standard text editor such as Notepad or Notepad++.

2. Find the line starting with <DynClamp (with space before) which marks the start of the dynamic clamping information and the line starting with </DynClamp which marks the end of this information.

3. Within the dynamic clamping information you will see the information for each model, which includes a comment giving the model name. The information for each model will vary, but mostly it will consists of name="value" pairs (including the double-quotes).

4. Within the information for each HH(A/B) model, you have to find all of the item names (the bit before the equals sign) which start with a '3' and change this '3' to a 'T'. This only applies to the item <u>names</u>, NOT the item values (the bit after the equals sign). So, for example, you would change "3Av=30" to "TAv=30". See below for a table showing all of the possible item names that may need to be changed.

5. Once you have successfully edited the configuration file, save it back to disk (I suggest that you use a different name in case of errors), and check that it can now be read by Signal. Note that this new file can be read without error by earlier versions of Signal, but that the model information for component 3 will not be present.

The possible item names you may find (not all of these will always be present) and what they should be changed-to are:

| Item name | New name |
|-----------|----------|
| 3AFunc | TAFunc |
| 3Ak | TAk |
| 3Av | TAv |
| 3As | TAs |
| 3AArbF | TAArbF |
| 3BFunc | TBFunc |
| 3Bk | TBk |
| 3Bv | TBv |
| 3Bs | TBs |
| 3BArbF | TBArbF |

While making these changes, do take great care to not change anything else, otherwise the sampling configuration file may be irreparably corrupted.

# Hodgkin-Huxley (Tau) model

This model simulates the membrane current generated by a population of ion channels that obey the Hodgkin-Huxley equations using a formulation based upon thresholds and threshold sensitivities. These are first-order non-linear differential equations:

$$I = G_{max}\, m^p\, h^q\, (E_r - V)$$

$$\tau_m(V)\frac{dm}{dt} = m_\infty(V) - m$$

$$\tau_h(V)\frac{dh}{dt} = h_\infty(V) - h$$

where $p$ and $q$ are integers (1 to 4), $E_r$ is the reversal potential of the current, $V$ is membrane potential, $m$ and $h$ are the activation and inactivation variables and $G_{max}$ is the maximum conductance in nanosiemens. The calculation of $E_r - V$ is done this way around (rather than the more obvious $V$ - $E_r$) so that the system generates the correct polarity of output current. The $\tau_m$, $\tau_h$, $m_\infty$ and $h_\infty$ values are all functions of $V$ and represent the changes in activation and inactivation:

$$m_\infty, h_\infty = \frac{1}{1 + e^{\left(\frac{V - Vth}{Vsl}\right)}}$$

$$\tau_m, \tau_h = \tau_0 - \frac{\tau_a}{1 + e^{\left(\frac{V - Vth\tau}{Vsl\tau}\right)}}$$

The values of $m$ and $h$ lie between 0 and 1 and represent fractions of the total conductivity. Given starting conditions, the equations can be solved dynamically:

$$\delta m = \frac{m_\infty(V) - m}{\tau_m(V)}\, \delta t$$

$$\delta h = \frac{h_\infty(V) - h}{\tau_h(V)}\, \delta t$$

That is, over a sample interval $\delta t$ given an initial value of $m$ and $h$, we can approximate the new value at the end, $m + \delta m$ and $h + \delta h$, by assuming $\delta m/\delta t = dm/dt$. The only problem left is to set an initial condition. To do this we assume that at the start of sampling the model is in a steady state (so dm/dt and dh/dt are zero), giving us:

$$m = m_\infty(V), \quad h = h_\infty(V)$$

## Hodgkin-Huxley (Tau) configuration

Adding or editing a  Hodgkin-Huxley (Tau) model brings up a dialog allowing the parameters of the model to be defined.

The Model Name field is used to enter a name for the model to help you remember which model is which, it can be up to 19 characters long. Input Channel is the waveform channel number (as displayed in the file view) for the membrane potential. It must have the units of mV in order to be recognised as a valid channel. The Control DAC field sets the DAC used to control injection of the current, the DAC selected must be calibrated in either pA or nA and have zero calibration offset – these values are set in the DAC scaling items in the Outputs page in the sampling configuration.

If either of the Input Channel or Control DAC fields are invalid then the text for the problematic item will be displayed in red and, after a pause, an error message describing the problem shown at the bottom of the dialog.

The rest of the dialog sets parameters for the model. See the equations above for precise details of how the various numeric parameters are used.

The $G_{max}$ field sets the maximum conductance of the ion channels being simulated in  nanosiemens while the $E_r$ field sets the reversal potential for the current. The polarity of the injected current can be changed by changing the sign of the conductance, a positive conductance will behave as though you were adding an ion channel population while a negative conductance acts in such a way as to effectively remove ion channels.

Below these we have two sets of very similar parameters, one of which sets parameters that control the activation component m, the other controlling the inactivation component h. You can turn individual components on and off using the Include check boxes, not including the activation or inactivation component effectively forces m or h respectively to 1. The rest of the parameters set values for the two components that are used within the equations. Most of these values can be freely set, regardless of their physiological validity, but the dialog does prevent you setting a $T_a$ value greater than the value of $T_0$ as doing this guarantees that the equations will 'blow up' and generate unusable results. As an alternative to the Standard function with parameters you can select User defined, which replaces data generated by evaluating the equations above with data read from a user generated table. See below for more details on the user defined functions.

The Disable model check box at the bottom of the dialog disables and enables the entire model either across the board or for the currently selected state as appropriate. Disabling the model has the same effect as setting the $G_{max}$ conductance to 0.

The OK button saves any changes that have been made and closes the dialog. The Apply button is only enabled during sampling and saves all changes while leaving the dialog open, while the Cancel button closes the dialog and discards unsaved changes as usual. Any changes made will take effect within a few milli-seconds, even if the current sweep has not finished.

## User-defined function

You can select using the standard functions (as describe above) or alternatively a user defined function for separate sections of the model evaluation. A user defined function replaces the standard function used to calculate one of the $\tau_m$, $\tau_h$, $m_\infty$ and $h_\infty$ values with a lookup table read from a text file on disk. The table defines how the relevant value depends upon the membrane potential. With a user-defined function selected the function parameters are replaced with a field holding a text file name. There are many ways that such a text file could be created; one of the easiest is to use a Signal script. The example GenTau.sgs script that is installed with Signal is provided as an example of how to do this.

The text file name can be a fully qualified file path and name such as "C:\Signal7\MyHH\XHH_minf.txt" or it can be a simple file name such as "XHH_minf.txt" in which case the directory where the sampling configuration file is stored is searched for the file. If a partial path and file name such as "MyHH\XHH_minf.txt" is used then the directory path to the file is generated by starting with the directory where the sampling configuration was stored

and appending the text from the model file name. If no file extension is provided then .txt is appended. The table below illustrates this:

| File name in model | Configuration directory | Model data read from |
|---|---|---|
| C:\MyHH\XHH_minf.txt | C:\Signal7 | C:\MyHH\XHH_minf.txt |
| XHH_minf.txt | C:\Signal7 | C:\Signal7\XHH_minf.txt |
| MyHH\XHH_minf.txt | C:\Signal7 | C:\Signal7\MyHH\XHH_minf.txt |
| XHH_minf | C:\Signal7 | C:\Signal7\XHH_minf.txt |

Model functions that vary with the sweep state are generated by using separate file names in the model settings for each state. You can get Signal to automatically generate text file names that are derived from the sampling state by including the text "%ST%" in the text file name, this will be converted by Signal to the current sampling state as a two digit number (00 to 99) or three digits (100 to 256) as appropriate.

The data format in the text file is a pair of numbers on each line separated by spaces or a single tab character. Lines beginning with a single quote, " ' " are regarded as comments and will be ignored. The first number on the line is the membrane potential in millivolts as measured by the model input channel, the second is the corresponding value - a positive number from 0 upwards. The data in the text file should be sorted in order of membrane potential values with the lowest (most negative) value first and the highest (most positive) value last.

*Table usage and requirements*

When the text file data is used Signal reads all of the table values from the text file and converts them to a second lookup table which is 257 points long and exactly covers the ADC range. This conversion is done using cubic splining which will generally produce a very accurate result, but it does impose requirements upon the text file data. Firstly, the range of membrane potentials for which rate variable values are given should, in addition to covering the expected range of membrane potentials, exceed by a few points (5 at each end is ample) the entire range of potentials that could possibly be sampled from the input channel given the ADC port calibrations in use. This is necessary in order for the cubic splining to produce accurate results close to the table extremes. If the table values provided do not cover the entire ADC range then the generated table will be truncated at the first or last text file value as appropriate and a warning will be generated.

Secondly, the membrane potential values should be sufficiently close together to accurately represent the function behaviour. 100 values is probably the bare minimum that would be satisfactory and 250 to 500 is recommended. Evenly spaced potential values will probably be the easiest to produce but this is not a requirement - if your model displays quickly-changing behaviour at some potentials then you should design your text file to have more closely-spaced values at these points while fewer points in areas with essentially linear behaviour will save space and time & not cause inaccuracy.

# Alpha synapse model

This model simulates the presence of a synapse between two cells using the Alpha function to generate the post-synaptic conductance profile. The synapse can be triggered by the presynaptic potential crossing a predefined threshold (in either direction), by a signal applied to a 1401 digital input port or by internal timed triggers. The equation for the post-synaptic conductance is given by:

$$g(t) = G_{max} * (t/\tau) * exp(1-(t/\tau))$$

where t is the time since the synapse was triggered. The equation generates a linear rising phase and an exponential decay falling phase using a single time constant $\tau$, with the maximum conductance occurring when t = $\tau$. The overall effect (while not especially physiologically realistic) provides a reasonable approximation for some synapses and has been widely used because of its low computational overhead. This model is generated using an internally-generated waveform table of up to 32000 points so the maximum duration of the synaptic current is 32000/ADC sample rate.

**Alpha synapse configuration**

Adding or editing an Alpha synapse model brings up a dialog allowing the parameters of the model to be defined.

The Model Name field is used to enter a name for the model to help you remember which model is which, it can be up to 19 characters long.

The Trigger section of the dialog defines how the synapse will be triggered. The Type item selects the type of trigger and can be set to one of +Analogue, -Analogue, TTL high, TTL low or Internal. +Analogue and -Analogue select triggering when the membrane potential on the specified pre-synaptic channel crosses a preset threshold (rising or falling respectively), with an optional Delay after the crossing before the trigger activates. For analogue triggers Presynaptic Channel is the waveform channel number (as displayed in the sampled data file) for the membrane potential (this must have units of mV in order to be recognised as a valid channel) and a separate field is used to enter the threshold potential. TTL high and TTL low select triggering when a specified 1401 digital input (upper byte) changes state (to high or low respectively), again with an optional Delay. Internal triggering generates one or more triggers at the specified times relative to the start of the sampling frame. The Sum triggered outputs check box at the bottom of the trigger details, if checked, allows the effect of up to 20 triggers to be summed to generate the overall model output - if this is cleared then each trigger detected terminates the effect of any previous trigger, resetting the conductance to zero and initiating another conductance pulse at the point when the trigger Delay expires.

The Postsynaptic Channel item is the waveform channel number used to measure the postsynaptic membrane potential so that the effect of the reversal potential can take effect, as usual this must channel have units of mV in order to be recognised as valid. The Postsynaptic Control DAC is the number of the DAC used to inject the current, this DAC must be calibrated in either pA or nA and have zero calibration offset – these values are set in the DAC scaling items in the Outputs page in the sampling configuration

The $G_{max}$ field sets the maximum postsynaptic conductance for the synapse, as used in the equation above, in nanosiemens, the polarity of the injected postsynaptic current can be changed by changing the sign of the conductance, a positive conductance will behave as though you were adding an ion channel population while a negative conductance acts in such a way as to effectively remove ion channels. The $\tau$ field sets the time constant for the model, in milliseconds.

The Receptor type field defines the postsynaptic conductance behaviour, it can be set to Linear (e.g. non-NMDA), GHK (e.g. GABA and Glycine), Boltzmann (e.g. NMDA) or User defined. The different types show different relationships between the postsynaptic conductance ($g(t)$ in the equation above), the postsynaptic membrane potential and the current generated by the model. The Linear receptor type shows a simple ohmic relationship relative to a reversal potential $E_r$ the GHK receptor generates current through the diffusion of an ion through a membrane via ion channels with the diffusion driven by the differing ionic concentrations on either side of the membrane and the voltage difference across it (there is no $E_r$, this being implicit in the differing ionic concentrations), Boltzmann type receptors resemble the linear type but with an additional factor that represents a voltage-dependent blockage of the current and the User defined receptor type uses a user generated lookup table to derive a scaling factor for $g(t)$ based upon the membrane potential at time $t$. See the description of the Leak model for complete details of the behaviour of the various type of receptor, in all cases the postsynaptic receptor behaves as a leak of the corresponding type whose conductance varies with time according to the above equation but note that the values in the User defined table act as a scaler of $g(t)$ rather than being a current in pA; the current is calculated by multiplying the scaled $g(t)$ by the postsynaptic membrane potential, the table data should be calculated in such a way as to take the required $Er$ into account.

# CPG synapse model

This model simulates the presence of a synapse between two cells with the pre-synaptic potential mediating the release of a synaptic transmitter which causes a post-synaptic current to flow, the current being injected into the post-synaptic cell. This particular model mimics the behaviour of synapses in the central pattern generation region of the lobster CNS. The equation for the current is given by:

$$I = G_{max} Sh (V_s - V_y)$$

$$1 - S_\infty(V_x) \tau_s \frac{dS}{dt} = S_\infty(V_x) - S$$

$$\tau_h(V_x) \frac{dh}{dt} = h_\infty(V_x) - h$$

The $h$ component is the short term depression and is mathematically identical to the standard Hodgkin-Huxley case. It is not normally used, in which case it has the constant value 1. $V_x$ is the presynaptic and $V_y$ is the post-synaptic cell membrane potential. If $V_x$ exceeds some threshold $V_{th}$:

$$S_\infty(V_x) = \tanh \left( \frac{V_x - V_{th}}{V_{sl}} \right)$$

Otherwise, $S_\infty$ is 0.0. The value of $S$ lies between 0 and 1. As in the Hodgkin-Huxley case, we can solve the equations using the Euler method:

$$\delta S = \frac{S_\infty(V_x) - S}{1 - S_\infty(V_x) \tau_s} \delta t$$

At the start of sampling we assume a steady state and $S$ is set to $S_\infty$

### CPG synapse configuration

Adding or editing a CPG synapse model brings up the dialog allowing the parameters of the model to be defined.

The Model Name field sets a convenient name for the model as usual. The Presynaptic Channel and Postsynaptic Channel items are the waveform channel numbers for the pre- and postsynaptic membrane potential data respectively, as usual these must have units of mV in order to be recognised as valid inputs of the membrane potential. The Postsynaptic Control DAC is the number of the DAC used to inject the current, which must be calibrated in either pA or nA and have zero calibration offset – these values are set in the DAC scaling items in the Outputs page in the sampling configuration. The other fields at the top and bottom of the dialog are identical to those in the Hodgkin-Huxley (Tau) model already described.

## Destexhe synapse model

This model simulates the presence of a synapse between two cells using the Destexhe model to generate the post-synaptic conductance profile. The synapse can be triggered by the presynaptic potential being above or below a predefined threshold (in either direction), by a signal applied to a 1401 digital input port or by internal timing. The triggering acts as a level so the trigger is either in an on or an off state rather than a trigger being detected at a given instant in time.

The Destexhe model uses two time constants, $T_{rise}$ and $T_{decay}$. When the trigger input is in the on state, the synaptic conductance tends (with an exponential decay process set by $T_{rise}$) towards the maximum synaptic conductance,

while when the trigger is off the conductance decays exponentially (with time constant $T_{decay}$) towards zero. The equation for the post-synaptic conductance during the trigger on state is given by:

$$g(t) = G_{max} + (G_{start} - G_{max}) \, exp(-t/\tau_{rise})$$

where $t$ is the time since the start of the on state and $G_{start}$ is the conductance at the start of the on state. The equation for the conductance in the trigger off state is:

$$g(t) = G_{start} \, exp(-t/\tau_{decay})$$

where t is the time since the start of the off state and $G_{start}$ is the conductance at the start of the off state.

**Destexhe synapse configuration**

Adding or editing a Destexhe synapse model brings up a dialog allowing the parameters of the model to be defined.

The Model Name field is used to enter a name for the model to help you remember which model is which, it can be up to 19 characters long.

The Trigger section of the dialog defines how the synapse will be triggered. The Type item selects the type of trigger and can be set to one of +Analogue, -Analogue, TTL high, TTL low or Internal. +Analogue and -Analogue select triggering when the membrane potential on the specified pre-synaptic channel is above and below a preset threshold respectively. For these triggers Presynaptic Channel is the waveform channel number (as displayed in the file view) for the membrane potential (this must have units of mV in order to be recognised as a valid channel) and a separate field is used to enter the threshold potential. TTL high and TTL low select triggering when a specified 1401 digital input (upper byte) is in a high or low state. Internal triggering generates one or more fixed-length triggers at the specified times relative to the start of the sampling frame.

The Postsynaptic Channel item is the waveform channel number used to measure the postsynaptic membrane potential so that the effect of the reversal potential can take effect, as usual this channel must have units of mV in order to be recognised as valid. The Postsynaptic Control DAC is the number of the DAC used to inject the current, this DAC must be calibrated in either pA or nA and have zero calibration offset – these values are set in the DAC scaling items in the Outputs page in the sampling configuration

The $G_{max}$ field sets the maximum postsynaptic conductance for the synapse, as used in the equation above, in nanosiemens, the polarity of the injected postsynaptic current can be changed by changing the sign of the conductance, a positive conductance will behave as though you were adding an ion channel population while a negative conductance acts in such a way as to effectively remove ion channels. The $\tau_{rise}$ field sets the time constant for the on trigger state, while $\tau_{decay}$ sets the time constant for the off trigger state, both values are in milliseconds.

The Receptor type field defines the postsynaptic conductance behaviour, it can be set to Linear (e.g. non-NMDA), GHK (e.g. GABA and Glycine), Boltzmann (e.g. NMDA) or User defined. The different types show different relationships between the postsynaptic conductance ($g(t)$ in the equation above), the postsynaptic membrane potential and the current generated by the model. The Linear receptor type shows a simple ohmic relationship relative to a reversal potential $E_r$, the GHK receptor generates current through the diffusion of an ion through a membrane via ion channels with the diffusion driven by the differing ionic concentrations on either side of the membrane and the voltage difference across it (there is no $E_r$, this being implicit in the differing ionic concentrations), Boltzmann type receptors resemble the linear type but with an additional factor that represents a voltage-dependent blockage of the current and the User defined receptor type uses a user generated lookup table to derive a scaling factor for $g(t)$ based upon the membrane potential at time $t$. See the description of the Leak model for complete details of the behaviour of the various type of receptor, in all cases the postsynaptic receptor behaves as a leak of the corresponding type whose conductance varies with time according to the above equations but note that the values in the User defined table act as a scaler of $g(t)$ rather than being a current in pA; the current is calculated by multiplying the scaled $g(t)$ by the postsynaptic membrane potential, the table data should be calculated in such a way as to take the required $Er$ into account.
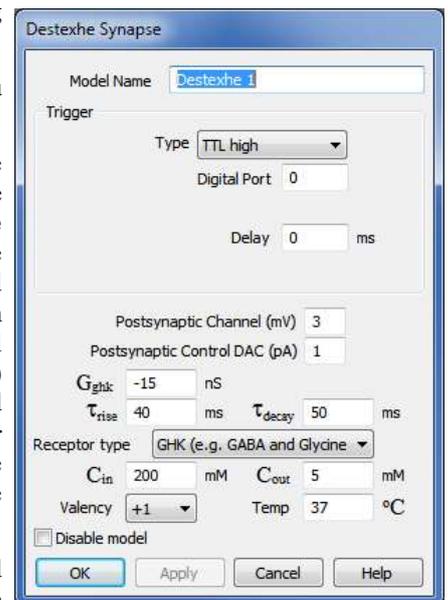
# Electrical synapse model

This is a simpler form of dynamic clamping which simulates two cells being electrically connected (a gap junction) to create a synapse. There are two output currents, these being:

$$I_y = -I_x = G_{es} (V_x - V_y)$$

Where $V_x$ is the presynaptic and $V_y$ is the postsynaptic cell membrane potential and $G_{es}$ is the conductance of the synapse in nanosiemens. Therefore this model simulates the existence of a resistor with a value $1/G_{es}$ ohms between the presynaptic and postsynaptic cells, with current flowing in one direction or another according to the voltage difference between the electrodes used to measure membrane potentials. The electrical synapse model can also be used in an alternative rectifying form where current only flows from the presynaptic to the postsynaptic cell and not in the other direction

### Electrical synapse configuration

Adding or editing an Electrical synapse model brings up the dialog allowing the parameters of the model to be defined.

The Model Name field sets a convenient name for the model as usual. The Presynaptic Channel and Postsynaptic Channel items are the waveform channel numbers for the pre- and postsynaptic membrane potential data respectively, as usual these must have units of mV in order to be recognised as valid inputs of the membrane potential. The Presynaptic Control DAC and Postsynaptic Control DAC items are the DACs used to inject the relevant currents and must be calibrated in units of either pA or nA and have zero calibration offset – these values are set in the DAC scaling items in the Outputs page in the sampling configuration.

You can select the rectifying form of the synapse by checking the No output if postsynaptic > presynaptic voltage check box. Note that this option selects between two fundamentally different styles of operation and, unlike the actual model parameters, can only be edited when state 0 is selected, cannot differ between states and cannot be changed online.

The parameters of the model are the synapse conductance $G_{es}$ in nanosiemens and the model enable flag controlled by the Disable model check box.
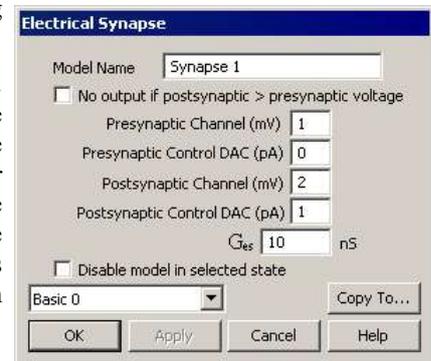
# Exponential synapse model

This model simulates the presence of a synapse between two cells using an instantaneous rise time and a single exponential for the post-trigger decay. The synapse can be triggered by the presynaptic potential crossing a predefined threshold (in either direction), by a signal applied to a 1401 digital input port or by internal timed triggers. The equation for the post-synaptic current is given by:

$$g(t) = G_{max} \, exp(-t/\tau_{decay})$$

where t is the time since the trigger. This model is generated using an internally-generated waveform table of up to 32000 points so the maximum duration of the synaptic current is 32000/ADC sample rate.

## Exponential synapse configuration

Adding or editing an Exponential synapse model brings up a dialog allowing the parameters of the model to be defined.

The Model Name field is used to enter a name for the model to help you remember which model is which, it can be up to 19 characters long.

The Trigger section of the dialog defines how the synapse will be triggered. The Type item selects the type of trigger and can be set to one of +Analogue, -Analogue, TTL high, TTL low or Internal. +Analogue and -Analogue select triggering when the membrane potential on the specified pre-synaptic channel crosses a preset threshold (rising or falling respectively), with an optional Delay after the crossing before the trigger activates. For analogue triggers Presynaptic Channel is the waveform channel number (as displayed in the sampled data file) for the membrane potential (this must have units of mV in order to be recognised as a valid channel) and a separate field is used to enter the threshold potential. TTL high and TTL low select triggering when a specified 1401 digital input (upper byte) changes state (to high or low respectively), again with an optional Delay. Internal triggering generates one or more triggers at the specified times relative to the start of the sampling frame. The Sum triggered outputs check box at the bottom of the trigger details, if checked, allows the effect of up to 20 triggers to be summed to generate the overall model output - if this is cleared then each trigger detected terminates the effect of any previous trigger, resetting the conductance to zero and initiating another conductance pulse at the point when the trigger Delay expires.

The Postsynaptic Channel item is the waveform channel number used to measure the postsynaptic membrane potential so that the effect of the reversal potential can take effect, as usual this must channel have units of mV in order to be recognised as valid. The Postsynaptic Control DAC is the number of the DAC used to inject the current, this DAC must be calibrated in either pA or nA and have zero calibration offset – these values are set in the DAC scaling items in the Outputs page in the sampling configuration

The $G_{max}$ field sets the maximum postsynaptic conductance for the synapse, as used in the equation above, in nanosiemens. The polarity of the injected postsynaptic current can be changed by changing the sign of the conductance, a positive conductance will behave as though you were adding an ion channel population while a negative conductance acts in such a way as to effectively remove ion channels. The $\tau_{decay}$ field sets the decay time constant for the model, in milliseconds.

The Receptor type field defines the postsynaptic conductance behaviour, it can be set to Linear (e.g. non-NMDA), GHK (e.g. GABA and Glycine), Boltzmann (e.g. NMDA) or User defined. The different types show different relationships between the postsynaptic conductance ($g(t)$ in the equation above), the postsynaptic membrane potential and the current generated by the model. The Linear receptor type shows a simple ohmic relationship relative to a reversal potential $E_r$, the GHK receptor generates current through the diffusion of an ion through a membrane via ion channels with the diffusion driven by the differing ionic concentrations on either side of the membrane and the voltage difference across it (there is no $E_r$, this being implicit in the differing ionic concentrations), Boltzmann type receptors resemble the linear type but with an additional factor that represents a voltage-dependent blockage of the current and the User defined receptor type uses a user generated lookup table to derive a scaling factor for $g(t)$ based upon the membrane potential at time $t$. See the description of the Leak model for complete details of the behaviour of the various type of receptor. In all cases the postsynaptic receptor behaves as a leak of the corresponding type whose conductance varies with time according to the above equation but note that the values in the User defined table act as a scaler of $g(t)$ rather than being a current in pA; the current is calculated by multiplying the scaled $g(t)$ by the postsynaptic membrane potential, the table data should be calculated in such a way as to take the required $Er$ into account.

# Exponential difference synapse model

This model simulates the presence of a synapse between two cells using the difference between two exponentials (one for the rising phase and one for the decay) to generate the post-synaptic conductance profile. The synapse can be triggered by the presynaptic potential crossing a predefined threshold (in either direction), by a signal applied to a 1401 digital input port or by internal timed triggers. The equation for the post-synaptic current is given by:

$$g(t) = G_{max} f(exp(-t/\tau_{decay}) - exp(-t/\tau_{rise}))$$

where $t$ is the time since the trigger and $f$ is a normalisation factor to ensure that the peak conductance equals $G_{max}$. The single exponential model is retrieved when $\tau_{rise}$ is set to zero, the alpha model is retrieved when $\tau_{rise}$ is set equal to $\tau_{decay}$. The actual effect of the equations is to make the rising phase be controlled by the smaller of the two time constants so if you set $\tau_{rise}$ to greater than $\tau_{decay}$ you will find that it acts as $\tau_{decay}$ and vice-versa. This model is generated using an internally-generated waveform table of up to 32000 points so the maximum duration of the synaptic current is 32000/ADC sample rate.

### Exponential difference synapse configuration

Adding or editing an Exponential difference synapse model brings up a dialog allowing the parameters of the model to be defined.

The Model Name field is used to enter a name for the model to help you remember which model is which, it can be up to 19 characters long.

The Trigger section of the dialog defines how the synapse will be triggered. The Type item selects the type of trigger and can be set to one of +Analogue, -Analogue, TTL high, TTL low or Internal. +Analogue and -Analogue select triggering when the membrane potential on the specified pre-synaptic channel crosses a preset threshold (rising or falling respectively), with an optional Delay after the crossing before the trigger activates. For analogue triggers Presynaptic Channel is the waveform channel number (as displayed in the sampled data file) for the membrane potential (this must have units of mV in order to be recognised as a valid channel) and a separate field is used to enter the threshold potential. TTL high and TTL low select triggering when a specified 1401 digital input (upper byte) changes state (to high or low respectively), again with an optional Delay. Internal triggering generates one or more triggers at the specified times relative to the start of the sampling frame. The Sum triggered outputs check box at the bottom of the trigger details, if checked, allows the effect of up to 20 triggers to be summed to generate the overall model output - if this is cleared then each trigger detected terminates the effect of any previous trigger, resetting the conductance to zero and initiating another conductance pulse at the point when the trigger Delay expires.

The Postsynaptic Channel item is the waveform channel number used to measure the postsynaptic membrane potential so that the effect of the reversal potential can take effect, as usual this must channel have units of mV in order to be recognised as valid. The Postsynaptic Control DAC is the number of the DAC used to inject the current, this DAC must be calibrated in either pA or nA and have zero calibration offset – these values are set in the DAC scaling items in the Outputs page in the sampling configuration

The $G_{max}$ field sets the maximum postsynaptic conductance for the synapse, as used in the equation above, in nanosiemens. The polarity of the injected postsynaptic current can be changed by changing the sign of the conductance, a positive conductance will behave as though you were adding an ion channel population while a negative conductance acts in such a way as to effectively remove ion channels. The $\tau_{rise}$ field sets the rise time constant and $\tau_{decay}$ sets the decay time constant for the model, both in milliseconds.

The Receptor type field defines the postsynaptic conductance behaviour, it can be set to Linear (e.g. non-NMDA), GHK (e.g. GABA and Glycine), Boltzmann (e.g. NMDA) or User defined. The different types show different relationships between the postsynaptic conductance ($g(t)$ in the equation above), the postsynaptic membrane potential and the current generated by the model. The Linear receptor type shows a simple ohmic

relationship relative to a reversal potential $E_r$ the GHK receptor generates current through the diffusion of an ion through a membrane via ion channels with the diffusion driven by the differing ionic concentrations on either side of the membrane and the voltage difference across it (there is no $E_r$, this being implicit in the differing ionic concentrations), Boltzmann type receptors resemble the linear type but with an additional factor that represents a voltage-dependent blockage of the current and the User defined receptor type uses a user generated lookup table to derive a scaling factor for *g(t)* based upon the membrane potential at time *t*. See the description of the Leak model for complete details of the behaviour of the various type of receptor. In all cases the postsynaptic receptor behaves as a leak of the corresponding type whose conductance varies with time according to the above equation but note that the values in the User defined table act as a scaler of *g(t)* rather than being a current in pA; the current is calculated by multiplying the scaled *g(t)* by the postsynaptic membrane potential, the table data should be calculated in such a way as to take the required *Er* into account.

# User defined synapse model

This model simulates the presence of a synapse between two cells using data from a text file to define the post-synaptic conductance profile. The text file data consists of a sequence of conductance values (which are scaled by an overall conductance) which are output at the same rate as the ADC sampling whenever the synapse is triggered. The synaptic output can be triggered by the presynaptic potential crossing a predefined threshold (in either direction), by a level change on a 1401 digital input or by internal timing. The model output can sum the effect of multiple triggers or subsequent triggers can replace any previously triggered output.

### User defined synapse configuration

Adding or editing a User defined synapse model brings up a dialog allowing the parameters of the model to be defined.

The Model Name field is used to enter a name for the model to help you remember which model is which, it can be up to 19 characters long.

The Trigger section of the dialog defines how the synapse will be triggered. The Type item selects the type of trigger and can be set to one of +Analogue, -Analogue, TTL high, TTL low or Internal. +Analogue and -Analogue select triggering when the membrane potential on the specified pre-synaptic channel crosses a preset threshold (rising or falling respectively), with an optional Delay after the crossing before the trigger activates. For analogue triggers Presynaptic Channel is the waveform channel number (as displayed in the sampled data file) for the membrane potential (this must have units of mV in order to be recognised as a valid channel) and a separate field is used to enter the threshold potential. TTL high and TTL low select triggering when a specified 1401 digital input (upper byte) changes state (to high or low respectively), again with an optional Delay. Internal triggering generates one or more triggers at the specified times relative to the start of the sampling frame. The Sum triggered outputs check box at the bottom of the trigger details, if checked, allows the effect of up to 20 triggers to be summed to generate the overall model output - if this is cleared then each trigger detected terminates the effect of any previous trigger, resetting the conductance to zero and initiating another conductance pulse at the point when the trigger Delay expires.

The Postsynaptic Channel item is the waveform channel number used to measure the postsynaptic membrane potential so that the effect of the reversal potential can take effect, as usual this must channel have units of mV in order to be recognised as valid. The Postsynaptic Control DAC is the number of the DAC used to inject the current, this DAC must be calibrated in either pA or nA and have zero calibration offset – these values are set in the DAC scaling items in the Outputs page in the sampling configuration

The $G_{scale}$ field sets the scale factor for the synapse postsynaptic conductances, the actual conductance at any time is the result of multiplying the relevant value from the text file by $G_{scale}$. The polarity of the injected postsynaptic current can be changed by changing the sign of the conductance, a positive conductance will behave as though you were adding an ion channel population while a negative conductance acts in such a way as to effectively remove ion channels.
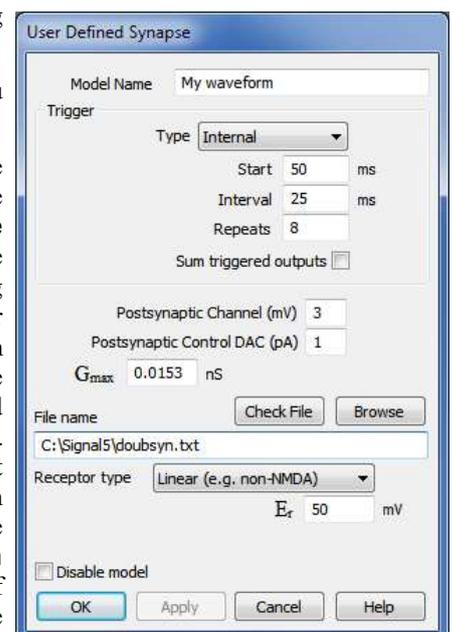
The Receptor type field defines the postsynaptic conductance behaviour, it can be set to Linear (e.g. non-NMDA), GHK (e.g. GABA and Glycine), Boltzmann (e.g. NMDA) or User defined. The different types show different relationships between the postsynaptic conductance as read from the user table, the postsynaptic membrane potential and the current generated by the model. The Linear receptor type shows a simple ohmic relationship relative to a reversal potential $E_r$ the GHK receptor generates current through the diffusion of an ion through a membrane via ion channels with the diffusion driven by the differing ionic concentrations on either side of the membrane and the voltage difference across it (there is no $E_r$, this being implicit in the differing ionic concentrations), Boltzmann type receptors resemble the linear type but with an additional factor that represents a voltage-dependent blockage of the current and the User defined receptor type uses a user generated lookup table to derive a scaling factor for the user-defined conductance based upon the membrane potential at the time. See the description of the Leak model for complete details of the behaviour of the various type of receptor. In all cases the postsynaptic receptor behaves as a leak of the corresponding type whose conductance varies with time according to the above equation but note that the values in the User defined table act as a scaler of *g(t)* rather than being a current in pA; the current is calculated by multiplying the scaled *g(t)* by the postsynaptic membrane potential, the table data should be calculated in such a way as to take the required *Er* into account.

*Synapse table details*

A user defined synapse uses a lookup table read from a text file on disk to define the synapse conductivity over time. This allows you to generate customised synapse conductivity profiles that follow any required behaviour. With a user-defined synapse the extra synapse parameters are replaced with a field holding a text file name. The text file name can be a fully qualified file path and name such as "C:\Signal7\MySynapses\XSyn.txt" that completely specifies the directory and file name or it can be a simple file name such as "XSyn.txt" in which case the directory where the sampling configuration file is stored is used. If a partial path and file name such as "MySynapses\XSyn.txt" is used then the path to the file is generated by starting with the sampling configuration directory and appending the text from the model. If no file extension is provided then .txt is appended. The table below illustrates this:

| File name in model | Configuration directory | Model data read from |
|---|---|---|
| C:MySynapses\XSyn.txt | C:\Signal7 | C:\MySynapses\XSyn.txt |
| XSyn.txt | C:\Signal7 | C:\Signal7\XSyn.txt |
| MySynapses\XSyn.txt | C:\Signal7 | C:\Signal7\MySynapses\XSyn.txt |
| XSyn | C:\Signal7 | C:\Signal7\XSyn.txt |

Synaptic conductivity profiles that vary with the sweep state are generated by using separate file names in the model settings for each state. You can get Signal to automatically generate text file names that are derived from the sampling state by including the text "%ST%" in the text file name, this will be converted by Signal to the current sampling state as a two digit number (00 to 99) or three digits (100 to 256) as appropriate.

The Check File button will check that the file name specified is correct, that the file exists, and contains properly formatted data. The Browse button allows you to browse for a text file to use. Note that the conductance file name can vary with the sampling state in multiple states sampling along with the usual trigger settings.

*Text file details*

The file should hold a sequence of numbers specifying the synaptic conductivity, one per line, up to 4,096,000 values can be provided. The values are converted into actual conductances by multiplying by $G_{scale}$, the actual use of this value is up to the user but two straightforward possibilities are to use actual conductances in the file and set $G_{scale}$ to 1.0, or the values in the file could run from 0 to 1.0 and $G_{scale}$ used to scale them to the actual conductance.

Blank lines in the file are ignored and lines starting with an apostrophe (') character are treated as comment lines.

# Leak model

The Leak model provides a range of simpler dynamic clamping behaviours where the simulated conductance does not have a time-dependent aspect. A number of different types of leak are supported within this model, currently Linear, GHK, Boltzmann and User-defined types are available.

## Linear leak

A linear leak is a very simple form which represents current leakage through a cell membrane with constant conductance, in other words a simple ohmic resistance relative to a reversal potential. The relevant equation is:

$$I = G_{leak}\left(E_r - V\right)$$

where V is the membrane potential, $E_r$ is the reversal potential and $G_{leak}$ is the leak conductance. You will recognise this as being Ohms law modified to take the reversal potential into account, as elsewhere the difference between $E_r$ and $V$ is calculated in such a way as to generate a leak current in the correct direction. If you want to use this model to cancel-out an existing membrane leak you should set a negative $G_{leak}$ value.

## GHK leak

The Goldman-Hodgkin-Katz leak simulates the behaviour of a leak caused by diffusion of an ion through a membrane via always-open ion channels with the diffusion driven by the differing ionic concentrations on either side of the membrane and the voltage difference across it. This might be used to simulate an unvarying GABA receptor conductance which is driven by the differential concentrations of Cl⁻ ions. The equation used was originally described by Goldman, Hodgkin and Katz:

$$I = -V\frac{G_{ghk}}{C_{out}}\left(\frac{C_{in} - C_{out}\exp\left(\frac{-VFZ}{RT}\right)}{1 - \exp\left(\frac{-VFZ}{RT}\right)}\right)$$

where $V$ is the membrane potential, $G_{ghk}$ is the slope conductance ($dI/dV$) at minimum membrane potential, $c_{in}$ and $c_{out}$ are the ionic concentrations inside and outside the cell respectively, $F$ is Faraday's constant, $Z$ is the valency of the ion in question, $R$ is the universal gas constant and $T$ is the absolute temperature. This equation explodes when $V$ is equal to zero, but we can use a Taylor expansion to derive the limiting value as $V$ tends to zero and get:

$$I = -G_{ghk}\left(\frac{C_{in} - C_{out}RT}{FZC_{out}}\right)$$

## Boltzmann leak

This form of leak current resembles a linear leak but with an additional factor in the equation that represents a voltage-dependent blockage of the leak. This might be used to simulate an unvarying NMDA receptor conductance, with its voltage-dependent block by extracellular magnesium. The equation is:

$$I = \frac{G_{leak}\left(E_r - V\right)}{1 + \exp\left(\frac{V - V_{half}}{V_{sl}}\right)}$$

where $V$ is the membrane potential, $E_r$ is the reversal potential and $G_{leak}$ is the maximum leak conductance, while $V_{half}$ is the potential at which 50% of the total leak conductance is available and $V_{sl}$ controls the rate at which the leak conductance is blocked, this corresponds the change in potential over which a certain amount of conductance is blocked, so a smaller $V_{sl}$ means that the blockage is applied more abruptly as the potential changes.

**User-defined leak**

A user defined leak uses a lookup table read from a text file on disk to define the leak behaviour. This allows you to generate customised leak currents that follow any required behaviour, no reversal potential is defined so if this behaviour is required it should be incorporated into the table values. With a user-defined leak the extra leak parameters are replaced with a field holding a text file name, the text file defines how the leak current (note well: current not conductance) varies with the membrane potential. There are many ways that such a text file could be created; one of the easiest is to use a Signal script. The example GenLeak.sgs script that is installed with Signal is provided as an example of how to do this.

The text file name can be a fully qualified file path and name such as "C:\Signal7\MyLeaks\XLeak.txt" that completely specifies the directory and file name or it can be a simple file name such as "XLeak.txt" in which case the directory where the sampling configuration file is stored is used. If a partial path and file name such as "MyLeaks\XLeak.txt" is used then the path to the file is generated by starting with the sampling configuration directory and appending the text from the model. If no file extension is provided then .txt is appended. The table below illustrates this:

| File name in model | Configuration directory | Model data read from |
| --- | --- | --- |
| C:\MyLeaks\XLeak.txt | C:\Signal7 | C:\MyLeaks\XLeak.txt |
| XLeak.txt | C:\Signal7 | C:\Signal7\XLeak.txt |
| MyLeaks\XLeak.txt | C:\Signal7 | C:\Signal7\MyLeaks\XLeak.txt |
| XLeak | C:\Signal7 | C:\Signal7\XLeak.txt |

Leak currents that vary with the sweep state are generated by using separate file names in the model settings for each state. You can get Signal to automatically generate text file names that are derived from the sampling state by including the text "%ST%" in the text file name, this will be converted by Signal to the current sampling state as a two digit number (00 to 99) or three digits (100 to 256) as appropriate.

The table data format in the text file is a pair of numbers on each line separated by spaces or a single tab character. Lines beginning with a single quote, " ' " are regarded as comments and will be ignored. The first number on the line is a membrane potential in millivolts as measured by the leak model input channel, the second number is the leak current in pA (the actual current injected will be the inverse of this to take account of the internal current injection electrode). The actual leak current that is generated is scaled by the $I_{scale}$ model parameter, allowing you to generate leak table values that run from 0 to 1 and scale them to the required leak current using $I_{scale}$ or alternatively put absolute values in the table and set $I_{scale}$ to unity. Negative leak currents are permitted and correspond to a leak current injection that would act to cancel-out existing leaks. The data in the text file should be sorted in order of membrane potential values with the lowest (most negative) value first and the highest (most positive) value last.

*Table usage and requirements*

When the text file data is used Signal reads all of the table values from the text file and converts them to a second lookup table which is 257 points long and exactly covers the ADC range. This conversion is done using cubic splining which will generally produce a very accurate result, but it does impose requirements upon the text file data. Firstly, the range of membrane potentials for which leak values are given should, in addition to covering the expected range of membrane potentials, exceed by a few points (5 at each end is ample) the entire range of potentials that could possibly be sampled from the input channel given the ADC port calibrations in use. This is necessary in order for the cubic splining to produce accurate results close to the table extremes. If the table values provided do not cover the entire ADC range then the generated table will be truncated at the first or last text file value as appropriate and a warning will be generated.

Secondly, the membrane potential values should be sufficiently close together to accurately represent the leak behaviour. 100 values is probably the bare minimum that would be satisfactory and 250 to 500 is recommended. Evenly spaced potential values will probably be the easiest to produce but this is not a requirement - if your model displays quickly-changing behaviour at some potentials then you should design your text file to have more closely-spaced values at these points while fewer points in areas with essentially linear behaviour will not cause inaccuracy.

### Leak configuration

Adding or editing a Leak model brings up a dialog allowing the parameters of the model to be defined.

The Model Name field sets a convenient name for the model as usual. The Input Channel is the waveform channel number as displayed in the file view in Signal, as usual it must have units of mV in order to be recognised as a valid input of the membrane potential. The Control DAC is the DAC used to inject the current and must be calibrated in units of either pA or nA and have zero calibration offset – these values are set in the DAC scaling items in the Outputs page in the sampling configuration.
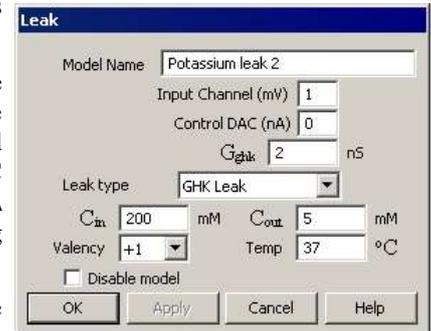
The leak conductance in nanosiemens for all types of leak is set by the $G_{leak}$ field – this changes to the minimum slope conductance $G_{ghk}$ for the GHK form. The Leak Type selector just below selects between the various types of leak, the remaining parameters such as the reversal potential $E_r$ are shown or hidden depending upon the type of leak selected and are described with the individual types of leak, the final parameter is the model enable flag controlled by the Disable model check box.

The settings for a user-defined leak replaces the parameter controls used for the other leak models, with extra fields and controls to define and check the leak table. The File name field sets the name of the text file holding the table data, the Browse button allows you to browse the disk to select a file while the Check File button checks the selected file for satisfactory contents.

# Noise model

The Noise model provides a mechanism for generating simulated noisy conductances, which are increasingly recognised as having an important effect on neuronal behaviour. Currently a single type of noise generation mechanism is available; this uses an Ornstein-Uhlenbeck process to model noisy behaviour. Ornstein-Uhlenbeck noise is available in a standard form and three scaled forms that use data from a user-defined lookup table, GHK behaviour and Boltzmann behaviour to scale the noise according to the membrane potential at any given time.

### Ornstein-Uhlenbeck noise

This model generates noise using an Ornstein-Uhlenbeck process, also known as the mean-reverting process, which is a stochastic process $X_t$ given by the following stochastic differential equation:

$$dG_t = 1/\tau((G_{base} - G_t)dt) + \sigma N$$

where $G_t$ is the noisy conductance at time t, $\tau$ is a time constant controlling the decay of the noise back to a baseline conductance value set by *Gbase*, σ scales the standard deviation of the noise and N is a random value obeying Gaussian statistics with a mean of zero and standard deviation of 1.

### Noise configuration

Adding or editing a Noise model brings up a dialog allowing the parameters of the model to be defined.

The Model Name field sets a convenient name for the model as usual. The Input Channel is the waveform channel number as displayed in the file view in Signal, as elsewhere it must have units of mV in order to be recognised as a valid input of the membrane potential. The Control DAC is the DAC used to inject the current and must be calibrated in units of either pA or nA and have zero calibration offset – these values are set in the DAC scaling items in the Outputs page in the sampling configuration.

The Noise Type selector chooses between the various types of noise, currently the selections available are Ornstein-Uhlenbeck and Scaled O-U. The Rectify model conductivity check box, if set, ensures that the overall conductance - the base conductance plus noise values - never becomes negative by replacing all negative values with zero, only in rare circumstances would you want to allow negative conductances. The $G_{base}$ parameter sets the base conductance for the noise - if this value is non-zero it behaves in exactly the same way as a linear leak conductance added to the noise. The $E_r$ parameter sets the reversal potential both for the $G_{base}$ component and the generated noise. The *SD* parameter is used to calculate the σ value used to scale the random values (σ is calculated as `SD*sqrt((interval*2)/τ)`) and sets the standard deviation of the noisy conductance in nS and τ is the decay time constant for the generated noise in milliseconds, the final parameter is the model enable flag controlled by the Disable model check box.
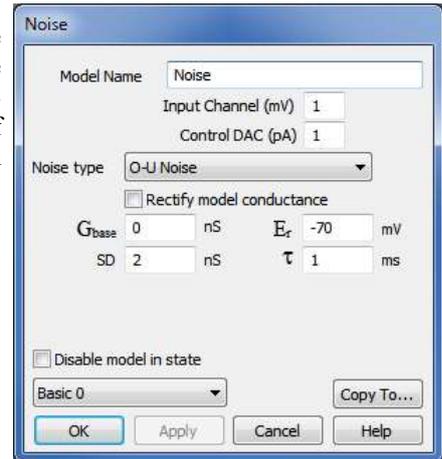
**Scaled Ornstein-Uhlenbeck noise**
The standard form of the Ornstein-Uhlenbeck noise model generates current in a linear, ohmic, fashion based upon the current $G_t$ value plus $G_{base}$, the model reversal potential and the membrane potential at the current time. This is satisfactory for many purposes but may behave rather differently from some sources of biological noise. Therefore three scaled forms of the model are available: GHK, Boltzmann and User.
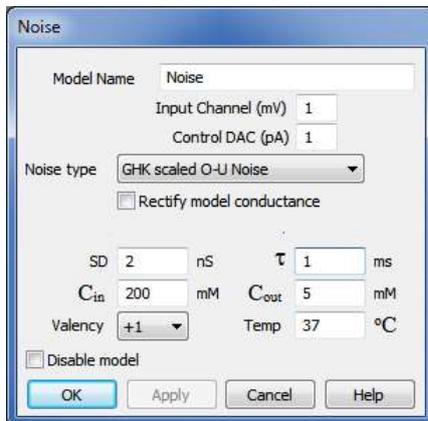
*GHK scaling*
This form of the noise model generates noise using the same Ornstein-Uhlenbeck process as the standard O-U model but the noise level is scaled using the GHK equation so that it behaves like current generated by an ion-specific channel driven by differing ionic concentrations inside and outside the membrane. See the documentation of the leak model for complete details of the GHK equations, here the $G_{ghk}$ value is replaced by the output of the noise generator and the $G_{base}$ and $E_r$ parameters are not used - there is also no reversal potential as this is implicit in the effect of the differing ionic concentrations.

*Boltzmann scaling*
This form of the noise model generates noise using the same Ornstein-Uhlenbeck process as the standard O-U model but the noise level is scaled using the Boltzmann equation so that it shows voltage-dependent blocking of the ion channel. See the documentation of the leak model for complete details of the Boltzmann equation, here the $G_{leak}$ value is replaced by the output of the noise generator. The $G_{base}$ and reversal potential parameters are used, $G_{base}$ is added to the noisy output after the Boltzmann scaling. Finally the output is converted to current using the membrane and reversal potentials.

## User-defined scaling

This form of the noise model generates noise using the same Ornstein-Uhlenbeck process as the standard O-U model but the noise level is scaled by data from a lookup table indexed-into by the membrane potential. This allows the amplitude of the noisy conductance to vary in an arbitrary manner controlled by the membrane potential, for example you could use this mechanism to generate a noisy non-linear leak. The lookup table data is read from a text file stored on disk, there are many ways that such a text file could be created; one of the easiest is to use a Signal script. The example GenLeak.sgs script that is installed with Signal is provided as an example of how to do this, though note that in this model the table data is a conductance scaler rather than an actual output current.

The settings for scaled Ornstein-Uhlenbeck noise are mostly the same as for the standard O-U noise, with extra fields and controls to define and check the scaling table. The File name field sets the name of the text file holding the lookup table data, the Browse button allows you to browse the disk to select a file while the Check File button checks the selected file for satisfactory contents.
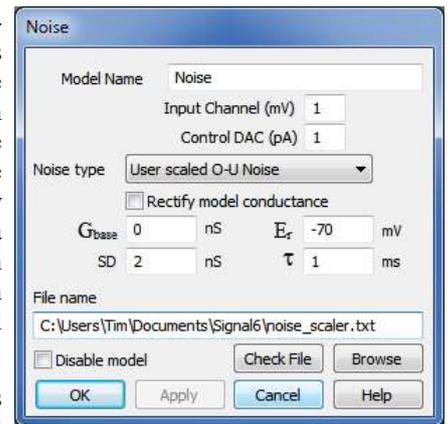
The text file name can be a fully qualified file path and name such as "C:\Signal7\MyNoise\XNoise.txt" that completely specifies the directory and file name or it can be a simple file name such as "XNoise.txt" in which case the directory where the sampling configuration file is stored is used. If a partial path and file name such as "MyNoise\XNoise.txt" is used then the path to the file is generated by starting with the sampling configuration directory and appending the text from the model. If no file extension is provided then .txt is appended. The table below illustrates this:

| File name in model | Configuration directory | Model data read from |
| --- | --- | --- |
| C:\Signal7\MyNoise\XNoise.txt | C:\Signal7 | C:\Signal7\MyNoise\XNoise.txt |
| XNoise.txt | C:\Signal7 | C:\Signal7\XNoise.txt |
| MyNoise\XNoise.txt | C:\Signal7 | C:\Signal7\MyNoise\XNoise.txt |
| XNoise | C:\Signal7 | C:\Signal7\XNoise.txt |

Noise that varies with the sweep state can be generated using different SD values or by using separate file names in the model settings for each state. You can get Signal to automatically generate text file names that are derived from the sampling state by including the text "%ST%" in the text file name, this will be converted by Signal to the current sampling state as a two digit number (00 to 99) or three digits (100 to 256) as appropriate.

The table data format in the text file is a pair of numbers on each line separated by spaces or a single tab character. Lines beginning with a single quote, " ' " are regarded as comments and will be ignored. The first number on the line is a membrane potential in millivolts as measured by the noise model input channel, the second number is a scaling factor that will be used scale the noise. The noisy conductance that is generated using the standard parameters is multiplied by the relevant lookup table value, then the $G_{base}$ model parameter is added to get the final overall conductance. The data in the text file should be sorted in order of membrane potential values with the lowest (most negative) value first and the highest (most positive) value last.

## Table usage and requirements

When the text file data is used Signal reads all of the table values from the text file and converts them to a second lookup table which is 257 points long and exactly covers the input ADC range. This conversion is done using cubic splining which will generally produce a very accurate result, but it does impose requirements upon the text file data. Firstly, the range of membrane potentials for which noise scaling values are given should, in addition to covering the expected range of membrane potentials, exceed by a few points (5 at each end is ample) the entire range of potentials that could possibly be sampled from the input channel given the ADC port calibrations in use. This is necessary in order for the cubic splining to produce accurate results close to the table extremes. If the table values provided do not cover the entire ADC range then the generated table will be truncated at the first or last text file value as appropriate and a warning will be generated.

Secondly, the membrane potential values should be sufficiently close together to accurately represent the noise behaviour required. 100 values is probably the bare minimum that would be satisfactory and 250 to 500 is recommended. Evenly spaced potential values will probably be the easiest to produce but this is not a requirement - if your model displays quickly-changing behaviour at some potentials then you should design your text file to have

more closely-spaced values at these points while fewer points in areas with essentially linear behaviour will not cause inaccuracy.

# Dynamic clamp and multiple states

When using multiple states sampling, the model parameters are extended by the number of states in use to give a separate individually configurable set of parameters for each state. If the number of states in use is increased then the parameters for the highest pre-existing state will be copied to any new states created. Signal automatically switches between these parameter sets whenever the state changes during sampling, so you can quickly and easily observe the effect of different model parameters or, as models can be enabled and disabled independently in the various states, different models.

If multiple states are enabled within the Signal sampling configuration, two extra items appear in the model dialogs. The first is a drop-down selector where the state whose parameters are to be viewed and edited is chosen. Each state has its own set of model parameters and enable switch and these items can be edited separately. The Model name, Input channel and Control DAC numbers, the rectification option for the electrical synapse model and the Leak type item for leak models all behave differently as these values are global to the model and can only be edited when the basic state 0 is selected.

The other item provided for multiple states operations is a button labelled Copy to… which, when pressed, brings up a dialog in which you can select other states to which you wish to copy the currently selected state's parameters. There is a list of check boxes, one per possible destination state, which you can set to enable copying of data to the corresponding state. The All button sets all the check boxes, while the None button clears them all.

# Example leak models

Setting up dynamic clamping requires you to get all of the ADC and DAC calibrations correct as well as to set up the models. Many of the models are quite complex in their operation so it can be very hard to be sure you have got things right. To make things easier, we will start with examining the behaviour of the simpler leak models first, as with these models it is easier to understand what the model is doing and we can concentrate more on general understanding and getting the ADC and DAC calibrations correct.

All of these example models will assume a patch amplifier connected to a model cell with an internal 500 MOhm internal resistance. We shall assume that our amplifier (in current clamp mode) has an external command sensitivity of 2 nA/V and that the output gain settings give a scaled membrane potential sensitivity of 10 mV/mV. We shall also assume that the Power1401 analogue range is set to the standard -5 to +5 volts - this can be changed to -10 to +10 volts using the Try1401 test and setup utility in which case you would need to adjust the calibration calculations.

### ADC and DAC setup

The first thing we have to do is to make sure that we are sampling the ADC inputs that we need to obtain the membrane potential and current injection signals. The list of ports (ADC inputs) to be sampled is set at the bottom of the General page of the sampling configuration, data from the first port in the list appears in channel 1 of the sampled data file, the second port generates data for channel 2 and so forth. It does not matter which ports you use, you just have to make sure that the relevant signals from your clamp amplifier are connected to the correct BNC inputs on the front of the 1401 and that the calibration information that you enter is for the matching ports. You should also choose a DAC output that you will use to control the current and connect this to the amplifier external command input. If your amplifier is able to provide an additional membrane current output in addition to the main membrane potential output that should also be sampled, otherwise the DAC output controlling the current should be connected to both the amplifier external command input and the ADC input of your choice. The amplifier main or membrane potential output should be connected to the ADC input you will use for sampling membrane potential.

The sampling rate for all of the ADC inputs is also set in the General page of the sampling configuration dialog, this should be set fast enough to give a rapidly updated dynamic clamp output while not overloading the 1401. A rate of 20 to 50 KHz will do nicely, if you encounter instability with the Hodgkin-Huxley models you may find that still higher sampling rates will help with this. You might also want to adjust the length of the sampled sweep to suit your requirements.

Then we must ensure that the ADC input ports are correctly calibrated, with units set to mV for the membrane potential signal and nA or pA for the current channel, and that the scale factors match the amplifier settings. Ports calibration is done in the Ports page of the sampling configuration, you get a dialog allowing you to change the settings for a port by double-clicking on the line holding the relevant port's information. For the membrane potential the port units should be set to mV and the Zero value should be 0 (because the amplifier generates a zero output for zero membrane potential). With an amplifier output sensitivity of 10 mV/mV and a 5 volt full scale ADC input, the full-scale calibration value should be 500 mV.

Similarly, the port used to sample the external command signal from the DAC should use units of nA (or pA if you prefer), again the zero value should be 0 and the Full value should be 10 nA (or 10000 pA).

Finally, the DAC calibration for the external command DAC controlling the current should to be set up, this is done in the Outputs page of the sampling configuration. We shall calibrate the DAC in nA (again you could use pA). So we set the units field for the DAC to nA and the Zero value to 0 (because a zero external command voltages causes zero current to be injected). The command sensitivity is 2 nA/V, with a 5 volt full-scale output this gives us a full scale value of 10 nA (or 10000 pA). This matches the calibration of the ADC input used to sample the current control signal, ensuring that they are correctly measured.

### Linear leak model

With the ADC inputs and the DAC output defined and calibrated we can now set up a leak model, we shall start with the simple linear leak. That is done relatively straightforwardly. First open the main dynamic clamp setup dialog by pressing the Dynamic Clamp button in the Clamp page of the sampling configuration. If the Clamp page is not visible, you should enable clamping features by using the Clamp page of the Preferences dialog.

Once you have the main dynamic clamp setup dialog open, use the Add Leak button to add a leak model and open the leak settings dialog to configure it. You must specify the correct input channel (the channel used to record membrane potential) and control DAC (the one connected to the amplifier external command input), select the linear leak form of the model and set the conductance to 2 nS and the reversal potential to 0 mV. Finally set the Disable model check box as we want to start off sampling without the leak. Press OK to save the leak model once you are done.

To see the leak in action we will need a varying input so add a current stimulation pulse to the sampling configuration by enabling the current control DAC for pulses output and adding a 100 millisecond square pulse, amplitude 0.2 nA, using the pulses configuration dialog.

### Linear leak model operation

We start off sampling with the model disabled. During the sampling sweep the current injection will be controlled purely by the DAC output set in the pulses configuration dialog so we will see:

1. At the start of the sweep, the DAC output will be zero, so the current through the model cell will be zero, and the membrane potential would also be zero.

2. When the DAC output pulse occurs, the DAC output will be changed to generate a 0.2 nA current through the model cell (at the specified command sensitivity this will be a 0.1 volt output). This current will flow through the 500 MOhm model cell resistance, producing a 100 mV increase in the membrane potential (by Ohms law). Because of the internal capacitance in the model cell this increase in potential will be gradual, but the increase will flatten out at this level.

3. Now show the Dynamic clamp setup dialog by using the Sample menu Dynamic clamp models command or pressing the Models button in the clamp control bar. Double-click on the linear leak model to show the model configuration dialog, enable the leak model by clearing the Disable model check box and press the Apply button. This generates a 2 nS leak across the membrane, simulating to a 500 MOhm resistance in parallel with the model cell. This reduces the effective membrane resistance to 250 MOhm, so the change in membrane potential will be halved to 50 mV.

4.  To finish our experiments with the linear leak model, change the model reversal potential ($E_r$) to -70 mV and press Apply again. The leak will change its behaviour to 'pull' the membrane potential towards -70 mV, as the leak is in parallel with the cell resistance the effect of this is divided in half to give a baseline level of -35 mV, rising to 15 mV during the stimulation pulse.

**GHK leak model**

We will finish off our investigations of the leak model by switching to a GHK leak that generates a simulation of permanently-open potassium ion channels. Finish off sampling and go to the leak model configuration dialog. Set Leak type to GHK, $G_{ghk}$ to 4 nS, $C_{in}$ to 200 mM, $C_{out}$ to 5 mM, Valency to +1 and Temp to 37 degrees. You do not need to change any of the ADC or DAC calibrations as they are set for the amplifier settings and the connections used.

With the model enabled you will see that the baseline membrane potential has been shifted (driven by simulated ion diffusion) strongly negative to about -80 mV. The effect of the stimulation pulse is also reduced to about 15 mV - the stimulation current is shunted across the membrane by the effect of the leak.

# Example Hodgkin-Huxley model

A Hodgkin-Huxley model is more complex than a linear leak because it simulates ion channels whose behaviour is voltage dependent with complex dynamics. The use and  calibration of the ADC inputs and control DAC are the same as for the example leak models and we can use the same calibration values as long as the amplifier settings remain unchanged. This example will aim to simulate action potentials (reasonably accurately) by starting with the example GHK model in the previous example and adding suitable voltage-dependent sodium channels.

## Model configuration

Starting with the sampling configuration used for the GHK leak example, open the main dynamic clamp setup dialog, select and and use the Add HH (AB) button to create a new Hodgkin-Huxley model based on multiple time constants and open the dialog to edit it. The model has a large number of parameters which have to be set correctly:



| $G_{max}$ | 17000 nS |
| --- | --- |
| $E_r$ | 84 mV |

| | *Activation* | | *Inactivation* |
| --- | --- | --- | --- |
| *Include* | ON | *Include* | ON |
| $p$ | 3 | $q$ | 1 |
| *Fa(x)* | x/(exp(x)-1) | *Fa(x)* | exp(x) |
| $k_a$ | 0.01 ms$^{-1}$ | $k_a$ | 0.07 ms$^{-1}$ |
| $V_a$ | 40 mV | $V_a$ | 65 mV |
| $S_a$ | 10 mV | $S_a$ | 20 mV |
| *Fb(x)* | exp(x) | *Fb(x)* | 1/(1+exp(x)) |
| $k_b$ | 4 ms$^{-1}$ | $k_b$ | 1 ms$^{-1}$ |
| $V_b$ | 60 mV | $V_b$ | 35 mV |
| $S_b$ | 12 mV | $S_b$ | 10 mV |
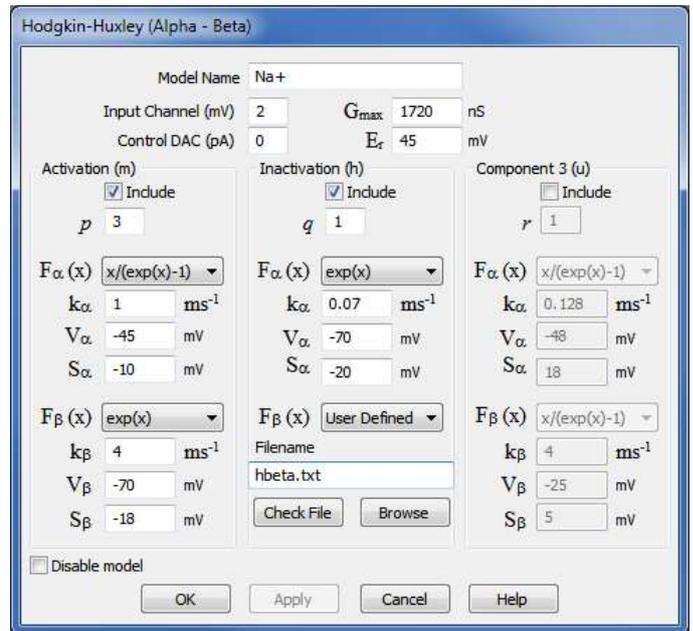
When you run this sampling configuration with both models enabled you will still see the potassium leak holding the resting potential down, but you should see a much more spectacular response to the 0.2 nA stimulation pulse:
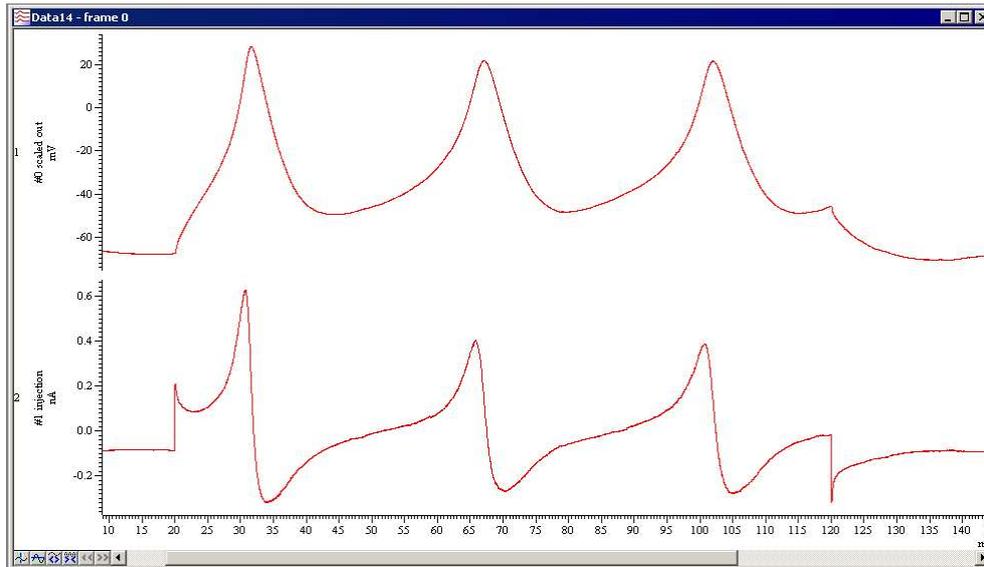
If you experiment with changing the size of the stimulation between 0.1 and 1 nA you will see a wide range of behaviours.

# Sampling with dynamic clamp

When data is sampled using a sampling configuration that includes dynamic clamping, Signal provides a dynamic clamping toolbar that you can use to control models and watch for slow updates.

This toolbar is part of the basic clamping control toolbar and is attached to the bottom or right of the standard clamp set controls (if any). Like all of the sampling control bars within Signal, it can be positioned docked at any edge of the application window or floating anywhere on the screen. The dynamic clamping toolbar provides



four controls; a Models... button which provides the main dynamic clamping setup dialog, a Disable all check box which can be used to enable or disable all models, a DB (deblock) button which, when pressed, temporarily disables all dynamic clamping models and a text area where information about any slow updates in the last sampling sweep will be displayed. Note that the Disable all check box acts to enable or disable all the models in all the states, it is shown unchecked if any models are enabled in any of the sampling states. The DB button is often more useful for recovery from instability than disabling all models as it automatically clears when the mouse button is released.
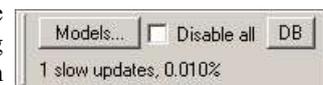
The Sample menu also has a Dynamic Clamp models item which is enabled if sampling with dynamic clamping is in progress. When this item or the Models... button is used the main dynamic clamping setup dialog is provided in its online form. When used online this dialog and the individual model dialogs look and behave differently from when they are used to define a sampling configuration as follows:

- The Add button in the main setup dialog is disabled.

- The Delete button in the main setup dialog is replaced by an Enable/Disable button which shows and changes the enabled state of the currently selected model. If it is used to change the state of a model it enables or disables the model in all states, the button shows Disable if any of the states for the model are enabled, otherwise it shows Enable.

- When a dialog to view and edit model parameters is activated by double-clicking on a model or using the Edit button, the main setup dialog remains active and can be used to open other model parameter dialogs, allowing multiple model parameter dialogs to be simultaneously available.

- Both the main setup dialog and the individual model parameter dialogs run modelessly which means that you can interact with the rest of Signal while the dialogs remain visible and in use.

- In the individual model dialogs the global model parameters such as the name, input channels and output DACs cannot be changed – these are the values that can only be changed if state zero was selected when editing offline.

- The Apply button in the individual model dialogs are now active. Pressing Apply will copy any changed parameter information back into the sampling configuration and update the 1401 without quitting the dialog or

terminating sampling. Updating model parameters during sampling should not cause any problems beyond, in infrequent special cases, momentary instability.

In addition to control of dynamic clamping online using the standard dialogs and the clamping control bar, the Signal outputs sequencer also provides a DCON instruction that can be used to enable or disable dynamic clamping models dynamically during a sweep as well as in between sweeps.

### Update rates and slow updates

When running dynamic clamp models, every time a set of ADC samples has been taken & stored the 1401 then iterates the dynamic clamp models to calculate new DAC values and update the DACs. Thus the update rate of the dynamic clamp is the same as the ADC sampling rate and forced to run in step with the sampling. If the DACs have not been updated by the time the next set of ADC samples has been taken Signal will detect a slow update. The dynamic clamp section of the clamping toolbar shows information about slow updates in the last sweep taken; a warning message will be generated at the end of sampling if any slow updates were detected.

Slow updates means that the dynamic clamp system is not completely keeping up with the sampling. They are not necessarily a problem but may be associated with short glitches or other small errors in the DAC outputs and therefore the warning message indicates that the DAC outputs cannot be wholly trusted. If your sampling configuration suffers from slow updates the only reliable remedy is to reduce the ADC sampling rate, but you might be able to get rid of them by setting the Maximise waveform rates check box in the Outputs page of the sampling configuration dialog, this acts to give dynamic clamp updates a higher priority compared to standard outputs generation.

If, however, by the time the DACs have been updated two complete sets of ADC samples has been taken then the dynamic clamp updates have failed to keep up at all and Signal will terminate sampling with an error. This failure is generally indicated by a -1,0 error from the 1401, though it can show up as other errors, but you will not encounter it unless you are using sampling rates very much higher than the dynamic clamping can cope with.

In tests carried out at CED, four Hodgkin-Huxley models were seen to cope with ADC input rates of up to 60 kHz before slow updates were reported. With 15 Hodgkin-Huxley models it was possible to sample eight ADC channels at 30 kHz without seeing any slow updates while with only a single model a sample rate of 100 kHz can be attained.

### Amplifier gains

Many of the dynamic clamp models, in particular the Hodgkin-Huxley, CPG synapse and noise models, are driven by pre-calculated lookup tables held inside the 1401. These tables are used to convert a membrane potential value into values used to execute the model. Because the table is a fixed size (currently 256) which spans the entire range of a model input channel, the voltage step between table entries wholly depends upon the amplifier gain. Consider a Hodgkin-Huxley model driven by an amplifier with the gain set to 50. This will give an ADC input range (assuming a 1401 set to 10 volt range) of -200 to +200 millivolts. With a table size of 256 this gives a voltage step size of about 1.56 millivolts which will give an excellent representation of the model behaviour. However if the amplifier gain is reduced to 2 the ADC input range becomes -5000 to +5000 millivolts and the table step size becomes much larger, about 40 millivolts. You could hardly expect such a table to be able to represent model behaviour when the entire voltage range of interest is covered by only 5 table entries! And indeed if you try this out with a model cell you will see the model failing when the gain gets below about 10. The upshot of all this is that you should try to use the highest amplifier gain possible for the membrane potential signals that drive dynamic clamp models, without of course setting the gain so high that the available signal range is too low for the signals generated. A range of about -200 to +200 millivolts would seem optimal for most purposes.

### Model stability issues

Most of the dynamic clamp models are inherently stable and so long as the amplifier setup matches the ADC and DAC scalings they should always behave as you expect. However the Hodgkin-Huxley equations are highly non-linear and can easily exhibit instability which can make them very hard to use. I cannot give you a simple fix for this, but I can make some suggestions. Firstly you should make sure that the current clamp amplifier is correctly configured - this goes well beyond setting the correct mode and gain to include capacitance and series resistance compensation and, very importantly, bridge balance set up. Secondly, we find that any instability issues are at least reduced by higher sampling rates so use the fastest ADC sampling rate that you can and enable burst mode sampling. You can also reduce the effective calculation delay slightly by ensuring that the membrane potential channels are sampled at the end of the list of ADC ports, which may help in some circumstances. Finally, we have found that the models become more unstable if the Gmax values are higher so, if your experimental technique allows this, you could try reducing these.

# File menu

The File menu is used for operations that are mainly associated with files (opening, closing and creating) and with printing.

## New

This command creates a new Signal file. This can be a sampling document, an XY file, or a new text-based file. You can activate this command with the `Ctrl+N` shortcut key or from the menu or toolbar.

The command opens the New File dialog in which you select the type of new document to create. You can create five types of document: Signal data documents, script documents, sequencer documents, text documents and XY documents. Select the type of document, click OK and Signal will open a new window holding an empty document of the specified type.

### Data Document

This choice opens a sampling document window plus additional windows as set by the sampling configuration (see *Sampling data* for details). Sampling documents are not initially stored in memory, like most new files, but are kept on disk. Until they are saved after sampling these are temporary files, without any extension, stored in the directory set by the Filing path in the Automation page of the sampling configuration or if that is blank, the new data directory set in the Sampling page of the Preferences dialog. When they have been saved, the file name extension is `.cfs` and the files hold CFS (CED Filing System) format files.

### Script Document

This choice opens a script editor window in which a new script can be written, run and debugged. A script document is a specialised form of text document. The file name extension is `.sgs`.

### Sequencer Document

This choice opens a  sequence editor window in which you can type, edit and compile an output sequence. The file name extension is `.pls`.

### Text Document

This choice opens a text window. Text documents can be used to take notes, build reports and to cut and paste text between other windows and applications. The Log view is a specialised type of text document which is always present. The file name extension is `.txt`.

### XY Document

This choice opens an XY window. XY windows are used to draw user-defined graphs with a wide variety of line and point styles. These views can be created interactively when generating a trend plot or measurements, their other major use is from the script language. The file name extension is `.sxy`.

## Other file types used by Signal

In addition to the standard document types, Signal also uses a number of other types of file:

### Resource files

Signal creates resource files with the extension .sgrx, older versions of Signal used files with a .sgr extension. Each resource file is associated with a data file of the same name but with the extension .cfs. The resource files hold configuration information about the data file display so that Signal can restore the display on loading. These files are not essential to Signal and if you delete them the associated data file is not damaged in any way, but you will lose your display settings, experimenter's notebook data and any virtual and memory channels that you have added to the data file.

### Configuration files

Signal stores sampling configuration information in files with the extension .sgcx, older versions of Signal used files with a .sgc extension. These store all of the information needed to carry out sampling: the sampling parameters, the position of the sampling control panel and any other windows, the position and display configuration of the data document window (including any duplicate windows) plus any online processing required, the online processing parameters and the position and display configuration of the memory views showing the results of online processing.

### Application preferences

Signal stores its preferences in the Windows registry, this holds the position of the application window on the screen, the colour palette and the main preferences information from the preferences dialog. The main reason for doing this rather than storing preferences in a file is that Windows provides a different registry for each logged-on user, so each user can have Signal set up the way they prefer. Signal initialises itself with data from the registry whenever it is started and saves the current state of the software into the registry when it exits.

### Filterbank files

These files hold descriptions of digital filters and have the extension .cfbx. , older versions of Signal used files with a .cfb extension. They are used by the dialogs provided by the Analysis menu Digital filters command.

# Open

This command opens a file into a Signal document of any type. You can activate this command with the Ctrl+O shortcut key or from the menu or toolbar. It shows the standard file open dialog for you to select a file. You can open five types of file with this command: a Signal data file with the standard extension .cfs, a text file with the extension .txt, a sequencer file with the extension .pls, a script file with the extension .sgs or an XY data file with the extension .sxy. The type of the file is selected with the Files of type field, if this is set to All files(*.*), Signal will try to open the file selected as a CFS file whatever its extension.
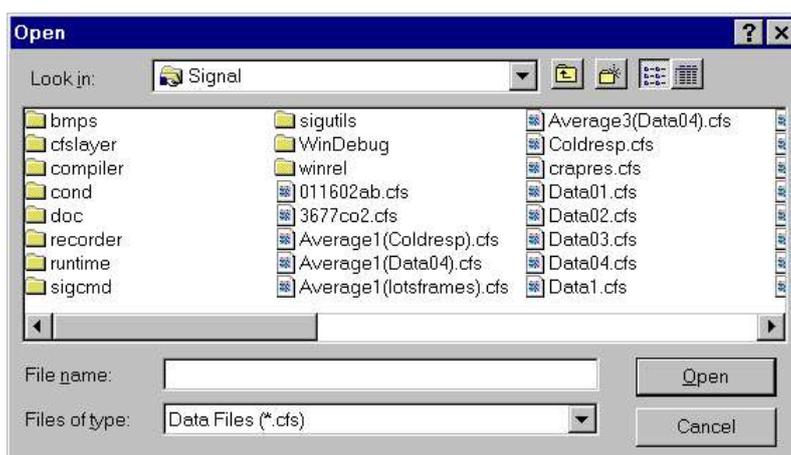
When you select a CFS data file, Signal also looks for a file of the same name, but with the file extension .sgrx (or .sgr for older files). If this is found, the new window displays the file in the same state and screen position as it was put away. Several windows may open if the file was closed with the Close All command. See the Close and Close All commands for more details.

If a read-only CFS data file is opened, you will be warned that the file cannot be modified.

If a text file is opened, a simple text edit window is opened. This facility can be used as a notepad or as a repository for data copied from other parts of Signal. If a script, sequencer or XY file is opened, a window of the appropriate type is created for it.

# Import data

Signal can translate data files from other formats into Signal data files. The File menu Import command leads to a standard file open dialog in which you select the imported file format and the file to convert. You then set the file name for the result; Signal will suggest the same name with the extension changed to .cfs. The details of the conversion depend upon the imported file type.

Supported formats include the SON files used by CED programs such as Spike2 as well as data from many third party vendors. Signal searches the import folder in the Signal installation folder for CED File Converter DLLs. If

you need to translate a file format that is not covered, please contact us and describe your requirements. The script language command used to import files is `FileConvert$()`.

Data import has two main phases:

1. Scanning the input files to detect the file contents, channel types and data and time ranges present.

2. Transfer of data into a Signal CFS data file

The scanning phase is entirely handled by the file converter DLL. This can take a considerable time, especially if the data is held in many separate files. The transfer of data is a co-operative effort between Signal and the file converter DLL.

### Display during import
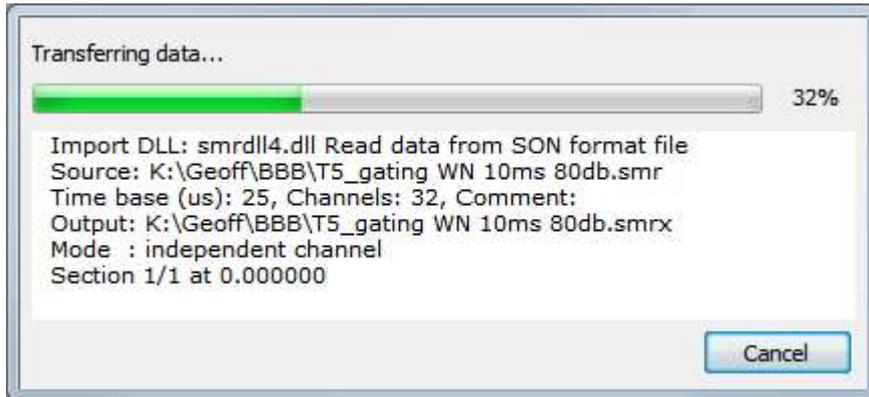


Data file import can take a considerable time, especially if the imported file is very large or the data is spread over many files. Signal will open a resizeable progress dialog to give you information about how the import is proceeding and to give you the option of cancelling the import. This dialog may be unresponsive with some importers, particularly during the scanning phase. Modern importers will give you more feedback during scanning but in some cases, you may not be able to cancel import until the data transfer phase has started.

The progress dialog closes when the file has been transferred; the text contents of the dialog are written to the Log view when the file closes, which can be useful if there were problems during the import.

### Importer Configuration Files
Some importers store configuration files in the import folder; these files have the extension .icf. The last used configuration for importer XXX is saved in XXX_Last.icf. If a default format exists, it is saved in XXX_Def.icf. If the _Def.icf file exists, it is used, otherwise the _Last.icf file is used. Most importers will survive without any configuration information as the built-in default settings will do something useful. However, if you intend to import a binary or text file with the `FileConvert$()` script command it is essential that you have created configuration files, otherwise these importers will not know what to do with the data.

## File types we import
We import data from a wide variety of file types and we attempt to preserve as much of the original data content and accuracy as we can. If there is a file type that you would like imported that is not in the list, please contact us. To import a new type we need accurate file format information. We would like to thank the companies that made such information available to us for their help. Some companies do not release their file formats; in a few cases we have managed to figure out enough to extract useful information. In cases where companies do not release the information and we have not been able to figure it out, you can sometimes export data in other formats that we can import; of course, every change of format will tend to reduce the quality and quantity of data that is imported.

If you find a problem in any importer, please contact us and explain the problem; in many cases the importers were written with only a small number (sometimes 1) of examples of the source data. We will need a source data file that fails to convert.

| Data source | Extensions | DLL | cmd$ | Notes |
|---|---|---|---|---|
| Alpha MED Sciences | `DAT,MODAT` | `med64` | | Conductor, Performer and Mobius data. Can import files larger than 2 GB. |
| Alpha Omega Engineering | `MAP, MPX` | `alpha` | | Old format event data supported. MPX supported. |
| Axon Instruments (Molecular Devices) | `DAT,ABF` | `axon` | | **32-bit only.** Imports 16-bit integer and 32-bit float. Library updated Jan/2014. |
| Axona | `BIN` | `axona` | | |
| Binary data | `BIN,DAT` | `binary` | Yes | Can import 8/16/32/64 bit signed and unsigned integers data in big and little-endian formats. |
| Bionic/Cyber kinetics | `NEV,RND,NS*` | `bionic` | | Can import files larger than 2 GB. |
| CED SON (Macintosh) | `SMR,SON` | `sondll` | | |
| COLD_Datein | `*` | `colddll` | | Pulsion Medical Systems. |
| CONSAM | `SSD,DAT` | `consdll` | | From Prof. D. Colquhuon. Supports version 1001, 1002. |
| Data Sciences International | `*` | `dataq` `dataqv5` | | **32-bit only.** You need a dongle to import this file format. Contact CED for details. Files with 4 character extensions supported. Can import all files up to version 5. You should use the dataqv5 importer unless you have problems. |
| DATAPAC | `PAR,PBR,PCR` | `datapaq` | | |
| DATAQ Instruments | `ACQ` | `codas` | | (Codas) |
| DataWave | `UFF,CUT EWB` | `uffdll` | | Imports both Discovery and Workbench files. |
| Delsys Files | `EMG` | `delsys` | | Version 4 supported. |
| Elmiko medical | `*` | `elmiko` | | |
| European Data Format(+) | `EDF,BDF` | `edf` | Yes | Imports 16 and 24-bit data. |
| Grass-Telefactor | `BIN` | `polyview` | | (PolyView) We import versions from 2.0 |
| Heka Data Format | `DAT` | `heka` | | |
| HLR Data Format | `HLR` | `hlr` | | From software Platon and Pyton version 3.00 and 4.00. |
| Intan | `*` | `intandll` | | |
| MindSet (16/24) data Files | `BIN` | `mindset` | | MindSet, MindMeld |

| Data source | Extensions | DLL | cmd$ | Notes |
|---|---|---|---|---|
| MindWare data files | `MW` | `mindw` | | |
| Motion Lab Systems | `C3D` | `c3d` | | From 8.12 the length of channel units was limited to 32 characters. |
| Multi Channel System (Mc_Rack) | `MCD` | `mcdata` | | We can import 12 and 16-bit data. Library updated Aug/2015, when 64-bit support was added. |
| Native ADC | `ADC` | `nadcdll` | | Used by Polish universities as a common data format. |
| Neuralynx | `NEV,NCS,`<br>`NTT,NSE,`<br>`NST` | `nlynx` | | Waveform channel y scaling factor corrected Jan/2014. |
| NeuroScan | `EEG, CNT` | `neuro` | | Can read both 16 and 32-bit data. |
| NewBehavior | `HEX` | `neurolog` | | (Neurologger) old and 2012 formats. Jan/2104: added a dialog to select the recording session to import. |
| Plexon | `NEX,PLX,`<br>`DDT,PL2` | `plexon` | | Can read data generated by the Plexon version 1.07 library. |
| RC Electronics | `PRM,INX,`<br>`DAT` | `rcelectr` | | Imports 12 and 16-bit data. |
| Ripple Neuro | `NEV,NS*,`<br>`NF*` | `ripple` | | Extends the `NEV` file format to handle floating point data (`NF*` files). |
| Spike2 | `SMR,`<br>`SMRX` | `smrdll` | | |
| Text files | `TXT,ASC` | `asciidll` | Yes | Fixed a Marker timing issue in Jan/2014. Treats `.csv` files as comma separators only (version 8.03). |
| TMS International | `S00` | `tmsi` | | Marker format is untested as we have no example of it. |
| Tucker-Davis Technologies | `TSQ` | `tdtdll` | | You also need the `.tev` file to import the data. |
| WAV (Microsoft) | `WAV` | `mswave` | | Handles 8, 16, 24 and 32-bit integer `WAVE_FORMAT_PCM` format data. |
| WaveMetrics Igor Pro | `IGO,IBW,`<br>`PXP` | `igor` | | Imports both PC and Mac files up to and including version 5. |
| Xltek Neuroworks | `STC` | `neurowrk` | | Imports version 3.3 onwards. The `.stc` file holds a list of all the data files that make up the data (`.erd`/`.etc`). We import all the raw waveforms, an event channel of Trigger data, and user-comments made during sampling (`.ent`). |

## Import problems

If there are disk read problems or the input is damaged the import is likely to end with an error message. However, there are other problems that can occur during data import that we can work around.

Data is input section by section. Some files may have only one section, others may have many. A section can refer to a single sweep of data from a sweep-based capture system or it may refer to data captured after a particular time marker; it is a flexible concept.

Within each section, data is captured channel by channel in ascending time order.

The output data is written in strictly ascending time order into the Signal data file. Signal data files have the rule that within a channel, all data is in ascending time order and no two items can occur at the same time.

When we import data, we check that all the data meets these requirements. When we import event-based channels (event, level event, marker, WaveMark, TextMark and RealMark data), we read the data block by block and convert all data to simple markers. We perform the following operations on each block:

1.  If we have to apply a time offset, we do so (this can happen when importing sweep-based data where each sweep has an independent time base)

2.  We check that the data is in ascending order of time, if not in ascending time order we sort it into order

3.  We adjust the times of data that occurs at the same time as the previous data point.

# Binary importer

The binary importer uses configuration files `Bin_Last.icfx` and `Bin_Def.icfx`. If the `.icfx` files are not found, old format `.icf` files are used, otherwise even older `.cim` files are used. These files are in the current user specific, application data folder.

The binary importer supports the following names in the `cmd$` string for the `FileConvert$()` command.

| name | value |
|------|-------|
| `Conf` | Configuration file name. If this string is not empty then the provided file name including its full path will be used to load the configuration file. This keyword is always applied first, regardless of its position in the cmd$ string. |
| `Bigend` | Input file origin. Set to 0 if this is a little-endian file (which is almost always the case on modern systems) or to 1 if this is a big-endian file. The default is 0. In a little-endian file, lower significance bytes are at lower file offsets than higher significance bytes. If the 32-bit hexadecimal number 0x87654321 was stored at the start of a binary file, the first 4 bytes would be 0x21, 0x43, 0x65, 0x87 in a little-endian file and as 0x87, 0x65, 0x43, 0x21 in a big-endian file. |
| `Type` | Input data type. Set to 0 for 8-bit signed integers, 1 for 8-bit unsigned integers, 2 for 16-bit signed integers, 3 for 16-bit unsigned integers, 4 for 32-bit signed integers, 5 for single-precision floating point numbers and 6 for double-precision floating point numbers. The default is 2. |
| `Rate` | File sampling rate in Hz. If present it sets the overall file sampling rate. The default rate is 1000 Hz. |
| `Chans` | Sets the number of imported waveform channels. The default is 3. |
| `Names` | Channel name. You provide a list of channel names for each imported channel separated by commas, for example: `Names=Resp,HR,BP`. The default name is `Chan(n)` where n is the channel number. |
| `Units` | Channel units. You provide a list of units for each imported channel separated by commas, for example `Units=bpm,bpm,mmHg`. The default units are `V`. |

Resources are used in the following order:

1. A default state is set.

2. `Bin_Def.icfx` is used, if it exists. If not found, `BinDef.icf` is used, if it exists.

3. If no `Bin_Def` files are found, `Bin_Last.icfx` is loaded, and failing that, `Bin_Last.icf`

4. If the `Conf` keyword is used, and the nominated file exists, it is loaded and overrides the current setup.

5. Any other keywords are then applied.

### An example
The following example imports a binary data file and passes in information as described in the table above. The configuration is read from the file: `c:/s/b.icfx`, then the number of channels to import and the channel names and units are set to override whatever was set in the configuration file. In most cases, you will set a configuration file and not need to override any of the values set in it.

```
var ret$;
var src$, dest$, flag%, err%, scom$;
src$  := "";   'Blank name so user is prompted for the source file
dest$ := "";   'Blank name so user is prompted for the destination file
flag% := 0;
scom$:="Conf=c:/s/b.icfx;Chans=2;Names=HR,BP;Units=bpm,mmHg";
ret$ := FileConvert$(src$, dest$, flag%, err%, scom$);
```

# EDF importer

The EDF importer reads EDF, EDF+, BDF and BDF+ files. It does not implement logarithmic compression of data (but let us know if this is essential for your use and we will consider adding it). It is targeted at reading continuous recordings of data; it may work with discontinuous recordings, but we have no examples of this data type.

EDF data waveforms are treated as 16-bit data with scale and offset and are imported as Waveform data.

BDF data waveforms are 24-bit data and we import them as RealWave data.

BDF files can hold a Status channel that holds up to 16 channels of digital information. If any of these channel holds changes, we import the channel as a level event channel. Currently these channels start as low at the beginning of the recording and each change toggles the state. These digital signals have the same timing resolution as the waveform data.

The time resolution of the imported data file is set as close to 1 microsecond as possible while exactly matching the file sample rate.

### Annotations in EDF+ and BDF+ files

Annotations are time stamps with optional attached data and duration. We ignore annotations at negative times (before the recording started) as this does not fit the Signal file model (these are permitted by the EDF+ standard but we have never seen one). Each annotation record in the raw data has the format:

```
[+-]Time{<21>Dur}<20>{Text1<20>}{Textn<20>}*<20><0>
```

| | |
|---|---|
| `[+-]` | One of + or -. We ignore annotations that start with -. This is the time of the annotation in seconds from the start of sampling. |
| `Time` | The time of the annotation in seconds, written using 0-9 and . (though we will accept almost any reasonable number format). |
| `{...}` | An optional block (`...` represents the contents). |
| `<n>` | A byte with the value `n`, for example `<20>` is a byte with the decimal value 20. |
| `Dur` | A duration, in seconds in the same format as `Time`. |
| `*` | The preceding optional block is repeated 0 or more times |
| `Text1` | The first annotation text. If this is not empty and does not exceed a set length (currently 30 characters), we look it up in a table of known annotations. If it is found, we add a code to the TextMark, if not found, we add this to the table of annotations and give it the next available code. Text in these blocks can only be ASCII codes 1-126 and must not include <20> or <21>. |
| `Textn` | Further annotations at this time with the same restrictions as Text1. We do not look these up in the annotation table. |

If we parse the annotation without error, and the annotation held text other than the time, we add a TextMark at the given time with the annotation code, or 0 if it is too long (assumed to be free text). The standard format of the text attached to the TextMark is:

```
Text1{|Textn}*{ [Dur]}
```

Most annotations we have seen in examples either have no text, or have just `Text1` and may have a duration. However, the format we have used should make it relatively easy to parse the text. If there is a duration, it is added to the end of the marker enclosed in square brackets and separated from the test of the text by a space. You can select an alternative format by using the `FileConvert$()` script command with the `Cmd$` argument.

If none of the annotations in an annotation channel hold any text or duration, we add a simple event channel with events at the annotation times.

## Standard annotation codes

The EDF format definition suggest that there is a standard list of annotations. We have implemented this list and given them fixed codes 1 to 35:

| Code | Annotation |
|---|---|
| 0 | Text longer than the limit or no text |
| 1 | Lights off |
| 2 | Lights on |
| 3 | Sleep stage W |
| 4 | Sleep stage 1 |
| 5 | Sleep stage 2 |
| 6 | Sleep stage 3 |
| 7 | Sleep stage 4 |
| 8 | Sleep stage R |
| 9 | Sleep stage ? |
| 10 | Movement time |
| 11 | Sleep stage N |
| 12 | Sleep stage N1 |
| 13 | Sleep stage N2 |
| 14 | Sleep stage N3 |
| 15 | Apnea |
| 16 | Obstructive apnea |
| 17 | Central apnea |
| 18 | Mixed apnea |
| 19 | Hypopnea |
| 20 | Hypoventilation |
| 21 | Periodic breathing |
| 22 | CS breathing |
| 23 | RERA |
| 24 | Limb movement |
| 25 | PLMS |
| 26 | EEG arousal |
| 27 | Sinus Tachycardia |
| 28 | WC tachycardia |
| 29 | NC tachycardia |
| 30 | Bradycardia |
| 31 | Asystole |
| 32 | Atrial fibrillation |
| 33 | Bruxism |
| 34 | RBD |

35   `RMD`

36-... Sequentially assigned codes for other annotations discovered in the file.

The codes we assign to other annotations are shared between all annotation channels (files can have multiple annotation channels and they may relate to similar signals). There is nothing significant in the codes, other than 1-35 are standard and 36 onwards are in the order of discovery. If you use the `FileConvert$()` script command with the `Cmd$` argument you can remove these predefined annotations and define your own.

Codes are not limited to 0-255, and can extend to much higher numbers if you have many different annotations. Markers in Signal have four marker codes. If the code exceeds 255, then marker code 0 is (code `band` 255) and marker code 1 is (code / 256). That is, marker code 0 is the lowest byte of the integer code, marker code 1 is the next byte (and so on if you have more than 65535 codes).

# EDF Cmd$ keywords

When used from `FileConvert$()`, the following keywords are available through the `Cmd$` argument:

| name | value |
|---|---|
| TMSep | This can be assigned the values 0 (the default, same as omitting the keyword) and 1. If you set 0, TextMark strings are generated to be easily human readable. Multiple annotations in one marker are separate by the vertical bar character \| and any duration is added to the end of the marker as " [duration]". This format is can have problems if the annotation text already contains vertical bars or square braces. If you set 1, multiple annotations are separated by the ASCII character code `Chr$(20)` and any duration is preceded by `Chr$(21)`. For example, text might appear in one or other of these forms: |

```
"Annotation 1|Annotation 2 [2.234]"
"Annotation 1"+Char$(20)+"Annotation 2"+Chr$(21)+"2.234"
```

You might prefer the second format if you were going to use a script to further process the imported data and you wanted to unambiguously separate annotations and durations (characters `<20>` and `<21>` cannot be used except as separators in a legal EDF file).

NoPreDef    If this keyword is present, all the predefined Standard Annotation Codes are removed. The next added annotation will get code 1 unless `AnBase` is used.

AnBase    This sets the first code to use for added annotations. Each time an annotation is added by the `AnList` keyword (see below), or by encountering an unknown annotation that is not more than 30 characters in length, the annotation gets the current `AnBase` value, then the value is incremented for the next item. Put another way, the annotations have consecutive codes from `AnBase`. If you do not use `NoPreDef`, the default `AnBase` value is 36, if you do use `NoPreDef`, it is 1.

AnList    You can pre-set a list of annotations. These are assigned consecutive codes, starting at the current `AnBase` value. To be added, the annotation must not be more than 30 characters long. For example:

```
Cmd$:="AnBase=1;AnList=One,Two,Three,\"this, or that\",last";
```

Annotations `One`, `Two` and `Three` have codes 1, 2 and 3, `this, or that` has code 4 (note use of quotes to include a comma which would otherwise be a separator). `last` has code 5.

AnCode    Use this to set a list of annotations with defined codes. There are pairs of values assigned to the keyword. The first of each pair is a code, the second is the annotation. For example:

```
Cmd$:="AnCode=49,One,50,Two,57,Nine";
```

This assigns codes 49, 50 and 57 to `One`, `Two` and `Nine`. These are the ASCII codes for "1", "2" and "9".

# Heka importer

Heka files are arranged in a tree. You can visualise the structure in the user dialog that is displayed when you use the interactive File menu Import command:
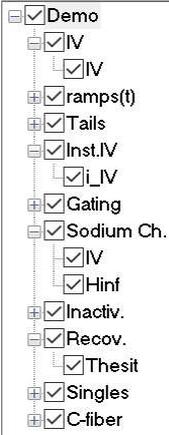
The tree from a typical file (the Heka demonstration file from their web site) is displayed on the left with some of the branches open and some closed. Any file you open will have a similar structure, but the names and number of the branches will vary, depending on the file contents.

The points where the branches meet the Root or the parent branch are called *Nodes*. Nodes of the tree are indicated by a small box holding either a + symbol (click it to open the node and display the contents) or a – symbol (click it to fold up the node and hide the contents).

All nodes of the tree have a check-box. If you uncheck a node, it, and all the data that belongs to that node will not be imported. If you unselect the *Root* of the tree (*Demo*), nothing will be imported and you will get an error if you try to import.

Below the Root are the *Groups*. A Group is composed of (presumably) related experiments. However, there is no requirement that the members of a Group are compatible (in the sense that they can be imported into the same data file). In this case, the names of the Groups are: IV, ramps(t), Tails, Inst.IV, Gating, Sodium Ch., Inactiv., Singles, C-fiber. There is no requirement for the names of the groups to be different, and in many cases the names will repeat. If you uncheck a group, it will not be imported.

Within each Group are *Series*. A Series holds one or more *Sweeps* of data, and all the sweeps in a series are compatible (in the Spike2 sense of being readable into the same file). The name of each series indicates protocol used to generate the data. If you uncheck a series, the data for the series will not be imported. In the example, there are visible series with the names: IV, i_IV, Hinf and Thesit.

By default, all data in the tree is selected for import. However, having items selected does not guarantee that they are imported. When you close the dialog to import the data, the importer searches the tree to find the first Series that is selected for import. This generates the template that is used to create an output file. The importer then scans the tree of selected items and unselects any Series that is not compatible with the template. Currently, being compatible means:

1. The Series has the same number of data channels as the template.

2. The channels have the same type as the template.

3. The channels have the same sample rate as the channels in the template.

A series does not have to have the same number of data points as the template (and some series may have a channel with no data points). In the future, we may allow more matching conditions to be applied (such as matching the Series or Group name).

## Gap-free
Normally, every sweep of data is imported as a separate chunk of data and the Marker channel is used to indicate the start and end of each sweep. If you check the Gap-free check box, the data is imported as if it was sampled in one continuous section with no indication of where each internal sweep started or ended. This is not usually what you want.

## Group and Series TextMark channels
In addition to the waveform data channels found in the file, we add two additional Marker channels. These are added to the file each time a sweep is imported when the Group or Series in the original data changes. Data is added even if the Group or Series has the same name as the previous one. The importer also keeps track of the Group and Series name and allocates them codes, starting at 1 based on the order in which they are discovered in the original file (without any regard to which check boxes are clicked). These codes are given to the Marker items to allow you to identify them. If you are running the importer from a script, you can pre-set the identifiers using the GROUP= and SERIES= values in the FileConvert$() cmd$ argument.

## FileConvert$() command cmd$ argument
From the script language you can preset the codes associated with group or series names. The items are added before scanning the data file for names, so these script-added items get codes starting from 1. For example setting

the `cmd$` argument to `"GROUP=Gr1,Gr2,Gr3;SERIES=S1,S2,S3,Series4"` will pre-set three group names (note that name matching is case insensitive), and 4 series names.

### New features at Signal version 7.01

Before version 7.01, the importer would import one sweep of data from the first selected item in the tree. It was also limited to files with a maximum of 20 Groups and 100 Series within a group. It did not import the names of the Groups and Series as Marker data.

# Text importer

When a user imports a text file interactively, the last user configuration is saved as an XML format file called `Ascii_Last.icfx` in the current users application data path. If you rename this (in the same place) to `Ascii_Def.icfx`, then this will always be used to set the initial configuration unless you use the `Conf` option (described below). You can save configuration files with your choice of name from the text import dialog that appears when you use the text importer interactively.

Configuration files are applied after doing an initial scan of the file. They can only make changes that are compatible with the data that is discovered by the initial scan; they cannot create additional channels.

The text importer has its own interactive Help file with more information about supported text file formats. If your file holds a header followed by a rectangular table of columns, each column representing equally spaced data (waveforms), then you should have little difficulty importing it. You will need to refer to the specific importer help to cope with columns containing event times or TextMark or RealMark data.

The text importer supports the following names in the `cmd$` string for the `FileConvert$()` command.

| name | value |
|------|-------|
| Conf | Configuration file name (including the `.icfx` extension). If this string is not empty then the provided file name including its full path will be used to load the configuration file. This keyword is always applied first, regardless of its position in the `cmd$` string. |
| Find | In cases where the text holds data blocks of different types you can change how the initial scan chooses the block that is most typical of the data, which determines the number of columns. You can set the values: |

0  This is the default. The block with the largest number of numeric items (columns-first numeric column)*lines.
1  The block with the largest number of items (columns * lines).
2  The block with the most lines.
3  The block with the most columns.

### What to do if the importer does not recognise your file

If your text data does not meet the rules for import by the importer, you could consider writing a script to do the task. This is usually not very difficult (as long as you can use the script language); the complications of the general text importer come from the fact that it has to cope with a wide variety of data formats.

Files you want to import will usually be in a fixed format that you know well. You will usually have multiple columns of data and possibly a file header. The commands you will need to use will include:

`FileOpen`  You will want to open files of type 8 (an external text file with no associated window).

`FileNew`  To create a new data file of type 7.

`FileSaveAs` To save the resulting data file; use type -1.

ReadSetup   To configure the Read command so you can easily split the lines in the file into columns

Read        To collect a single line of input and convert it into variables. If you need to inspect the lines firs, you
            may need to use `Read()` to collect an entire line, then use `ReadStr()` to sort out the data.

ReadStr     To convert a string into variables

MemChan     It is usually easier to create memory channels, write the data into them, then save the result as a
            channel on disk.

MemSetItem  Add a data item to an existing memory channel.

# XItek Neurowork importer

This importer expects to find all the files for one import in a folder. All the files (and the enclosing folder) have a
common (and somewhat long) name. This is usually of the form:

```
User-defined_xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx
```

where the `xxxx` are hexadecimal digits. There can be problems if the full path name to a file reaches 260
characters, which is not that difficult to achieve when the folder name and file name account for more than 100
characters.

A typical set of files in a folder could be:

```
User-defined_xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx.eeg
User-defined_xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx.ent
User-defined_xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx.epo
User-defined_xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx.erd
User-defined_xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx.etc
User-defined_xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx.snc
User-defined_xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx.stc
User-defined_xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx_001.erd
User-defined_xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx_001.etc
User-defined_xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx_002.erd
User-defined_xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx_002.etc
User-defined_xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx_003.erd
User-defined_xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx_003.etc
```

The `.erd` and `.etc` pairs hold the raw data and there can be many pairs of these files. The importer looks for the
`.stc` file as the key to the import process as this file holds the information to tie the experiment together. For our
importer to work, the file names must all be correct and match the `.stc` file name (but we do not care if the folder
name is different). We import all the raw waveforms, an event channel of Trigger data, and a channel of TextMark
data indicating the original raw data file starts.

In addition to importing data from the `.erd`/`.etc` files, we also parse the `.ent` files, which hold descriptions of
montages and user comments and automated comments. These comments are imported as a TextMark channel.
Each comment in this file has a type (Comment/Data/Type), and we use the type to set the first marker code of the
TextMark. We observe that user-entered comments seem to be given a type of "Custom", and these are given type
code 00. Any other types seen are given sequential type codes from 00, in the order that we encounter them in the
file. We can assign additional fixed codes to other types, if required. There is an additional field (Comment/Origin)
that we have observed set to "Acquisition" or "Review". If this is "Review", we set the second type code to 01,
otherwise it is 00.

# Import Op Cl

Signal can import idealised traces from the CED Patch software for analysis, these are files of the type `*.res` or
`*.r??` generated by the Pulsed Analysis or Continuous Analysis programs. You can also import idealised traces
from David Colquhoun's SCAN analysis software, these are files of type `*.scn`. To use this feature, first open the
relevant cfs data file, then import the idealised trace data – the trace data associated with the current data file will
be used. Once imported the idealised trace will be stored in the resource file associated with the data file (i.e. the
`*.sgrx` file) and can be processed using the Analysis menu (see *Single channel analysis* for more details).

# Global Resources

If the current view is not a data or XY view, this command appears in the menu, otherwise it appears in the Resource Files popup menu.

Normally, each Signal data and XY file has an associated resource file with the same name and `.sgrx` extension (in older versions of Signal the `.sgr` file extension was used). These per-file resources remember the screen layout, cursor positions, active cursor parameters and other settings. They allow each file to open with exactly the same screen appearance it had when it closed at the cost of an extra file on disk.

However, per-file resources do not let you use the same display and cursor settings for a sequence of files. If you enable global resource files, you can use a single resource file (or one resource per folder) to control the screen appearance of multiple data files. This is particularly powerful when you review data stored on a read-only device such as a DVD drive.

Unlike the per-file resource files, which are updated automatically whenever you close a resource file, global resource files are never updated automatically. Use Update Global Resource to update the current global resource and Save Resources As to create a new resource file with the current settings for the current view. The equivalent script function is `FileGlobalResource()`.

The use of global resources is managed from the Global file resources dialog, which has these fields:

### Use global resource files
Check this box to use global resources. If this is unchecked, no global resource files are used and each data file has its own resource file. Please remember that using this dialog makes no difference to the resources used by any open data view. It changes the resource file that is used when a saved file view opens. If you check this box and no global resource file is found, Signal behaves as if the box was not checked.

### Name of file
This field sets the name of the resource file. The name should not include a path or the `.sgrx` file extension. These are added automatically, as required. If this field is blank, no resource file is used.

### File location
This field sets where to search for the global resource file. You have three choices: *System folder list only*, *Data file folder then System list*, *Data file folder only*. The System folder list is searched in sequence, stopping at the first folder holding the resource file - at each location Signal looks first for a new file with the `.sgrx` file name extension and then for an old file with the `.sgr` extension - so an old file earlier in the list 'beats' a new file at a later location. The list used starts with the `Signal7` folder inside `My Documents`, next the `Signal7Shared` folder inside the documents folder for all users and finally the Signal installation folder (wherever the `cfsview.exe` application file is located). The data file folder is the folder from which the data file is opened.

### Only use global resource file if
If you do not check any boxes in this area of the dialog, global resource files are applied to all data files. This dialog area allows you to restrict the files that use the global resource file in place of the per-file resources.

### Data file is within the path shown below
If you set a path and check this box, only files that lie within this path are considered for global resources. For example, if you only wanted to apply global resources to files read from your DVD drive, you might set this to `E:\` (if `E:\` is the path to your DVD).

### Data file does not have its own resource file
Check this box if you would prefer to use the per-file resources if a resource file with the correct name and in the same folder as the data file exists.

# Resource Files

This menu item replaces Global Resources when the current view is a data or XY view. It opens a pop-up menu from which you can select Global Resources, Apply Resource File, Save Resources As and Update Global Resource.

### Apply Resource File

This command applies a user-chosen resource file to the current data or XY view. You can use this to apply a complex window arrangement or active cursor measurement to multiple files. The script language equivalent is `FileApplyResource()`. You can also use the Global Resources command to automatically apply a specific resource file.

### Save Resources As

This command saves the resources associated with the current data or XY view to a resource file, it opens a file save dialog for you to set the file name. The script language equivalent is `FileSaveResource()`.

### Update Global Resource

This command is enabled for the current data or XY view if global resources are enabled and a global resource file name is set. It updates the global resource file with the current view settings. If the global resource file does not exist, one is created in the folder set by the Global Resources dialog (or in the application folder if both the data file folder and the application folder are allowed).

# Close

This command is used to close the active window. It is equivalent to double-clicking the control menu icon at the top left of the window (in windows that have one) or pressing the right-hand top corner **X** button. If you use this command on a new memory or XY view, a newly sampled data document or a text-based view with text that has not been saved, a dialog will ask if you wish to save the text before closing the window. This behaviour can be disabled for memory and XY views using an option in the Data page of the Preferences dialog.

# Close All

This command closes the current window and all windows associated with the file. In addition to saving the state of the data file in a `.sgrx` resource file, Signal offers to save the state and contents of any memory view windows that belong to the data file. Next time you open the data file, all the data view windows and their contents will be restored. If you open saved memory view data then that will be opened as a file view and restored to its previous state as well.

# Save, Save As

Save will save the current document under its current name, unless it is unnamed, in which case you are prompted for a name before it is saved. Save As is used to save the document with a different name, leaving the original file intact. The Save command is not available for a data document unless the data has been changed since the last save. If you save a data document using the same name as an existing data file the existing data file will be overwritten, but it will be moved to the recycle bin to help you avoid losing important data by accident.

### Data documents

Data files are kept on disk, not in memory, as they can be very large. Changes made to a data file are permanent as they are made on disk. When you save newly sampled data, you are giving the data file on disk a name (replacing a temporary name). If you save it to a different drive from that set in the Sampling page in the Preferences dialog, Signal copies the file to the new drive and deletes the original. If the file is large this operation can take a noticeable time.

When you are working with a file view, the current frame for the view is held in memory and will be discarded when the view switches to a different frame. Any changes made to the frame data while it is held in memory must be saved before the new frame is loaded or the changes will be lost. You can write the changed data back into the

file using the Save command. The Preferences dialog allows you to select what happens if the frame is changed while data is unsaved; the default action is to query the user. Changes made to non-channel frame data (such as the frame state or flags) are always saved. Memory views hold all frames in memory and do not save changes until the entire document is saved.

### Other documents

Text, script and XY documents are held in memory. Changes made to them are not permanent until the document is saved to disk.

# Export As

This menu item, available only when the current window is a data or XY view, is used to save some or all of the view data to a new file in one of a number of formats. The dialog prompts you to choose a file name for the output, and lets you select the output format. You can choose between: Data file (*.cfs) to export as a new Signal data file, Text file (*.txt), Metafile (*.wmf, *.emf) for a scaleable image and Bitmap file (*.bmp), JPEG file (*.jpg), PNG file (*.png) or TIFF file (*.tif) for a copy of the screen rectangle containing the window. More file types will be listed if you have installed external exporters. Select one of the formats and set a file name, then use the Save button.
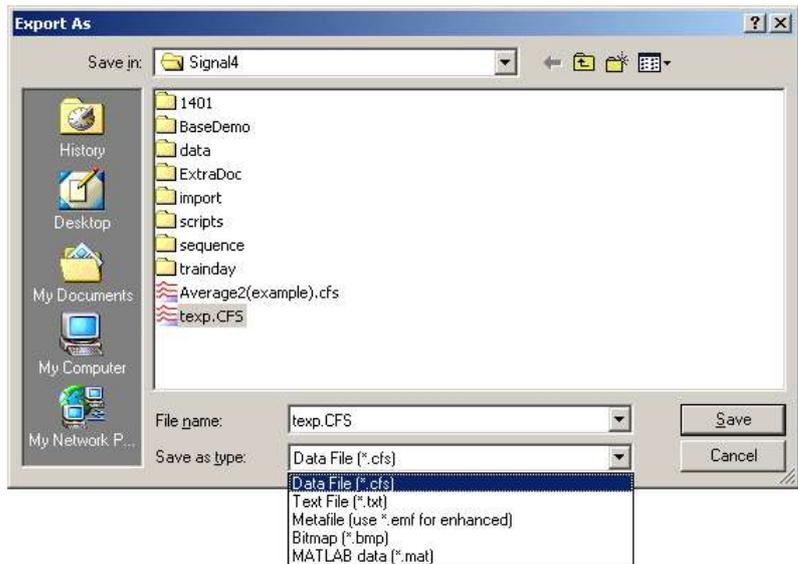
### Data file

This option, available for data views only, opens a dialog in which you can create a new data file from a time range and selected channels and frames of the current file. The CFS file format is currently only able to hold waveform and marker data, so any idealised trace and real marker channels selected will not be exported.
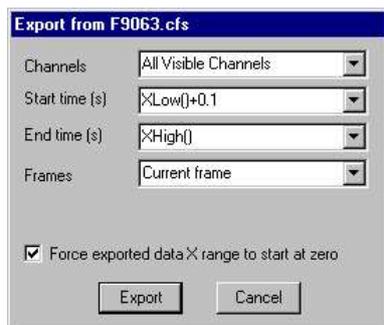
Use the dialog to select the channels, time range within the frames, and the frames to be exported. You can choose all the channels or all visible channels, individual channels, selected channels or enter a list of channel numbers directly. You can export all frames, the current frame, all tagged or un-tagged frames, frames with a given state, or you can enter a list of frames. The check box Force exported data X range to start at zero shifts the times of data points in the new file so that the time for the first point in each frame is forced to be zero.

Once you have selected the channels, frames and a time range, click the Export button to write the data to the new data file or Cancel to quit.

### XY file

This option, available for XY views only, saves all the XY view data to a new file.

### Text file

This is the same as the Edit menu Copy As Text command, but with the output sent to a text file and not the clipboard. See the Copy As Text command for details of the dialogs and text format that will be used.

### Metafile

This copies the view window as a Windows Metafile or enhanced Metafile, use the wmf file extension for a normal metafile and emf for an enhanced metafile. This file format can be scaled without losing any resolution and is usually the preferred format for moving Signal images to drawing programs or into reports. Note however that

some drawing programs can have problems using metafiles, particularly if you are using XY views with channels that are drawn filled.

### Bitmap file, JPEG image, Portable Network Graphic file, Tagged Image Format file

These options copy the screen area containing the window to an image file; the result is a copy at the screen resolution. If you need to scale the image, or want to edit it, a Metafile copy is often better. The Portable Network Graphic format will usually be the smallest on disk unless the screen contains a real-word image, when JPEG may be smaller (but at the cost of artefacts around axis lines and text). Bitmap images will usually be the largest.

### External exporters

These use the same dialog as for exporting to a data file to select the channels, frames and time range. There may be additional dialogs depending on the target format. See below for additional information on the external exporters. The only external exporter currently defined is for MatLab files.

# MATLAB external exporter

You can export data and XY view data to MATLAB by selecting MATLAB data (*.mat) as the Save as type from the File menu Export As dialog. To enable this functionality, you must select the MATLAB file export option when installing Signal. You do not need a copy of MATLAB to be installed on your computer. You can also carry out and control export to MATLAB format from a script.

Once you have selected MATLAB data, set a file name and click on Save in the Export As dialog. Signal will then proceed to dialogs where you choose the data to export and the format to export it in. The details of the following dialogs depends on the source view type.

If you have a problem getting the export to work, check that the file `MatExp.sxl` is present in the `Signal7\Export` folder. A whole bunch of other files are also needed in this folder, but if this one is there, it is likely that the rest have made it too. If the file is not present, you need to reinstall Signal and make sure that you select the option to install MATLAB file export support.

Mat-files represent a collection of workspace variables inside MATLAB. Signal creates mat-files containing one variable holding aggregated data for all waveform channels and separate variables for each other data channel that is exported. The variables are structures containing multiple fields that hold the channel data.

### Possible interaction with MatLabOpen() script function

Various users have reported that exporting data to MatLab files works normally but fails after the `MatLabOpen()` script command has been used. This appears to be caused by DLL version incompatibilities between the MatLab DLLs installed with Signal for the purposes of file export (which should be available even if MatLab is not installed) and the DLLs installed with MatLab (which are loaded by `MatLabOpen()`). It appears that the problem can be fixed, at least in some cases, by hiding the MatLab DLLs installed with Signal so that file export uses the DLLs installed with MatLab. The DLLs in question are all in the export directory inside the Signal export directory, they are: libmat.dll, libmx.dll, libut.dll, libz.dll, icudt32.dll, icuin32.dll, icuio32.dll, icuuc32.dll, msvcr71.dll and msvcp71.dll.

# Workspace variable naming

The workspace variables in a mat-file must have names that are both unique and legal. If a variable name is not unique it will overwrite the existing workspace data with the same name. MATLAB cannot read variables with illegal names. Signal mat-file export creates variables with names based upon the source view file name and the channel name. You can independently select if either of these is to be used. This produces variable names that are both useful and unique. The default setting is to use the source view file name but not the channel name. The variable name settings are used to build the variable name as follows:

If the source view name is to be used, it is placed at the start of the variable name followed by an underscore character '_'. Following this a channel identifier is added. The identifier is always '`wave_data`' for the aggregated waveform channels, for other channels it is either the channel title (if using the source channel name) or Signal builds a name using '`Ch`' followed by the channel number. For example, when exporting a view called

`Expt1` containing 2 waveform channels and a keyboard marker channel called 'Keyboard' we would get the following MATLAB variable names:

| | |
|---|---|
| Using source view and channel name: | `Expt1_wave_data, Expt_Keyboard` |
| Using source view name only: | `Expt1_wave_data, Expt1_Ch3` |
| Using channel name only: | `wave_data, Keyboard` |
| Using neither name: | `wave_data, Ch3` |

Having generated a name according to these rules, Signal then checks and, if necessary, modifies the name to guarantee that it is legal. The rules for a legal name are that it must not be more than 63 characters, must begin with an alphabetic character and must only contain alphanumeric characters and the underscore character '_'. Signal modifies the name by appending a 'V' to the start of the name if it does not begin with an alphabetic character, converting all illegal characters to underscores and finally truncating if more than 63 characters long.

If despite all this you end up with variable names that are the same, the variables will overwrite each other when you read the file into MATLAB and you will only see the last read variable.

# Data view data

When exporting data from a file or memory view you will first get a dialog that allows you to select the channels, frames and time range that you want to export. There is also a check box that selects offsetting the start of the time range being exported to zero. This dialog is the same as the dialog used to select data to export to a CFS file.

When you have selected your channels, frames and time range you are then presented with a second dialog that controls mat-file export options. These options allow you to define how the MATLAB workspace variables will be named, what data is generated for different channel types and the format used for various types of Signal data.

*Compatibility*
If you are using an older version of MATLAB (currentlybefore version 7.3) or want your exported data to be usable with earlier versions of MATLAB, you must use the compatibility field to select a suitable output format.

*Use ... in variable names*
These two options in the dialog control how the mat-file variable names are constructed as described previously.

*Waveform options*
These two controls specify how waveform data is handled. The Layout options item can be set to *Waveform only* or *Waveform and errors*, if the second option is selected and there are error values available these will be exported to a separate array. The Waveform data as item sets the type of data created for waveforms, it can be set to *Float (32-bit)* or *Double (64-bit)*. Floats use less memory while doubles are the most generally useful in MATLAB, it is probably best to leave this set to doubles unless you are suffering from memory problems with large amounts of data.

# XY view data

For an XY view you do not get a dialog to control what data is exported; all the visible channels are exported and only data points that lie within the displayed X and Y axis ranges are used. A MATLAB variable is still created for visible channels with no visible data.

A dialog is provided to control how the data is exported. This is a simplified version of the data view export options dialog; it only contains the compatibility selector and the two check boxes to select the use of the source view name and channel names to create the MATLAB variable names.

# Mat-file data format

A mat-file represents a collection of MATLAB variables rather than simple data values which allows for a complex representation of Signal data within MATLAB. Each Signal channel exported into a mat-file is represented by one variable, except for waveform channels which are grouped together into a single variable. The variables are structures containing fields that hold information about the channel and other fields holding the data. For example, a structure holding data from waveform channels has the fields `xlabel`, `interval` and `start` each of which is a simple (scalar) datum holding the x axis title, the sample interval in seconds and the time of the first data point. In addition there is a field called `wave_data` that is a 3D matrix holding the waveform data values and an optional field called `SD` which is a 3D matrix holding the standard deviation for the corresponding waveform points. The structure varies according to the channel type, though some fields are common to all channels.

# Waveform data

Waveform data is exported as a 3-dimensional array `<points x chans x frames>` of waveform values with an optional associated array of error values. All waveform channels are exported together in the same variable. There are separate arrays of structures holding information about the channels and frames that are exported. The fields in the waveform data structure are:

| | |
|---|---|
| `xlabel` | a string holding the X axis label, normally blank. |
| `xunits` | a string holding the X axis units, this is normally '`s`' even if you have opted to display time as milliseconds or microseconds. |
| `start` | a double holding the start time, in X axis units, for the waveform data. |
| `interval` | a double holding the waveform data sample interval, in X axis units. |
| `points` | an integer holding the maximum number of waveform data points exported for each channel for each frame. The actual number of points exported for each frame can vary if the source data has varying points per frame. |
| `chans` | an integer holding the number of waveform channels exported. |
| `frames` | an integer holding the number of frames exported. |
| `chaninfo` | an array of structures, length `chans`, holding information on the channels exported. Each structure contains the following items: |
| | `number` an integer holding the actual source file channel number. |
| | `title`  a string holding the channel title. |
| | `units`  a string holding the channel units. |
| `frameinfo` | an array of structures, length `frames`, holding information on the frames exported. Each structure contains the following items: |
| | `number` an integer holding the actual source file frame number. |
| | `points` an integer holding the number of data points for each channel in this frame. |
| | `start`  a double holding the start time, in seconds, for the frame. |
| | `state`  an integer holding the frame state code. |
| | `tag`    an integer holding the frame tag (0 or 1). |
| | `sweeps` an integer holding the frame sweep count. This will be zero for frames not produced by averaging. |
| | `label`  a string holding the label used for the frame state. |
| `values` | an array with dimensions `<points x chans x frames>` holding the waveform values. Depending upon the output options selected, this array could be double or single-precision real values. |
| `SD` | an optional array with dimensions `<points x chans x frames>` holding the waveform error values as standard deviations. Depending upon the output options selected, this array could be double or single-precision real values. |

# Marker channels

Marker channel data is represented as a 2-dimensional array of marker times and a corresponding 2-dimensional array of marker code values. Each marker channel is exported to a separate variable in the MATLAB workspace. The fields in the channel structure are:

| | |
|---|---|
| title | a string holding the channel title. |
| units | a string holding the channel units, normally blank. |
| resolution | a double holding the underlying timing resolution in seconds. |
| points | an integer holding the maximum number of markers in a frame. |
| frames | an integer holding the number of frames exported. |
| times | a <points *x* frames> 2-dimensional array of doubles holding the marker times in seconds. Times for markers not present in a frame are set to zero. |
| codes | a <points *x* frames> 2-dimensional array of integer values holding the marker codes. |

# Real marker channels

Real marker channel data is represented as a 2-dimensional array of marker times, a corresponding 2-dimensional array of marker code values, and a 3-dimensional array of marker floating point values. Each real marker channel is exported to a separate variable in the MATLAB workspace. The fields in the channel structure are:

| | |
|---|---|
| title | a string holding the channel title. |
| units | a string holding the channel units, normally blank. |
| resolution | a double holding the underlying timing resolution in seconds. |
| points | an integer holding the maximum number of markers in a frame. |
| reals | an integer holding the number of floating-point values attached to each marker. |
| frames | an integer holding the number of frames exported. |
| times | a <points x frames> 2-dimensional array of doubles holding the marker times in seconds. Times for markers not present in a frame are set to zero. |
| codes | a <points x frames> 2-dimensional array of integer values holding the marker codes. |
| values | a <reals x points x frames> 3-dimensional array of double values holding the marker real values. |

# Idealised trace channels

Idealised trace channel data is represented as a 2 dimensional array of structures, with each structure holding information on one trace segment. Each idealised trace channel is exported to a separate variable in the MATLAB workspace. The fields in the channel structure are:

| | |
|---|---|
| title | a string holding the channel title. |
| units | a string holding the channel units. |
| points | an integer holding the maximum number of trace segments in a frame. |
| frames | an integer holding the number of frames exported. |
| values | a <points *x* frames> 2-dimensional array of structures holding the trace segment information. Each structure contains the following items: |

| | |
|---|---|
| start | a double holding the start time of a segment. |
| period | a double holding the length of a segment. This will be zero for segments not in this frame. |
| amplitude | a double holding the level of a segment. |
| flags | an integer holding the flags set for a segment. |
| baseline | a double holding the baseline level for a segment. |
| level | an integer holding the level of a segment. This is zero for a closed segment, 1 to n for open levels. |

# XY channels

XY view channel data is represented as two 1-dimensional arrays holding X and Y data plus ancillary information. Each XY view channel is exported to a separate variable in the MATLAB workspace. The fields in the channel structure are:

| | |
|---|---|
| `title` | a string holding the channel title. |
| `yunits` | a string holding the Y value units. |
| `xunits` | a string holding the X value units. |
| `points` | an integer holding the number of XY points. |
| `xvalues` | a <`points` *x* 1> array of doubles holding the x values of the data. |
| `yvalues` | a <`points` *x* 1> array of doubles holding the y values of the data. |

# Script export to mat-files

Export to mat-files is also available from the script language by using the `FileExportAs()` function with the file type set to 100. With this type in use, the file name selection dialog (if provided) will use a '*.mat' filename filter, an extra `exp$` argument becomes available to allow MATLAB-export specific options to be set and the time range, frames and channels to be exported are set by `ExportTimeRange()`, `ExportFrameList()` and `ExportChanList()`. The extra `exp$` argument that sets options is a string of the form:

`name=value|name=value|…|name=value`

where `name` specifies some export option and `value` sets it's value. You can include as many options as you want, options that you omit are set to the default value. Option names are not case-sensitive. It is not an error to use an unknown option.

### *Data view export options*

| | |
|---|---|
| `UseSName` | selects use of the source name in the channel variable name. Set to 1 if you want to use the source name, 0 if not. The default is 1. |
| `UseCName` | selects use of the channel name in the channel variable name. Set to 1 if you want to use the channel name, 0 if not. The default is 0. |
| `WaveOpts` | selects generation of an array of waveform error values in the workspace variable, if errors are available. Set to 1 if you want errors, 0 if not. The default is 0. |
| `WaveData` | selects the sort of data generated for waveform channels. Set to 1 for single-precision reals and 2 for double-precision reals. The default is 2. |
| `Compat` | sets the compatibility option. Set to 0 for the most recent MATLAB file version (currently 7.3), 1 for version 4 or earlier, 2 for version 6, 3 for version 7 and 4 for 7.3 or later. The default is 0. |

### *XY view export options*

| | |
|---|---|
| `UseSName` | selects use of the source name in the channel variable name. Set to 1 if you want to use the source name, 0 if not. The default is 1. |
| `UseCName` | selects use of the channel name in the channel variable name. Set to 1 if you want to use the channel name, 0 if not. The default is 0. |
| `Compat` | sets the compatibility option. Set to 0 for the most recent MATLAB file version (currently 7.3), 1 for version 4 or earlier, 2 for version 6, 3 for version 7 and 4 for 7.3 or later. The default is 0. |

# Backup sgrx file

You can use this command with any type of data file. Information relating to the current file is stored in an associated `.sgxr` resource file, this includes things like the draw modes of the channels but more significantly it contains any idealised trace data associated with the file. In a lengthy idealised trace editing session you may wish to backup the `.sgrx` file to ensure edits are not lost. This option creates a new file called `filename.backup.sgrx` where filename is the stem of the original data file name, this file will contain all of the idealised trace data at the time of backup. In the unlikely event of needing to retrieve the backed-up `.sgrx` file you should first make sure that the original data file is not open and then copy the backup `.sgrx` file to `filename.sgrx`, it will be automatically picked up & opened when the data file is opened.

# Revert to Saved

You can use this command with a text file, sequencer file or a script file. The file changes back to the state it was in at the time of the last save to disk and any changes made since then are lost, which can be useful or might be a disaster - so use this command with care.

# Data update mode

This command controls how and if changed file view data is written back to the CFS data file. Signal holds the data for the current frame in memory where it can be accessed and modified by script commands or by the channel data manipulation commands. When the file is closed, or the view switches to another frame, the data update mode determines what happens if there is changed data in memory. Changes to the frame data can be written back to the disk, or they can be discarded, or the user can be asked to choose what is to be done. The Data page in the Preferences dialog sets the default data update mode for all files, this command changes the mode for the file attached to the current view.

If Query the user mode is selected, the Save changes dialog will appear when required. This allows changed frame data to be interactively discarded or saved, check the Adjust data file update mode to match check box if you don't want to see the dialog again for this file.
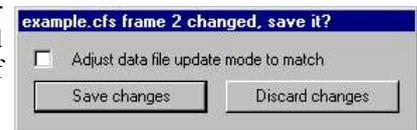
# Send Mail

If your system has support for email installed, you can use this command to send any document from Signal to another linked computer. The command vanishes if you do not have mail support on your PC. Text-based documents, XY views and memory views can be sent even if they have not been saved to disk (Signal writes them to a temporary file if they have not been saved). Signal data files can be sent as long as they have already been saved to disk.

# Load and Save Configuration As

These commands manage Signal configuration files. These hold the sampling parameters, the window arrangement required during sampling, the output setup and the types of on-line analysis required. Signal always has one configuration loaded, this is the configuration used for sampling and the sampling configuration dialogs.

Versions of Signal before 5.08 saved sampling configurations in files with a `.sgc` filename extension. Later versions of Signal save sampling configurations in XML files with a `.sgcx` filename extension. Signal is able to read both types of sampling configuration but it will only save sampling configurations as new-style XML files.

The Load Configuration and Save Configuration As commands transfer the Signal configuration between disk file and the application. They both open an appropriate file dialog to select a file for loading or saving to.

The Save Default Configuration command saves the current configuration as the default configuration; the one that will be automatically loaded whenever Signal is started.

### Default configuration files: default.sgcx, last.sgcx

If the configuration file `default.sgcx` is found, it is always loaded when Signal starts. To save the current sampling configuration as `default.sgcx` use the Save Default Configuration command. The standard file `last.sgcx` holds the last configuration that was used for sampling. If `default.sgcx` cannot be found, and `last.sgcx` exists, `last.sgcx` is loaded. Signal updates `last.sgcx` with the current sampling configuration each time sampling stops.

*Where does Signal search for these files?*

Now that we offer the option of installing Signal in the Program Files folder, the places where modifiable files can be stored has changed. As we want to remain compatible with the old system, we have to search a list of places to find the `default` and `last` files (the `default` file is searched-for first and if it is not found Signal searches for the `last` file). The places we search are:

1. The application data directory (but read below for what this means)

2. If the file is not found, then try the directory in which Signal is installed

In each directory we search for the `sgcx` file first, then for the `sgc` file (for backwards compatibility).

*Application data directory*
The application data directory is the first of the following that exists:

1. The current users application data directory
2. If that directory does not exist, the all-users application data directory
3. Then if that directory does not exist, the current user's "My Documents" folder

# Page Setup

This opens the printer page setup dialog. The dialog varies between operating systems and printers. See your operating system documentation for more information. The important options that are always present include the paper orientation (portrait or landscape), the paper source (if your printer has a choice), the printer margins and the choice of printer.

The Orientation option (portrait or landscape) applies to all output except the Print Screen command, which has it's own selector for the output orientation.

The printer margins will appear in inches or millimetres, depending upon the locale set for your computer. These margins are used for all printed output. The left and right margins are applied to everything including headers and footers. The top and bottom margins apply to everything except headers and footers which have their own top and bottom margin settings (see the Page Headers command). The top and bottom margins set here are further modified if a header or footer would collide with them.

Most printers have an unprintable area near the paper edge. If you set margins that are smaller than this unprintable area, the margins are increased so that all output is visible. If you set margins that reduce the printable area too much, the margins are ignored.

### Registry use
Signal saves the printer margins in units of 0.01 mm in the system registry. You can find them in the `HKEY_CURRENT_USER\Software\CED\Signal\PageSetup` folder as `REG_DWORD` values: `LeftMargin`, `RightMargin`, `TopMargin` and `BottomMargin`.

# Page Headers

You can apply headers and footers to all printed output. The headers set in this dialog are used with all File, Memory, XY and text-based views. Other printed output (for example from the Print Screen command) uses all of these settings except for the text, which each command provides separately.

### Header and footer positions
The horizontal position is set by the left and right margins in the Page Setup dialog. The vertical positions are set by the space above the header and the space below the footer fields in inches or millimetres, depending on the locale. If your header or footer encroaches on the top and bottom margins set in the Page Setup dialog, the top and bottom values are adjusted to keep the output clear of the header and footer.
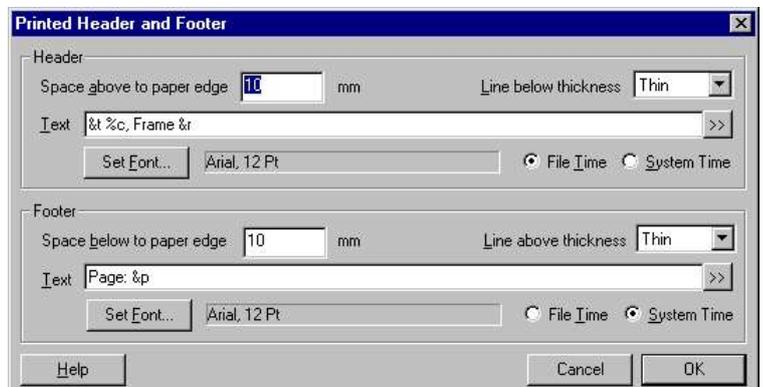
### Line thickness

You can choose between: None, Hairline, Thin, Medium and Thick. A Hairline is the thinnest line possible on the output device. The other settings should be self-explanatory. The header line is drawn below any header text, the footer line is drawn above any footer text. If there is no header or footer text, no line is drawn.

### Text

This is the text to display as the header or footer. If this field is empty, the header or footer is omitted (including any line). You can split the text into left-justified, centred and right-justified sections with the vertical bar character. You can also insert codes that are replaced by document and time information. The simplest way to do this is by clicking the **>>** button to the right of the text and choosing an option.

### Set Font

Click this button to choose a font for the header and the footer. Font sizes are limited to the range 2 to 30 points.

### File/System Time

You can insert times into both the header and the footer. However, you have to choose between the current time and the file time. This allows you to display the file time in the header and the current time in the footer, or vice versa.

### Document information

The codes shown below are replaced by the relevant document information. Except where an alternate effect is shown for the upper case code, upper and lower case are treated alike. A single ampersand ("&") character can be inserted using a double ampersand ("&&") code.

| | | | |
|---|---|---|---|
| &f | File and path | &F | Upper case file and path |
| &t | Document title | &T | Upper case document title |
| &p | Page number | &n | Total number of pages |
| &a | Absolute frame time | &s | Frame state code |
| &g | Frame flags | &c | Frame comment |
| &r | Current frame number | &l | Overdrawn frame list |

### Time and date codes

You can insert times and dates using % followed by a character code. The combination is replaced as described in the table. You can also use %#c and %#x for a long version of dates and times. You can remove leading zeros from numbers with #, for example %#j.

| | | | |
|---|---|---|---|
| %a | Short day of week (Mon) | %p | Indicator for A.M or P.M. |
| %A | Long day of week (Monday) | %S | Seconds as number (00-59) |
| %b | Short month name (Jan) | %U | Week of year, Sunday based (00-53) |
| %B | Long month name (January) | %W | Week of year, Monday based (00-53) |
| %c | Date and time for locale | %w | Sunday based weekday (0-6) |
| %d | Day of month (01-31) | %x | Date formatted for locale |
| %H | Hour in 24 hour format (00-23) | %X | Time formatted for locale |
| %I | Hour in 12 hour format (01-12) | %y | Year without century (00-99) |
| %j | Day of year (001-366) | %Y | Year with century, e.g. 2004 |
| %m | Month as number (01-12) | %z/Z | Time zone name |
| %M | Minute as number (00-59) | %% | The percent sign |

**Registry use**

Signal saves the header and footer settings in the system registry. You can find them in the `HKEY_CURRENT_USER\Software\CED\Signal\PageSetup` folder as strings: `Header`, `Header info`, `Footer` and `Footer info`. The `Header` and `Footer` items hold the text strings. The two `info` items are strings that code up the font name, point size, bold and italic settings, distance to the paper edge in units of 0.01 mm, line thickness (0=none, 1=Hairline, 2=Thin, 3=Medium, 4=Thick), and File time (0) or System time (1).

# Print Preview

This option displays the current window as it would be printed by the Print option. You can preview file, memory, XY and text based windows. You can zoom in and out, view single or two pages, step through pages of multi-page documents and print the entire document using a toolbar at the top of the screen. Use the Close button to leave this mode without printing.

In previous version of Signal, the print preview window took over the entire program and no other commands were available apart from those on the preview toolbar. From Signal version 4, the preview takes place inside the frame of the data or XY view. You can now access the File menu to change the headers and footers and print margins while within preview. If you use the menus to make changes to the



contents of the displayed data, the screen may not display the changes until you exit from Print Preview mode.
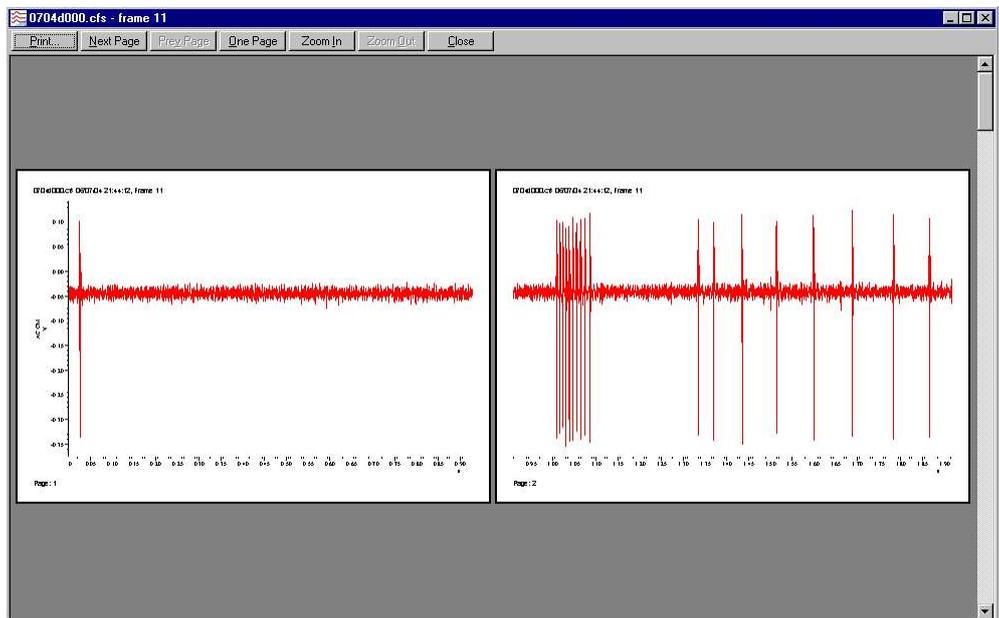
# Print visible, Print and Print selection

These commands print data views, XY views and text-based windows. The scroll bar at the bottom of the window is not printed. Print selection prints the selected area of a cursor or text window. Print visible prints the visible data in the current window. Print or its `Ctrl+P` shortcut prints a specified region of a data view at the scaling in the window (one page of paper holds the same x axis range as the current display, the output spans as many printer pages as are required to show the data selected). You must set the print range in a dialog, either by typing the start and end times directly, or by selecting them from the pop-up menus.

To print an entire frame, move to the frame required, set the visible width of the view to the x axis range per page required in the print, then select Print. Select Mintime() for the start position and Maxtime() for the end position. Print doesn't print multiple frames, this may be provided in later versions of Signal.

During the print operation (which can take some time, particularly if you have selected a lot of data) a dialog box appears. If your output spans several pages, the dialog box indicates the number of pages, and the current page so you can gauge progress. If you decide that you didn't want to print, click the Cancel button.
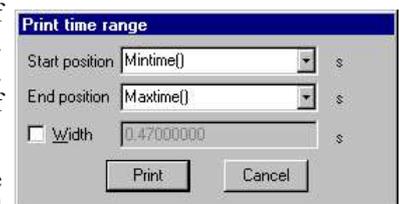
# Print screen

The Print Screen command prints all file, memory, XY and text based views to one printer page. The views are scaled to occupy the same proportional positions on the printed page as they do on the screen. The page margins are those set by the Page Setup and Page Headers dialogs.

The command opens a dialog with two regions: Page and Views. In the Page region you can set a page header and footer and choose to print in Landscape or Portrait mode. The header text is printed with an underline across the width of the page. The footer text is printed with a line over it across the page width. The fonts used and line thicknesses are as set in the Page Headers dialog If the header or footer is blank, both it and the associated line are omitted.

You can divide the header and footer into a left justified, centred and right justified sections with the vertical bar character, for example: Left|Centered|Right. You can include the current date and time in the header or footer by including, for example %c, as described for the Page Headers dialog. The >> button can be used to insert the time and date and vertical bar separators into the header and footer.

In the Views region you can choose to print view titles and draw a box round each view. You can also ask Signal to attempt to preserve the aspect ratio of characters in text windows. Without this, characters are scaled in an attempt to match the printed output to the text displayed in the view. However, this may not produce a very legible output.
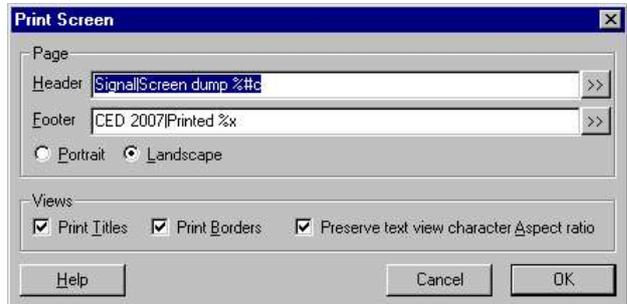
### Registry use

Signal saves the Print Screen settings in the system registry. You can find them in the `HKEY_CURRENT_USER\Software\CED\Signal\PageSetup` folder. The text strings are saved as `PSHeader` and `PSFooter`. The remaining values are saved as `REG_DWORD` values of 0 (not selected) and 1(selected): `PSViewTitle`, `PSBorder`, `PSScaleText` and `PSLandscape`.

# Exit

This command will close all open files and exit from Signal. If there are any data documents or text-based files open containing changes that have not been saved, you will be prompted to save them before the application terminates.

# Edit menu

This menu holds the standard Edit functions that all programs provide. The majority of the menu is associated with commands that move data to and from the clipboard and others that configure facilities for text-type views.

## Undo and Redo

In a text, script or output sequence window this is used to undo or redo the last text edit operation. You cannot undo operations that have been saved to disk or text operations that were done by a script.

In File, Memory and XY windows, you can undo most operations that change the appearance of the window. Cursor movements do not undo; this is by design so you can zoom in, adjust a cursor, then undo to zoom out, leaving the cursor adjusted.

## Cut

You can cut selected portions of editable text to the clipboard from any position in Signal where the text pointer is visible. This includes any text or numeric fields in dialogs and all text-based views. You cannot use this command in Signal data document windows.

## Copy

You can copy selected portions of editable text to the clipboard. If you use this command from a data document window, the window contents, less the scroll bar, are copied to the clipboard as text, a bitmap and also as a metafile. It is also copied as binary data that can be pasted into an XY view or the pulses dialog. See also Copy As Text.

### Text output
The visible data is copied to the clipboard as text. The text format used is the same as was last used in Export As to a text file or the Copy as Text command. If these commands have not been used then the default settings (only copy waveform data & times, with column headings) is used. See the Copy As Text command for details of the text output format. The text format used requires that all the visible waveform channels have the same sample interval, if this is not the case then no text data will be put onto the clipboard.

### Bitmap output
Make sure that the window is completely on the screen and that it is not covered by any other window before copying. Use this option when the required image is an exact copy of the screen. If you need to scale or edit the image Metafile format is usually better.

### Metafile output
The screen display is copied to the clipboard as a Windows Metafile. You can read an exported image into a drawing program as either a bitmap or as a metafile. Metafiles are often the preferred choice as you can treat the image as lines and text for editing. Use Paste Special and select Picture in the target application to select the metafile image.

### CED binary format
The visible waveform data is copied to the clipboard as binary (numbers) using a private CED format. This binary data can be pasted into the arbitrary waveform buffer used by the pulses dialog (see *Pulse outputs during sampling*), into another data view or into an XY view. This private clipboard format is not usable by other applications. The CED private format requires that all of the visible waveform channels have the same sample interval, if this is not the case then no binary data will be put onto the clipboard.

# Copy As Text Data view

This command copies data views to the clipboard as text. Text representations of Signal data can be very large and awkward to manipulate with the clipboard; as an alternative you can write the text output to a file with the File menu Export As command. The command leads to dialogs that set the text output format and the data to copy.

# Text output configuration

This dialog sets the text output format. The first section sets an interpolation method for Waveform channels. This is because the Waveform channels will be output in columns with each row showing data sampled at a particular time. If the data is not sampled in burst mode then the data points in the file will not all be sampled at the same time. This will result in a small error as data will appear shifted slightly to bring all the points into line. This can be overcome by using interpolation to estimate where the waveform on a given channel would have been at the time a point on another channel was sampled. Linear and Cubic Spline interpolations are available.

In the next section you can choose whether to output Data, Times values (where appropriate) and Headings for each of the of Waveform, Errors, Marker, Fitted, Idealised trace and Real Marker data types by checking the relevant boxes. Error data is always the standard deviation and is not generated unless waveform data is dumped.

All output is written in fields that are either numeric or text. A text field is a sequence of characters that may include spaces. Text fields may hold numbers, but numeric fields cannot hold text.
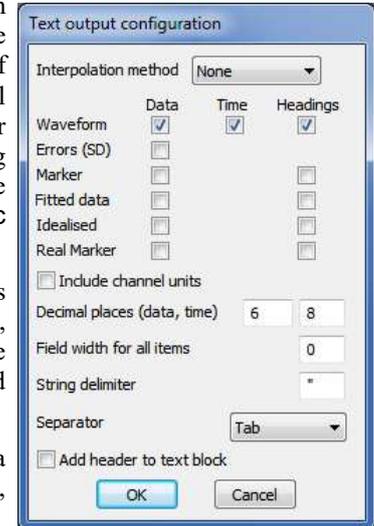
### Include channel units
If this option is checked then the headings for the Waveform channels will include a second row holding the channel units, as described below.

### Decimal places (data, time)
You can set the number of decimal places to use for both data values and times output.

### Field width for all items
This field sets the minimum width of each output field, in characters, leave this set to zero for the default field width.

### String delimiter
You can mark the start and end of a text field with special characters (usually ") so that a program reading the field can include spaces and punctuation within a field without confusion. Use this field to set a one or two character delimiter, or leave it blank. The example output below uses " as a delimiter.

### Separator
Signal outputs a separator between each field. It can be set to tab, space or comma with the Separator selector. The examples below use tab as a separator.

### Add header to text block
If you check this box, Signal outputs a header of the file name followed by the frame number and state code (with state label if available) before the text output. This header is normally disabled as it may interfere with reading exported Signal data into spreadsheets.

```
"Noisy.cfs"   "Frame 4"   "State 0"
```

### Format of generated text
All waveform data is output together, with the first column holding the time values, if enabled, and then one column per waveform channel, note that this arrangement requires that all the waveform channels have the same

sample interval. If error channels are enabled any error data for a channel is placed in a separate column after the column holding that channel's data. Waveform headings are a single line holding a title for each text column with, optionally, a second line holding channel units. Following this are multiple lines with the waveform data times and values:

```
"s"      "ADC 0"  "ADC 1"  "ADC 2"  "ADC 3"
"s"      " V"     "V"      " V"     "V"
0.23675  1.01074  4.61670  0.46875  0.23438
0.23700  1.37451  4.61182  0.44678  0.27832
...
0.33350 -1.69922 -0.18799  0.43945  0.21729
```

Each marker channel is output separately. The output starts with the channel headings, if these are enabled. The data follows in two columns: the marker time and the first marker code as a character or a number if display as a character is not possible.

```
"s"        "Keyboard"
20.20608  "a"
21.24544  "n"
```

Fitted data is output with two lines of headings, if enabled, followed by the data. The data values are the coefficients of the fitted curve.

```
"Single exponential fit on channel 1"
"Amplitude 1"  "Time constant 1"  "Offset"
0.27061    0.00575  -0.20811
```

Each idealised trace channel is output separately with two lines of heading, if enabled, followed by the data in separate columns.

```
"Open/closed times on channel 201"
"Start(ms)"  "Duration(ms)"      "Amplitude(pA)"      "Level"        "Flags"
0.00000000   0.01019555   0.196095     0      81
0.01019555   0.00141605   1.752232     1      129
0.01161160   0.00128840   0.201009     0      65
0.01290000   0.01020000   1.766908     1      129
```

Each real marker channel is output separately. The output starts with the channel headings, if these are enabled. The data follows in columns: the first two columns are the marker time and the first marker code as a character or a number if display as a character is not possible. The marker real values follow, each value in a separate column.

```
"s"     "Imemb"      "pA"
0.45957447     2      0.000000   1.000000    0.000000
0.57811550     0      0.000000   0.000000    0.000000
0.73130699     2      0.000000   0.000000    0.000000
0.78480243     0      2.000000   0.000000    0.000000
0.83708207     0      0.000000   0.000000    0.000000
```

# Copy data selection

Once you have defined the output format, you have to select the data to copy. This is done using the same dialog that is used to select the data to be exported to a CFS or text file. You can specify the channels to use, the time range for the data within each frame and the frames to be used, if you select Frames with state = xxx a separate field appears to set the state in the usual manner. The text format used requires that all of the waveform channels selected have the same sample interval, if this is not the case then no text data will be put onto the clipboard.

When you are satisfied with the selection, click on Cancel to quit or Copy to start the copy process.
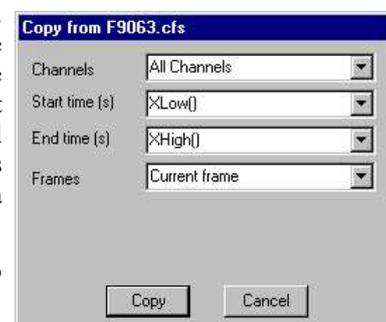
# Copy As Text XY view

This menu item is available in XY views. It copies the XY points for all visible channels to the clipboard. The first line of output for a channel holds "`Channel : cc : nn`" where `cc` is the channel number and `nn` is the number of data points. The data points are output, one per line as the X value followed by the Y value, separated by a tab character.

# Paste

You can paste text on the clipboard into any text-based document, or any text or numeric field in a dialog with the `Ctrl+V` key combination. Clipboard data using the private CED binary format can be pasted into compatible data views, an XY view or into the arbitrary waveforms output by the pulse dialog. Data is pasted into views within the displayed limits, if the clipboard holds more points or channels that the visible data, only the visible data is modified. This is not true for XY views, where all of the clipboard data is inserted as new channels. If the binary data holds fewer points or channels than the displayed data, only part of the visible data is modified. Pasting from the clipboard does not affect the data stored on the clipboard.

# Delete

This command is used to delete the current text selection, or if there is no selection, it deletes the character to the right of the text caret. Do not confuse this with Clear, which in a text field is the equivalent of Select All followed by Delete.

# Clear

When you are working with editable text, this command will delete it all. If you are in a memory data view, this command will set all the bins in all channels to zero and redraw the window contents. In an XY view whose data is created by processing, this command will remove all data points. If you are looking at frame zero in sampled data, Clear erases any overdrawn traces. Clear removes everything, Delete removes the current selection.

# Reload frame

This command is only available for File views, where it discards any changed data in the current frame and reloads the frame from disk. It is only enabled if the current frame is present in the disk file (so not for an appended frame) and the frame data or variables have been changed.

# Select All

This command is available in all text-based windows and selects all the text, usually in preparation for a copy to the Clipboard command. The short-cut keyboard command is Ctrl+A.

# Find, Find Next, Find Previous

The Edit menu Find command opens the Find Text dialog. The dialog is shared between all text-based views. It is closely linked to the Find and Replace Text dialog and shares all its fields with it; opening this dialog closes the Find and Replace dialog. Click Replace... to swap to the Find and Replace dialog. A successful search moves the text selection to the next matching string. Searches are line by line; you cannot search for text that spans more than one line.

Find Next starts a search for the text in the Find what field. The Mark All button sets a bookmark on all lines that match the search text, but does not move the selection. Searches are insensitive to the case of characters, unless you check Match Case. Check the Match whole word only box to restrict the search so that the first search character must be the start of a word and the last search character must be the end of a word.

### Search direction

Select Up to search backwards, towards the start of the text. Select Down to search forwards towards the end of the text. Select Wrap to search forwards to the end, then wrap around to the start and stop when you reach the current position. Searches do not include the currently selected text.

### Regular expression

Check this box to search for *regular expressions*. This disables Match whole word only as regular expressions have their own way to match word starts and word ends. It also disables Up searches; regular expression searches are forwards only. It enables the >> button, which displays a list of regular expressions to insert into the search string. The simplest pattern matching characters are:

^ Start-of-line marker. Must be at the start of the search text or it just matches itself. The following search text will only be matched if it is found at the start of a line.
$ End-of-line marker. Must be at the end of the search text or it just matches itself. The preceding text will only be matched if it is found at the end of a line.
. Matches any character.

To treat these special characters as normal characters with regular expressions enabled, put a backslash before them. A search for "^\^.\." would find all lines with a "^" as the first character, anything as a second character and a period as the third character.

You can use \a, \b, \f, \n, \r, \t and \v to match the ASCII characters BELL, BS (backspace), FF (form feed), LF (line feed), CR (carriage return), TAB and VT (vertical tab). Searching for \n and \r will not normally match anything as \n or \r\n mark line ends and the search is of complete lines ignoring end of line markers. \xnn can be used to search for an ASCII character with hexadecimal code nn.

To search for one of a list of alternative characters, enclose the list in square brackets, for example [aeiou] will find any vowel. For a character range use a hyphen to link the start and end of the range. For example, [a-zA-Z0-9] matches any alphanumeric character. To include the – character in a search, place it first or last. To include ] in the list, place it first. To search for any character that is not in a list, place a ^ as the first character. For example, [^aeiou] finds any non-vowel character.

You can search for the start and end of a word with \< and \>. Word characters are the set [a-zA-Z0-9], or [a-zA-Z0-9%$] in script views. For example, the regular expression \<[a-zA-Z]+\> will match a text word, but not if it contains numbers. You can also use \w to match a word character and \W to match not a non-word character. Likewise, \d matches a decimal number and \D to matches a non-number character and \s matches white space (space, TAB, FF, LF and VT) and \S matches non-white space. You can use \w, \W, \d, \D, \s and \S both inside and outside square brackets.

There are search characters that control how many times to find a particular character. These characters follow the character to search for:

* Match 0 or more of the previous character. So 51*2 matches 52, 512, 5112, 51112 and h.*l matches hl, hel, hail and B[aeiou]*r matches Br, Bear and Beer.
+ Match 1 or more. The same as "*", but there must be at least one matching character.

You can also tag sections of matched text by wrapping it in \( and \). You can then insert the tagged text later in the regular expression (or in the replace text in the Find and Replace dialog) using \n, where n is 1 for the first

remembered text, up to `9` for the ninth. For example, `\(foo\)-\1` matches `foo-foo`. More interestingly, the regular expression `\(\<[a-zA-Z]+\>\)-\1` matches `Jim-Jim`, `plum-plum` and the like.

Find Next and Find Previous repeat the previous search forwards or backwards.

# Replace

The Edit menu Replace command is available when the current view is text-based. It opens the Find and Replace Text dialog in which you can search for text matching a pattern and optionally replace it. The search part of the dialog is identical to the Find Text dialog. The search pattern set by the `Find what` field can be a simple match, or can be a regular expression. In regular expression searches, the replacement text can refer back to tagged matches in the search text. See the Find Dialog for details of regular expressions and tagged matches. The >> buttons are also enabled in regular expression mode and let you insert expressions into the search and replace text.

### Replace with
This field holds the text to replace the matched search text. In a regular expression search, you can include tagged matches from the search text using \1 to \9 as described for the Find dialog. For example, suppose you have variables named fred0 to fred17 that you want to convert into an array fred[0] to fred[17]. You can do this by setting the Find what field to \<fred\(\d+\)\> and the Replace with field to fred[\1].

### Replace
The Replace button tests if the current selection matches the Find what field and if it does, the field is replaced by the Replace with text field and the selection is moved to the next match. If the selection does not match, Replace is equivalent to Find Next.

### Replace All
This button searches for all matches in a forward direction starting from the beginning of the text to the end, and replaces them.

# Edit toolbar

The Edit toolbar gives you access to the edit window bookmarks and short cuts to the Find, Find next, Find previous and Replace commands. If you are unsure of the action of a button, move the mouse pointer over the button and leave it for a second or so; a "ToolTip" will reveal the button function and any short-cut key associated with it.

The four bookmark functions toggle a bookmark on the current line, go to the next or previous bookmark and clear all bookmarks. You can set bookmarks on all lines containing a search string with the Edit menu Find command.

If the Edit toolbar is not visible, click the right mouse button on an empty area of a toolbar or of the Signal main window. Then select Edit bar in the pop-up menu. You can use the same procedure to hide it. The bar can be docked to any side of the application main window or dragged off the application main window as a floating bar.

# File comment

This command is available with file and memory views, it allows you to see and edit the file comments. The file comments are five lines of text, each up to 72 characters long, attached to the data file; it can be entered at the end of sampling or by using this command.
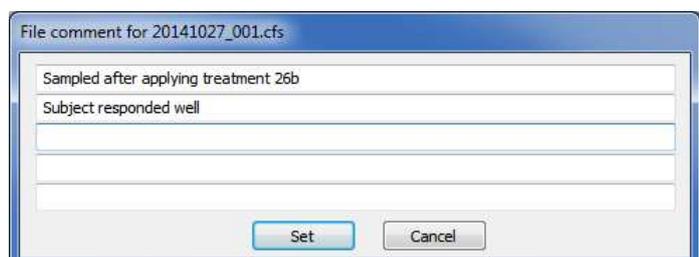
# Frame comment

This command is available with file and memory views, it allows you to see and edit the frame comment. The frame comment is a single line of text attached to each frame in the data file that is available for any purpose.

# Auto Format

Automatic formatting is available for script views only and applies standard formatting while you type and lets you reformat entire scripts or selected script regions. Formatting is done by indenting lines of text based on the script keywords. An indented line is one that starts with white space. The indenting is in units of the tab size set for the view in the Script Editor Settings dialog. Indentation is done with tab characters if you have chosen to keep tabs in the view, otherwise indentation is done with space characters. There are two sub-commands:

### Apply Formatting

If any text is selected in the current view, all lines included in the selection are formatted. If there is no selection, the entire document is formatted. If text is selected, you can right-click in the text view and choose Auto Format Selection from the pop-up menu to activate this option.

### Settings...

The Auto Format Settings dialog controls how text is formatted. Formatting is based on the same scheme that is used for folding text. Each script keyword that starts a block construction (`proc`, `func`, `if`, `docase`, `while`, `repeat`, `for`) increases the indent, and the keywords that complete blocks decrease the indent. However, many users prefer the look of the text with some extra outdents.

The standard CED formatting is to have all the boxes checked, however, it makes no difference to the script operation so, what you choose is a matter of personal taste. It is a good idea to use consistent indenting as it helps you to understand the structure of the script.
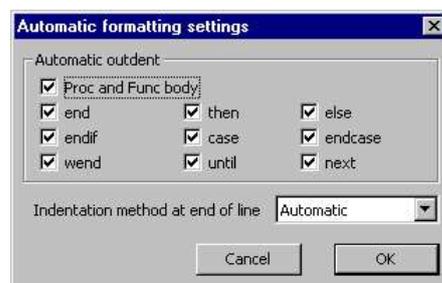
### Proc and Func body

If this box is not checked, all text within a function or procedure is indented. Check the box to outdent the text between the `Proc` or `Func` and the `end`.

### end...next

You can choose to outdent any line starting with one of the keywords `end`, `then`, `else`, `endif`, `case`, `endcase`, `wend`, `until` and `next`.

### Indentation method at end of line

This field determines what happens to the indentation of the current line and the new line when you type the `Enter` key. The settings are:

| | |
|---|---|
| None | No automatic formatting is applied. The new line is not indented. |
| Maintain | The new line is indented to match the indentation of the current line. |
| Automatic | The indentation of both the current line and the new line is adjusted based on the Auto Format settings. |

# Toggle Comments

This Edit menu command is available in script and text sequencer views, and is also available in the context menu when there is a selection. It adds or removes a comment marker at the start of the line for all lines in the current selection. It decides what to do based upon the first character of the first line in the selection.

# Auto Complete

In any script, you will find that you are typing the same text items repeatedly. The editor can save you some time by popping up lists of known words that match your typing. The matching is done by looking for a word break in the text before the text caret, then matching your typing against various categories of words known to the script. Matched words are displayed in alphabetical order and the current word is highlighted.

```
CursorRenumber();
i := ChanM
v :=  ChanMean          im%, 2, Cursor(1),
v +  ChanMeasure
Pri   ChanMult           t from %g to %g on
if meth% = 0 then
    ohms += ((v/i) * sadj);
    num% += 1;
```

You can use the up and down arrow keys to choose an item in the list and the `Enter` key to select an item or double click an item to enter it. The `Esc` key cancels the list. Alternatively, you can just keep on typing, which will narrow the choice of words to match. The Edit menu Auto Complete Setup dialog gives you some control over the words that are matched and when the auto-completion lists appears.

### Word lists

The Word lists section of the dialog lets you choose the categories of words to match your typing against. If you do not want to display auto-completion lists, clear all the categories.

You can choose from script keywords (`Proc`, `Func`, `EndCase`, ...), built in functions and procedures (`SampleSequencer`, `NextTime`, ...), words that already exist in the script and user-defined `Func` and `Proc` names in included files and global symbols in included files.

You may wish to disable the Script text option if you are working with a huge file and you notice an appreciable delay after typing before the list appears. Similarly, you may wish to un-check the Included files option if you use a large number of included files and there is a noticeable time lag before any list appears.

**Auto completion settings**

| | |
|---|---|
| Minimum characters for auto complete display | 4 |
| Initial size (lines) of the matching words list box | 5 |
| Do not display if more matching words than | 30 |

☐ Automatically insert when only one matching word
☐ Also select when user types ( or [ in addition to Enter
☐ Enable auto complete in comments

Word lists
☑ Keywords          ☑ Built-in procs
☑ Script text        ☐ Included files

Cancel    OK

### Minimum characters for auto complete display

This sets the number of characters in a name that must be typed before the list will appear. You can set this to 1 to 12 characters. Setting a low value can cause the pop-up display to become annoying. You need to choose a count that gives you enough help, but not too much. Try a value of 3 to start with.

### Initial size (lines) of the matching words list box

This field sets the maximum number of words to display at a time in the range 1 to 40. If there are more matching words, the list contains a scroll bar. After the list appears, you can re-size it by clicking and dragging on the horizontal edge furthest away from the matched text.

### Do not display if more matching words than

If there are more matching words than you set here, the list is not displayed. You can set this value in the range 1 to 200 words.

### Automatically insert when only one matching word

Because of the design of the script language, apart from variable declarations, all words you type in are likely to have been defined already. If you set this option, once your typing has reduced the list size to 1, the word is automatically inserted. You may want to increase the Minimum characters for auto complete display if you enable this option.

### Also select when user types ( or [

Normally, the list text is inserted when you type the `Enter` key or double-click the list text. If you check this option, the text is also inserted when you type an opening round or square bracket, followed by the bracket.

**Enable auto complete in comments**

Normally, automatic word completion is disabled in a comment. However, if refer to script functions in your code documentation you may find it useful to check this box.

# Preferences

The Edit menu Preferences dialog has multiple pages that are selected using tabs. There are pages that show display options, data options, sampling options, script options, signal conditioner setup, time scheduling, clamping options and compatibility with earlier versions of Signal. The preferences are stored in the Windows registry and are user specific. If you have several different logons set for your computer, each logon will have its own preferences. It is possible to change the preferences from a script; see the `Profile()` command for details.

# Display

The Display page in the Preferences dialog contains options relating to the way data is shown on the screen.

**Show time as**

The Show time as selector controls the time units used within Signal, you can select Seconds, Milliseconds or Microseconds. The selected units will be used in the sampling configuration, for all data files with a time-based X axis including cursors and cursor windows and for all appropriate dialogs including the various process settings dialogs. The main area of Signal not affected by this setting is the script language, which will always see and use values in seconds, though script programs can read the current settings and adjust their behaviour as required. This setting does not affect any data stored by Signal, just the way time is displayed to the user, so you can switch settings without causing problems with data collected using another setting. However time values which are saved by Signal as strings such as the parameters for memory view/measurement processing and active cursors may be misinterpreted after the time units are changed. You can avoid this potential problem by including time units with any value entered such as `"0.1s"` or `"100ms"`, though we assume that in most cases this preference will not be changed.

**Frame start as**

The frame start time shown in the status bar and in printouts may be set to be the time since sampling started in seconds; the same value shown as hours, minutes and seconds or the absolute time of day also shown as hours minutes as seconds.

**Line width for axes / data**

The two Line width items set the line width in points for drawing data and axes respectively. These are relatively unimportant for displays on the screen, where most reasonable settings will only select between lines one or two pixels wide and many different settings will produce the same display. The line width controls are particularly valuable for printing, where the lines drawn can get unsatisfactorily fine. At CED we find that values of 0.75 pt for data and 1.0 pt for axes look pleasant. The line widths also control various other drawing operations; for example the axis line width also controls the cursor widths and borders drawn around views and the data line width sets the basic size of drawn dots and XY view lines and symbols.

**Standard display shows lowest numbered channel at the top**

You can change the order in which data and memory view channels are drawn by clicking and dragging the channel numbers. The Standard display shows the lowest numbered channels at the top check box sets the order when you use the Standard display command or open a new window. If you do not check the box, lower numbered channels are at the bottom.

**Do not use the flicker-free drawing method for data**

Version 3.04 implemented a new buffered drawing method that reduces screen flicker on updates. However, this may impact the display speed. Check the box for the old method.

**Y axis drag may not invert axis range**

Set this check box to prevent a Y axis being inadvertently inverted by dragging it too far.

**Do not change colours if they are very similar to the background**

Signal normally attempts to avoid your data being invisible by overriding your colour choices if they are very similar to the view background colour. Set this check box if you want your selected colours to be used regardless.

**Use Direct2D in text editors**

The text editor we use (Scintilla), can use either the standard GDI method of drawing text or a potentially hardware accelerated Direct2D method (in Windows Vista onwards). The two methods should appear more or less identical.

**Colour scales**

The Colour scale dialog is used to manage and define colour scales that can be used for mapping, for example to show the intensity of a response to a stimulus across the brain surface.

**Text view settings**

These controls configure the appearance of text-based views (script, text sequencer, log views and text views created by a script) on screen and when printing. The dialogs accessed by the Script, Sequencer and Other files buttons will apply any changes globally (to all open files of the selected type and to all future files). The same dialogs can be used from the View menu Font command to make changes to the current view only (there is then an extra button to apply changes to all views).

**Print**

This sets how to print coloured text from a text view. The choices are:

| | |
|---|---|
| Screen colours | Use the displayed screen colours. This is very wasteful of ink or toner if the background is not white. |
| Invert light | If you display your text as a light colour on a dark background, this setting prints on a white background and inverts the text colours. |
| Black on White | Prints black text on a white background. |
| Colour on White | Prints in screen colours on a white background. |

# Colour scale dialog

The Colour Scale dialog is be opened by clicking on the Colour Scale button in the Edit menu, Preferences command Display tab. The dialog sets the colour scale for use in colour mapping, which is currently only possible by using the script language. The script commands ColourGet(), ColourSet(), ArrMapImage() and Spline2D() access the colour scales.

You can choose from a range of built-in and user-defined colour scales and create user-defined colour scales. The dialog is in three sections: an upper part where you edit the current colour scale, the centre line, where you name and save user-defined colour scales, and the Preset colour scales section where you select and manage named colour scales.
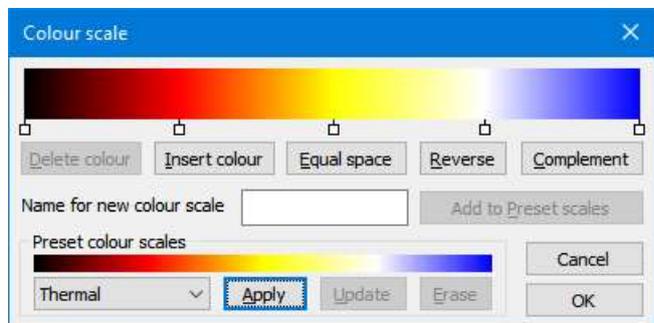
**The colour scale**

The colour scale is the band of changing colour across the upper part of the dialog. The small squares below the colour scale mark fixed colour points. Single click a small square to select it. Double-click a small square to change its colour. The colours between the fixed colour points are formed by linear interpolation. Click and drag the squares sideways to change their position. You cannot move the first and last fixed points. You can add a new fixed point by double-clicking on the colour scale away from any small square. The buttons immediately below the colour scale are:

| | |
|---|---|
| Delete colour | This deletes the selected fixed point; it is disabled if there is no selection. |
| Insert colour | If there is a selected fixed point, this adds a new point next to the selected point on the side with the larger gap. If there is no selected point, this adds a new fixed point in the largest gap between existing fixed points. You can also add a fixed point by double-clicking the colour map clear of any fixed point. You can have a maximum of 10 fixed points. |
| Equal space | This spaces the fixed points evenly across the colour bar. |
| Reverse | This reverses the order of colours. Reversing twice restores the original. |
| Complement | This replaces each colour by its complement, formed by subtracting each colour from white. Complementing twice restores the original. |

**Name for new colour map**

You can save up to 10 user-defined maps in the registry. Set a name and click the Add to Preset maps button to add a new colour map to the list. If the list is full you must use the Erase button to remove a colour map before adding a new one. If you type a name that cannot be used, the Add to Preset maps button is hidden and a text message explaining the problem replaces it. Names of user-defined maps must be unique and may not include the vertical bar character.

**Preset colour maps**

The drop-down list in this section holds the names of colour scales that you can copy into the current colour scale and displays the current list selection. The list starts with the built-in maps, named to suggest their contents, for example Rainbow, Thermal and Geography. The list ends with the names of colour scales that you have created and saved. The buttons to the right of the list are:

| | |
|---|---|
| Apply | Copy the current preset selection to the colour scale at the top of the dialog. |
| Update | Update the preset user-defined colour scale with the colour scale at the top of the dialog. |
| Erase | Delete a user-defined preset colour scale. You cannot delete the built-in maps. |

**Script access to colour maps**

From Signal 7.02 onwards, you can access the colour maps with the `ColourGet()` and `ColourSet()` script commands. The `ArrMapImage()` and `Spline2D()` script commands make use of the colour scales to generate images. You can record changes to user-defined colour scales.

**Storage of colour scale information**

The user-defined colour scales are stored in the registry in the folder:

`HKEY_CURRENT_USE\Software\CED\Signal\Settings\ColourMap`

with the names `ColourMap0` to `ColourMap19`. The user-defined colour scales are saved as `ColourMap10` to `ColourMap19`. The names `ColourMap0-9` are reserved. The built-in colour scales are (currently) hard-coded into the Signal program and cannot be changed by users. As we are finding more uses for these maps we may modify or extend the built-in scales.

**See also:**

Change other colours, `ArrMapImage()`, `Spline2D()`, `ColourGet()`, `ColourSet()`

# Script files

Open the Text view settings dialog from the Preferences dialog Display page to change settings for all views of a given type, or from the View menu Font command to change the current view only. When used for the current view, an extra button Apply to All appears at the bottom of the dialog to apply changes to all views. The Reset All button returns all the dialog values to standard settings.

The bottom half of the dialog holds example text suitable for the view type to show the effect of any changes.

**Style**

Each view type supports one or more text styles. For each style you can set a font (including proportionally spaced fonts). The font includes the size in points (in the range 2-256) and bold and italic settings. You can also choose to force upper or lower case and underlines for all text in the style. You can set a foreground and background colour for the style by clicking on the Foreground and Background rectangles.

All views have the Default style that the remaining styles are based on; it is used for everything that is not covered by one of the other styles. The tab size is based on the width of a space character in this style. If you change any aspect of this style, all the other styles that match that aspect will also change.

In a Script view, you can control the appearance of many different aspects of the display, based on the syntax of the language. There are settings for:
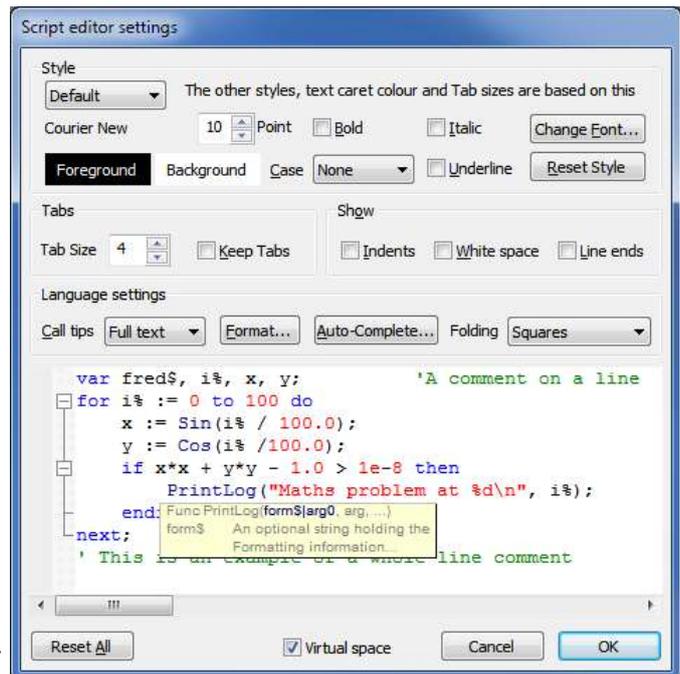
| | |
|---|---|
| White space | Style used when drawing spaces and tabs and control characters. |
| Comment | The style used when displaying comments. |
| Number | The style to use when displaying numbers. |
| Keyword | The style to use for script keywords, such as `for`, `next` and `end`. |
| String | The style for literal strings, such as `"This is a string"`. |
| Function | Used for built-in script functions, such as `PrintLog()`. |
| Operator | The style for script language operators, such as +, – and +=. |
| Identifier | The style for function, procedure and variable names created in a script. |
| Preprocessor | The style for special statements such as #include which are handled by the script preprocessor. |
| Line number | The style used for line numbers and the gutter. When you select this item, line numbers will appear in the example text area. |
| Call tips | The style for pop-up call tips. This style has an extra Tip highlight colour field, used to highlight the current argument in a function. This replaces the Bold, Italic, Case and Underline fields, which are not displayed. |
| Braces | The style to use for matched square and round brackets () and []. When the text cursor is next to a bracket, the editor searches for the matching bracket and applies this style if it is located. |
| Bad braces | The style to use when the text cursor is next to a bracket that does not have a matching bracket. |

**Reset Style and Reset All**

The Reset Style button reverts the current style to standard settings. Reset All reverts all styles.

**Tabs**

This dialog region controls the size of tabs (set in units of the width of a space character in the Default style) and if tabs are implemented by saving a tab character in the text (check Keep Tabs) or are implemented as multiple spaces (clear the Keep Tabs check box). If you use a fixed pitch font, such as `Courier New`, then it does not matter too much if you choose to keep tab characters or not. If you use a proportional font for anything except comments, it is better to keep the tab characters. When automatically formatting a script, the Keep Tabs setting determines if indents are generated with tab characters or spaces.

**Show**

In addition to displaying the text, you can also choose to display information about the white space in your file. The settings are:

Indents    It can be useful when manually lining up indented loops in a script to see the indent level. Check this box to display vertical lines at each tap stop in the leading white space of each line.

White space   Check this box to display spaces with a centred dot and tabs as right arrows.

Line ends   Check this box to see the Carriage Return (CR) and Line Feed (LF) characters that mark the end of a line. This can be helpful to understand what is happening when working with scripts that move the caret around.

**Language settings**

This area of the dialog is used by script and output sequencer views. It sets your preferences for call tips, automatic formatting, automatic text completion and folding. The Format and Auto Complete buttons duplicate Edit menu commands, and are described separately.

**Call tips**

A Call tip is a block of text that pops up in a script view when you type the opening brace of a function of procedure name. The call tip text contains a synopsis of the command and a list of the command arguments, with the current argument highlighted. You can choose from None (no call tips), Single line (just the command name and arguments) and Full text (includes a command synopsis).

**Folding**

Script views and output sequence views can display a folding margin, and allow you to fold the code by clicking in the margin, from the View menu Folding command, and by right clicking in the view and selecting Expand All Folds, Collapse All Folds or  Toggle all folds (which finds the first fold point in the document, reads its folded or unfolded state and then sets all folds to the opposite state). In a script view, fold points are based on a lexical analysis of the script text. You can choose from one of four folding styles, or have no folding margin.

**Virtual space**

If you check this box, you enable virtual space in the editor. This allows you to move the text caret beyond the end of a line by clicking with the mouse or by using cursor keys. This can be useful when you want to position text (for example comments) without the need to use tabs or spaces to position the caret. If the caret is in virtual space and you enter text, the editor inserts spaces up to the caret position and the position becomes "real". However, allowing virtual space (which was the default before version 5.08), can force you to use extra key presses to move to real start or end of a line (Home/End keys) when you press cursor up and down to move vertically through the text.

# Call tip details

A Call tip is a block of text that pops up in a script view when you type the opening brace of a function or procedure name or you hover the mouse pointer over a symbol name or keyword. For function and procedures, the call tip text contains the command name and arguments and a synopsis of the command. If the tip appears as a result of typing the opening brace, the editor attempts to highlight the current argument in the tip and if the command has multiple variants, you can select the variant to display by clicking on the arrows in the call tip.

In the Script file text settings dialog you can choose from None (no call tips), Single line (just the command name and arguments) and Full text (includes a command synopsis). In Single line mode (not when opened by hovering the mouse pointer) you can click on the body of the tip to show the full text.

**Call tips for built-in functions**

For built-in functions, the call tip text holds the command name and arguments plus one sentence of description. This information comes from the text file script.tip in the folder where Signal is installed.

**Call tips for user-defined functions**

For user-defined `Proc` and `Func` items, the arguments are taken from the line containing the `Proc` or `Func` keyword (expected to be the first item on the line). If the line before the `Proc` or `Func` is a comment, up to 10 lines of comment are used as a description. This works for `Proc` and `Func` items in the current source and in any included files. If the line before is blank, and the line after is a comment, up to 10 lines following are used as a description. Putting the comments after the `Proc` or `Func` line has the advantage that the comments can be folded within the `Proc` or `Func`.

The search is terminated by any "divider" lines. A divider line is one that is at least 10 characters long (after removing the initial comment mark ') and with no more than 2 characters being different. Here are some example divider lines:

```
'''''''''''''''''''''''''''''''''''''''''''''''
'-----------------------------------------------
'|=============================================|
```

If you document your functions and procedures as in this example, they will generate tidy call tips:

```
'This is an example of how to document for a nice call tip
'arg1  If the first word on a line is followed by more than one
'      space or a tab, the rest of the line is indented. If a line
'      starts with two or more spaces or a tab, it is indented.
'arg2  Description of second argument
'arg3  And something about the third argument
Func Example(arg1, arg2, arg3)
var x,y;  'the start of the code
...
```

The example text above would produce a call tip looking like this. The comment markers at the start of each line are removed, and the multiple spaces after the first word or at the start of each line are replaced with a tab character.

If you do not want a `Proc` or `Func` to have a call tip see the section, below, Disable call tip for an item. If you disable a call tip in an include file, this will also stop it appearing in an Auto-completion list.



**Call tips for const and var names**

From Signal version 6.03 onwards we also provide call tips for names defined in `const` and `var` statements and argument names defined in the first line of a `Proc` or `Func` statement. The tip will be the entire text of the line that includes the definition. Any tab characters in the line are replaced by space characters, then runs of space characters are replaced by a single space to make the tip a short as possible.

**Call tips for included files and Auto Complete**

You can have tips for global symbols for items in included files. You must enable this in the Edit menu Auto Complete command by checking Included files. Call tips and Auto Complete use the same internal logic

**Go to declaration**

If an item has a call tip, it also will have a right-click menu option. For user-defined items, this will let you Go to the declaration of the item. For built-in items, this will open Help for the item (as long as the help file is available).

**Limitations**

Call tips and Auto Complete have to work on scripts that are works in progress and that may not be syntactically correct (i.e. they may not compile). This means that we cannot use the same syntax analysis engine that the script compiler uses. To generate a useful result, we make the following assumptions when parsing scripts for user-defined `Proc`, `Func`, `const` and `var` statements:

- The `Func`, `Proc`, `const` or `var` keyword is the first non-white space symbol in a line.

- The `end` keyword that terminates each `Func` or `Proc` is the first non-white space symbol in a line.

- For symbol names to be detected, they must be on the same line as the `Func`, `Proc`, `const` or `var` keyword that introduces them.

It is important that the end keyword is recognised as we use this to skip over user-defined functions when searching for a global symbol.

**Disable call tip for an item**

Although having call tips is generally very useful, you may sometimes want to hide user-defined `Func`, `Proc`, `var` or `const` names from the call tip and **Go to** systems. For example, you might have an include file that implements some useful functionality through a single user-defined `Func`, but that has several helper functions and global `var` and const items that you do not want to be generally visible as they just cause clutter. There are two ways to do this. The simplest is to start a comment on the same line with an exclamation mark.

```
const kPrivateConst := 1.234; '! No call-tip or autocomplete
```

The second way is to separate the keyword (`Proc`, `Func`, `const`, `var`) from the symbol names that it introduces. However, it is possible that we will further develop the parser so it does not rely on text being on a single line, so this method may not work forever. We mention this as this method was documented in older versions of Signal.

```
Proc
PrivateFunction() 'This is only used within this file
...
end;
```

# Sequencer files

The editor settings dialog for output sequencer files is very similar to that for script files. The example text in the lower half of the dialog displays some typical sequencer code, and there is no support for Call tips, auto formatting or automatic completion.

**Style**

Each view type supports one or more text styles. For each style you can set a font (including proportionally spaced fonts). The font includes the size in points (in the range 2-256) and bold and italic settings. You can also choose to force upper or lower case and underlines for all text in the style. You can set a foreground and background colour for the style by clicking on the **Foreground** and **Background** rectangles.

All views have the **Default** style that the remaining styles are based on; it is used for everything that is not covered by one of the other styles. The tab size is based on the width of a space character in this style. If you change any aspect of this style, all the other styles that match that aspect will also change.

There is a list of styles that you can apply to the different elements of the sequencer text. The **Default** style is used in exactly the same way as for script views as the basis for all other styles and as the font that is measured to set the tab size. The remaining styles are:

White space   Style used when drawing spaces and tabs and control characters.

Comment   The style used when displaying comments.

Number   The style to use when displaying numbers and digital i/o expressions such as `[...0101]`.
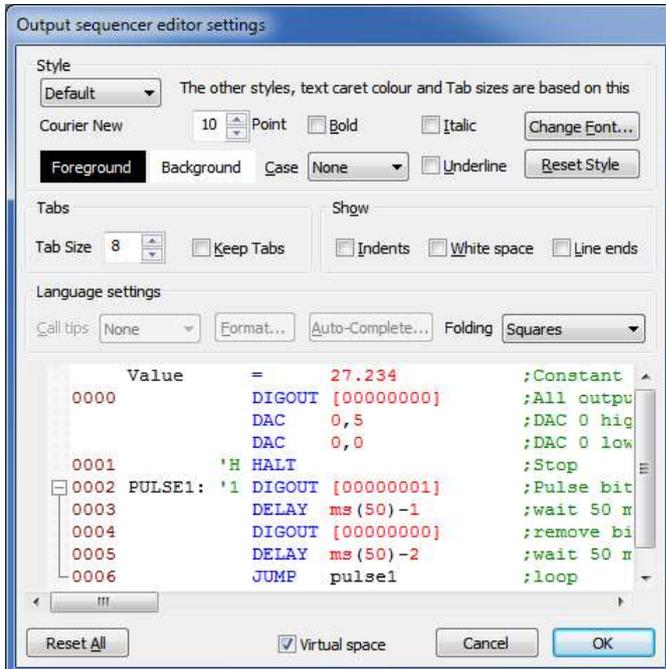
| | |
|---|---|
| Keyword | The style to use for standard output sequencer instructions. |
| Display | Text introduced by > for display on screen during sampling. |
| Directive | Items such as SET and VAR that do not generate output instructions. |
| Operator | The style for mathematical operators like +, -, * and /. |
| Identifier | The style for user-defined variable names and labels. |
| Step | The style for the optional step numbers at the start of an instruction line. |
| Key | The style for keyboard links introduced by a single quote, for example 'H |
| Functions | The style for built-in conversion functions like ms(). |
| Deprecated | The style to use for outmoded output sequencer instructions like DAC0 that have been replaced and may not be supported in future revisions. |
| Line number | The style used for line numbers and the gutter. When you select this item, line numbers will appear in the example text area. |
| Braces | The style to use for matched square and round brackets () and []. When the text cursor is next to a bracket, the editor searches for the matching bracket and applies this style if it is located. |
| Bad braces | The style to use when the text cursor is next to a bracket that does not have a matching bracket. |

**Folding support**

The output sequencer editor supports code folding based on the keyboard link entry points. That is, you can fold up all code from a line holding a keyboard link up to the next line holding a link.

# Other files

The text editor settings dialog for all views except script and sequencer views is the same as the dialog for script views, except that there are no language settings and there is only the Default and the Line number text style.

# Data

The Data page in the Preferences dialog controls the way data is stored and exported, plus other data-related choices.

### Data export format: Save waveform data as

The Save waveform data as field control decisions on how waveform data is written to CFS files when no other information is available to help Signal make this decision. Waveform data in CFS files can be stored either as floating point numbers or as scaled integers. Scaled integers are the format of data sampled by the 1401, occupy less space on disk and are the only format that can be read by the DOS SIGAVG software, but can be less accurate and can overflow (when a data value outside the possible integer range is required). Floating point numbers are more bulky on disk, but are more accurate and cannot overflow. In most circumstances, the format used for waveform data is derived from the data source or set by the destination file, but when creating a new CFS data file by other mechanisms (saving a some types of memory view in particular), the format used is controlled by this field. Select Real for maximum data accuracy, Integer to produce smaller data files or for SIGAVG compatibility.
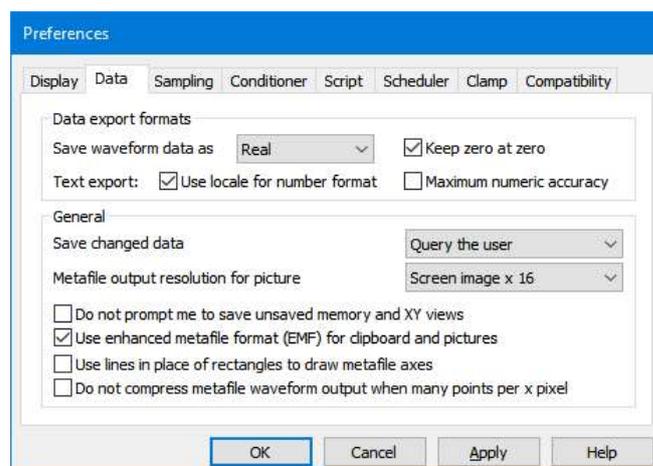
### Data export format: Keep zero at zero

Similarly, this item controls how the scaling factors for integer data are calculated if no other information to aid the choice is available. Signal tries to use the same integer scaling factors as the previous values for the frame (or the previous frame where appropriate), but may need to recalculate the factors if the data values become too large for the existing scaling or too small to represent accurately. If this check box is set, Signal will always calculate scaling factors that keep zero in the integer data corresponding to zero in the scaled values, which can be convenient in some circumstances.

### Text export: Use locale for number format

Normally, Signal uses standard compatibility options for all text output. If you check this option the local number formatting settings within the operating system will be used for all text export; data copied to the clipboard as text and data exported as a text file.

### Text export: Maximum numeric accuracy

Normally, when copying Real marker, Memory view and XY view data as text to the clipboard or file, Signal uses sufficient decimal places to give a reasonable representation of a floating point value. Check this box to force full accuracy. This may cause numbers you expect to see as 0.1 appear as 0.09999999999999998, however it means that no precision is lost if you transfer data as text to another program.

### Save changed data (default data update)

The Save changed data selector sets the default action for file data that has been changed in memory. For example, the script language could have been used to differentiate the data in a channel. You can choose to write changes back always, to only write changes after querying the user or to always discard changed data. This option sets the default initial state for all data files opened, use the File menu Data update mode command to control data write-back for a single file.

### Metafile output resolution for picture

Signal saves file, memory or XY views as pictures in either bitmap (screen image) or metafile format. A metafile describes a picture in terms of lines and text based on a grid of points. Metafiles have the advantage that they can be scaled without losing resolution.

You can choose the density of the grid. The higher the density, the more detail in the picture. The problem for time and result views with a lot of data points is that the higher the grid density, the more lines need to be drawn, and many drawing programs have limits on the number of lines they can cope with.

You can set a grid based on the screen resolution, or a grid such that the width of the image is a fixed number of points. If you are not sure what setting to use, start with *Same as screen image* and adjust it as seems appropriate for your use.

### Prompting to save views

Several users pointed out that it was very irritating to be asked if you want to save data that is derived from other data, and that can easily be derived again. This is especially true when you are developing a new script application. If you check the *Do not prompt me to save unsaved result and XY views* box, Signal will close and throw away result and XY views without requiring a confirmation. As this is potentially destructive, we suggest that you don't use this option when you care about the result or XY data.

### Use enhanced Metafile format

Signal supports two metafile formats: Windows Metafile (WMF) and Enhanced Metafile (EMF). WMF is a relic of 16-bit Windows and has limitations, but is widely supported. EMF is the standard for 32-bit programs and has many more features. However, some graphics packages do not support this fully (this was written in late 1998, but is still true in 2007).

### Use lines in place of rectangles…

This only affects metafile output. Some graphics programs cannot cope with axes drawn as rectangles; check this box to draw axes as lines. We use rectangles to make sure that axes drawn with pens of other than hairline thickness join up correctly.

### Do not compress metafile waveform output ...

When generating metafiles (and also when drawing to the screen, though that is not affected by this option) Signal can save space and time by drawing waveforms with a large number of points per pixel as a series of vertical lines, one pixel apart. This does not result in drawing errors because there are already many waveform points drawn on top of each other.

When drawing to a metafile this can cause problems, as there is not the same match between the vertical line width and vertical line spacing. If you check this box, no such compression is done when you generate a metafile. For the most precise metafile data, set the Metafile output resolution to screen*16 and check this box.

# Sampling

The Sampling page in the Preferences dialog contains user preferences relevant to the sampling of data.

### Directory for new data

When Signal samples data, the new data is stored in a temporary file while it is being collected and only stored in a final CFS file when the file is saved. The Directory for new data field sets the directory in which Signal puts this temporary file. If this field is blank, Signal will use the current directory (which may not be where you expect), so it is a good idea to set one. If you want to save over a network, for example, or store new data files on media that is slow to write, you can use this field to ensure that the temporary file directory (which is where all the real-time writing occurs) is on a fast hard disk.
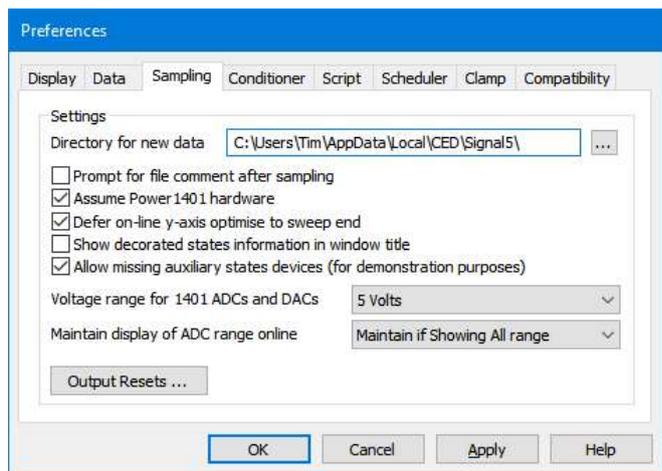
When you save a new data file, Signal prompts you for the file name. What happens next depends on where you choose to save the file. If the file is on the same volume (disk drive) as the temporary file, Signal just renames the file. If the volume is not the same, Signal copies the file, then deletes the original.

**Prompt for file comment after sampling**

The Prompt for file comment after sampling item is used to encourage entry of a file comment when a new data document has been created by automatically popping-up the file comment entry dialog.

**Assume Power1401 hardware**

In order to correctly show the sampling rates attainable and limits to the pulse output resolution, Signal needs to know if Power1401-class hardware (having a 10 MHz internal clock frequency) is in use. Signal tries to detect the 1401 type automatically when it starts up, this check box sets whether Signal will assume the presence of a Power1401 if it cannot detect the 1401 type directly. As all modern 1401s have a 10 MHz clock available, you should leave this checked unless you know you are going to be using an original micro1401.

**Defer on-line y-axis optimise to sweep end**

If a channel display is optimised in the y-direction and this check box is checked, Signal will wait until the sweep finishes before scanning the data collected in order to perform the optimisation, thereby ensuring that a complete sweep of data is used. If this box is unchecked then the optimisation will be done using only the data collected so far in that sweep. This may mean no data has been collected and axes will be set to default limits which may well not be ideal.

**Show decorated states information in window title**

When sampling using multiple states, Signal will display the frame state number (with optional state label) in the window title bar for frame zero. This is intended to make it easier to see what is going on during data acquisition. Signal can also display extra information showing the progress of the states sequencing or, if an auxiliary states device is in use, the auxiliary states device settings. This extra information is only shown if this check box is ticked, otherwise the state number and label only are shown.

**Allow missing auxiliary states devices**

When sampling with a configuration that uses auxiliary states devices, sampling will normally fail if the devices that are used are not present. This option allows sampling with a configuration that uses a missing device for demonstrations or training purposes.

**Voltage range for 1401 ADC and DACs**

Most 1401s have a ±5 Volt input range, but some are set up for ±10 Volts; more modern 1401 hardware can be switched between these voltage ranges using the Try1401 application. Signal detects the voltage range in use automatically in all supported types of 1401. Choose from 5 Volt, 10 Volt and Last seen hardware, note that this choice is not changing the hardware; it is changing how Signal expects the hardware to be configured (or, in the case of Last seen hardware, telling Signal to reconfigure itself every time it opens a 1401). You will be warned if Signal detects a conflict between the current preference settings and the installed hardware, normally you can simply tell Signal to adjust itself to match the hardware now being seen. The voltage range setting affects scaling in the sampling configuration and DAC output values in the output sequencer. It has no effect on scale values in previously sampled data files. If a sampling configuration generated with the voltage range set to 5 volts is read into a copy of Signal set up for 10 volts or vice versa, the scalings in the configuration are automatically adjusted to keep things correct. Older sampling configurations (from before Signal version 4.06) do not include this voltage range information and may need adjusting.

**Maintain display of ADC range online**

When a programmable signal conditioner settings are changed or a telegraph voltage changes during sampling, the display y-range for the relevant channel can do one of three things. Never, keep the y-axis limits will keep the y-axis unchanged. In this case there may be no visible indication that the amplifier gain has changed although the resolution of the data may change. Maintain it if Showing All range will alter the y-axis when the gain changes to show the new full range of the ADC provided the full range was previously displayed. Maintain ADC range percentage will maintain the ADC range previously displayed but show the new values corresponding to the amplifier settings. In all cases you should note that, for example, data drawn at 3 mV would continue to be drawn at 3 mV. It is only the axis limits that may change.

### OutputResets...

Press this button to open the dialog for the application-wide output reset settings. These allow you to define DAC and digital outputs that will be set before and after any sampling, for example to ensure that a stimulator is reliably turned off.

#### *Auxiliary states device*

Auxiliary states devices are optional hardware, normally stimulators, which can be automatically controlled during sampling and configured differently for each sweep state. Before Signal version 7.01, you could select an auxiliary states device to be used in this page, now a list of auxiliary states devices is maintained in the sampling configuration dialog States page.

# Output Resets

In most cases, when sampling is not in progress the signal levels on 1401 outputs are of no interest to the researcher, but if an output is driving equipment that needs to be held in an inert state (for example a skin stimulator) it can be very important (if only for the peace of mind of the experimental subject) to ensure that the 1401 outputs are in the 'off' state before sampling starts and are reliably restored to this state when sampling finishes. This can be a particularly difficult problem if sampling stops during the middle of a sweep as the 1401 outputs will be left in whatever state they were at the point where sampling stopped. To avoid such problems, Signal provides mechanisms to



define and apply forced output resets. There are two sets of forced output settings; application-wide settings which are controlled by this dialog available from the Sampling page in the Preferences dialog and a separate set of values held in the sampling configuration information. The two sets of information are very similar; any reset level defined for an output in the sampling configuration overrides the corresponding application-wide settings for the same output. Note that if you do not have outputs that need to be controlled in this manner you do not need to make use of these settings.
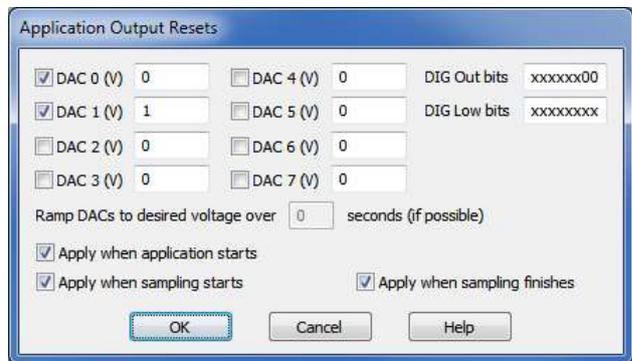
### DAC outputs

All reset DAC values are defined in volts, each DAC has a check box which enables the use of the reset value.

### Digital outputs

There are separate fields for the main digital outputs used by the pulse outputs and the DIGOUT sequencer instruction and for the lower digital outputs used by the DIGLOW sequencer instructions. Each field is eight characters long with the first character controlling the highest output bit and the last character controlling the lowest bit. An 'x' character leaves the corresponding output bit unchanged, a '0' clears the output bit (sets it to TTL low) while a '1' sets it (TTL high).

### Ramp DACs to desired voltage over ...

It may be necessary to avoid simply setting DAC outputs to the required voltage and instead ramp the output level to the correct level over a period of time. This option is not currently available but if required by significant numbers of users will be implemented in a future release.

#### *Apply when application starts*

If this check box is set then the application-wide output reset settings will be applied when Signal starts.

#### *Apply when sampling starts*

If this check box is set then the application-wide output reset settings will be applied when sampling is about to begin.

*Apply when sampling finishes*
If this check box is set then the application-wide output reset settings will be applied when sampling has finished.

# ADC and DAC range in older sampling configurations

In Signal version 4.06, the sampling configuration information saved on disk was extended to include the 1401 ADC and DAC voltage range for which it had been generated. This allowed sampling configuration information to be automatically adjusted for the 1401 ADC and DAC range currently in use so as to maintain correct calibration - the same voltage on a 1401 ADC input will represent the same calibrated value and the voltages generated on the DAC outputs will be the same. This will allow sampling configurations to be moved between different systems without problems.

Older Signal sampling configuration files do not contain any 1401 ADC and DAC voltage range information and this prevents Signal from making any adjustments - it has to assume the ADC and DAC scaling information is correct. **Usually this presents no problem at all**, but if the sampling configuration file was set up with a 1401 ADC and DAC range which is different from the current 1401 then the calibrations will be incorrect. In most cases, the issue is simply that the sampled data values will be a factor of 2 too large or too small and this can be corrected-for in the analysis or by re-calibrating data files. There could be more significant problems if stimulator levels (controlled by DAC voltages) are incorrect or if your experimental methods depend strongly upon measurements taken during the experiment - for example if you start by measuring the TMS stimulation level that produces a threshold response. Because of this, Signal generates a warning message if it encounters a sampling configuration without 1401 ADC and DAC voltage range information.

If you encounter this warning message **you should not assume that there is a problem**. Nearly all 1401s use a 5V ADC and DAC range and, if your 1401 hardware is 5 volt or you have no reason to believe that the sampling configuration was set up for a different 1401 ADC and DAC range, you can assume that all is well. If you want to be sure you should check that the ADC and DAC scalings in the sampling configuration are correct, the 1401 voltage range is shown in the Ports configuration page of the sampling configuration dialog.

If you save the sampling configuration data back to the same disk file, the new data will contain 1401 ADC and DAC range information and you will not be warned again.

# Conditioner

The Conditioner page in the Preferences dialog lets you select the signal conditioning hardware you wish to use, set the COM port used by a signal conditioner and provides support for debugging signal conditioner connections.

### Conditioner type
You can select the signal conditioning hardware you wish to use (or none) during Signal installation; your choice of hardware can be changed here at any time.

### COM port for signal conditioner control
Most, but not all, signal conditioner hardware is controlled using a serial (COM) port. If the hardware you have selected is controlled in this way you can select the COM port used here. You can select any port in the range COM 1 to COM 19. Every time you change the port, Signal searches for any conditioners attached to it and will update the status panel.

### First channel
This item is used to optimise the search for connected signal conditioners. For the 1902 signal conditioner the search for connected 1902s will start with the selected First channel so if the lowest channel number in your set of 1902s is higher than zero you can speed things up by setting this to your first 1902 channel number.
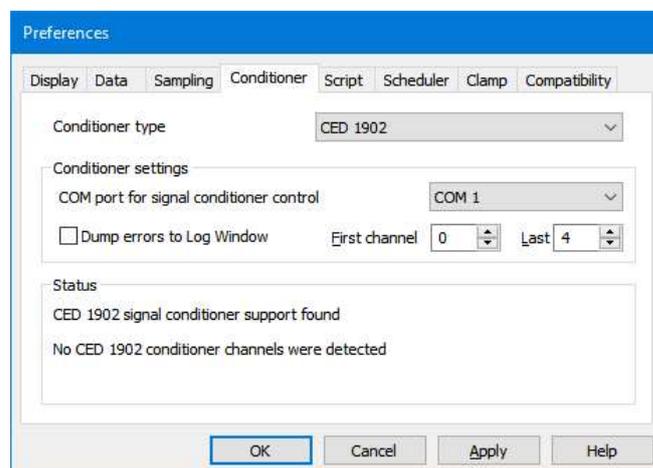
### Last channel

This item is used to optimise the search for connected signal conditioners. For the 1902 signal conditioner the search for connected 1902s will continue until at least the selected Last channel and will only stop when a 1902 with a higher channel number than Last channel is not found. Therefore this can be set to your last 1902 channel number, but you can also leave it set to zero of there are no gaps in your 1902 channels (which is the normal case).

### Dump errors to Log window

If you have trouble working out why your signal conditioners are not being detected, check the Dump errors to Log Window box to have diagnostic messages written to the Signal log window.

### Status

The Status panel below gives information on the type of signal conditioner support that is in use. If it can detect one or more signal conditioners, it also holds information about the channels that have conditioner support.

# Script

The Script page in the Preferences dialog contains items regarding the script editor and script behaviour.

### Save modified scripts and sequences before they run

Check this box so that any changes to a script will be automatically committed to disk before the script is run. This check box also enables automatic saving of modified sequence text before sampling is carried out.

### Enter the debugger when a script has a runtime error

If this box is checked then if a runtime error (a fatal error that stops script execution) occurs while executing a script happens then the script view will be displayed with the debugger running. This allows the user to check on the values stored in the variables as well as looking at the call stack.

### Maximum text lines in Log window

You can use this field to prevent huge amounts of text accumulating in the Log view. If there are more lines of text in the Log view than set here, and a script writes more, the oldest lines are deleted to leave room. Set this item to 0 for no limit.
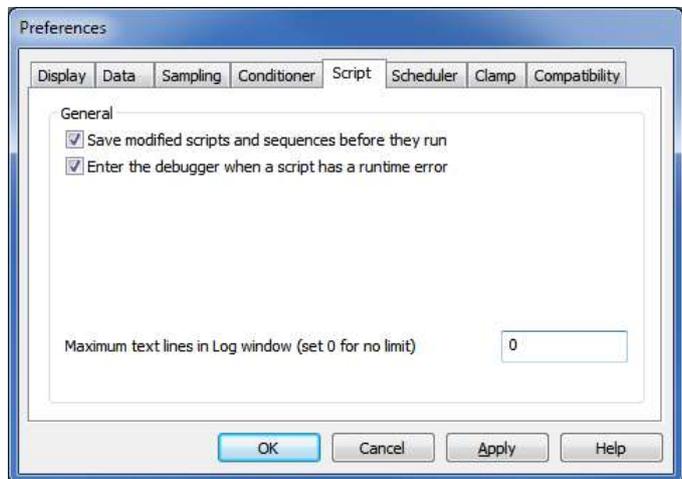
# Scheduler

The Scheduler page in the Preferences dialog holds controls used to limit the processor time consumed by the Signal user thread while sampling and idling with a script running. If Signal takes too much time, the system feels unresponsive. It does not affect the time-critical thread that writes sampled data to disk. You can read more about threads below. Signal runs a background routine when it has idle time and also periodically on a timer. The routine handles the following tasks:

- When sampling, it gives windows a chance to detect that the maximum time has changed, which may cause windows to update and processing to occur. Any invalidated windows will update the next time Signal gets idle time.

- If a script is running, it gives any "idle function" in the script a chance to run.

- In automatic file naming mode it starts the next file running.

There are three fields that limit the time used in the background routine. They have no effect on the time used when Signal runs a script that does not idle (see the `Yield()` or `YieldSystem()` script commands for this). The standard values work for most cases.
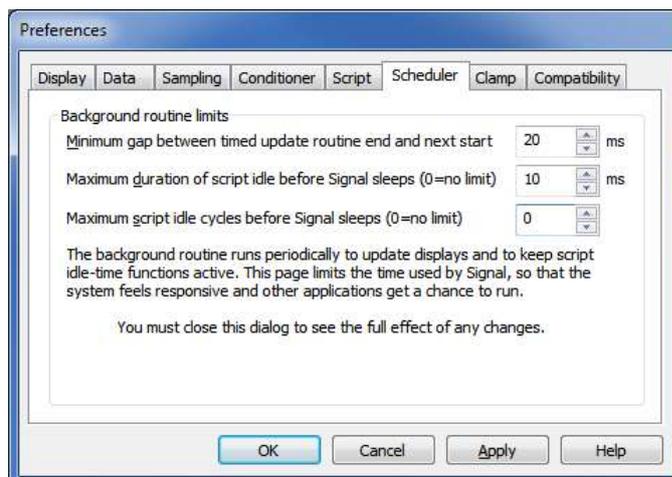
### Minimum gap between timed update routine end and next start

If the time interval set by this field passes without the background routine running, it is scheduled to run as soon as possible. You can use values in the range 1 to 200 milliseconds. The standard value is 20 milliseconds. The lower the value, the more time Signal will spend on background processing relative to other applications. This field also limits the time that Signal will sleep for (see the discussion of threads, below).

### Maximum duration of script idle before Signal sleeps

When Signal gets idle time (see the discussion of threads, below), you can limit the time it uses before Signal goes to sleep. Signal uses idle time to run script idle routines, such as those created by `ToolbarSet(0,...)`. You can set from 0 to 200 milliseconds (0 means no time limit). The standard value is 10 milliseconds. The larger the value, the more the script idle routine runs at the expense of other applications.

### Maximum script idle cycles before Signal sleeps

As an alternative to limiting the script idle routine by elapsed time, you can limit it by the number of times it is called. You can set from 0 to 65535 times (0 means no limit). The standard value is 0. Setting both this and the *Maximum duration...* field to 0 is unkind to other applications. Setting this to 1 is the most generous to other applications.

### *About threads*

A *thread* is the basic unit of program execution; a thread performs a list of actions, one at a time, in order. To give you the impression that a system with one processor can run multiple tasks simultaneously, the system scheduler hands out time-slices of around 10 milliseconds to the highest priority thread capable of running. Tasks at the same priority level share time-slices on a round robin basis. Lower priority tasks rely on higher priority tasks "going to sleep" when they have nothing to do or when they are blocked (for example, waiting for a disk read). If this did not happen, low priority tasks would not run.

When Signal gets a chance to run, it processes pending messages such as button clicks, keyboard commands, mouse actions and timer events and then updates invalid screen areas. Finally, Signal is given idle time until it says it does not need any or new messages occur. If Signal needs no more idle time it sleeps until a new message appears in the input queue. The *Minimum gap...* timer wakes up Signal if nothing else happens.

# Clamp

The Clamp page in the Preferences dialog is used to control aspects of the clamping features provided in Signal. There are three options available.

### Use Signal features for patch, voltage and current clamp

This option controls the availability and visibility of Signal features used for patch, voltage and current clamp experiments. The features controlled by this option are the online clamping support (including dynamic clamping), leak subtraction analysis and the generation and analysis of idealised single-channel traces. If you are not involved with patch, voltage or current clamping work then you may want to use this option to turn off and conceal these features to avoid confusion.
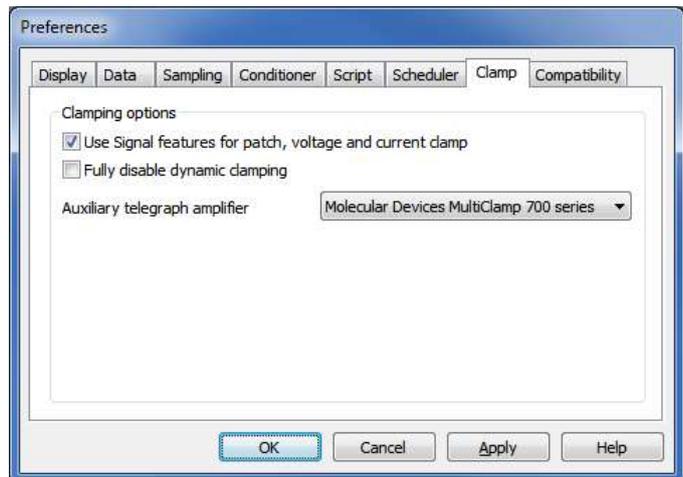
### Fully disable dynamic clamping

This option disables the use of dynamic clamping during sampling, even if the sampling configuration contains dynamic clamp models, it is intended to allow the use of sampling configurations with dynamic clamp settings with 1401 types that do not support dynamic clamping.

### Auxiliary telegraph amplifier

This option controls the use of the auxiliary telegraph support which can be used to interface with the Axon Instruments (now Molecular devices) MultiClamp 700 and AxoClamp 900A amplifiers and the Heka EPC 800 amplifier. The auxiliary telegraph amplifier type to be used can be selected during Signal installation; the installation choice can be changed here at any time. Note that telegraph support using analogue levels sampled by the 1401 is always available in addition to any auxiliary telegraphs in use.

# Compatibility

The Compatibility page in the Preferences dialog contains options for reverting Signal behaviour to that of earlier versions. As stated in the dialog, please change any scripts that depend upon the old behaviour provided by these check boxes - they may be removed in a future release of Signal. There are currently four options available:

### If showing frame zero, switch to frame 1 when sampling finishes

Signal changed in version 4.03 to show the last sampled frame - the most recent saved data - when sampling finishes. Checking this box will cause Signal to revert to showing frame 1.

### Don't defer sampling stop until the end of the current sweep

Signal changed in version 5.09 so that if you pressed Stop while a sampling sweep was in progress, Signal did not stop sampling until the current sweep had finished. Checking this box will cause Signal to revert to stopping sampling immediately which can leave a partial sweep behind that will be discarded when sampling finished.
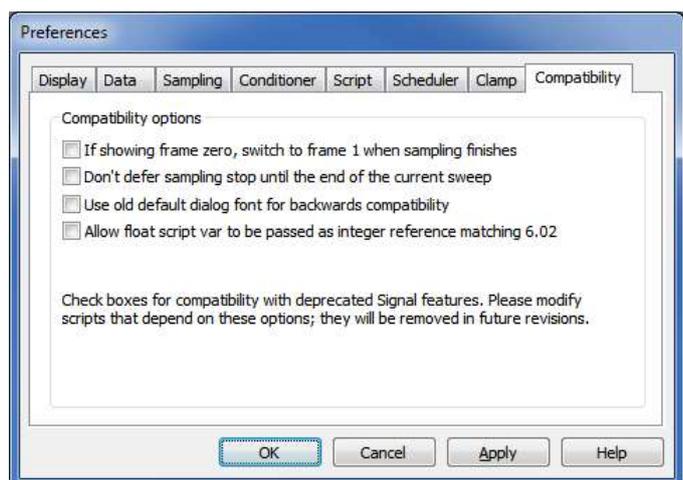
**Use old default dialog font for backwards compatibility**

Version 6.03 onwards sets the equivalent of DlgFont(1) each time the script starts rather than the old default of DlgFont(0). This generates better-looking text, however, it is possible that this will change the layout of user-defined dialogs, which could cause text to vanish, making a dialog difficult to use. Check the box to force the old behavior.

**Allow float script var to be passed as integer reference matching 6.02**

In earlier versions of Signal, for historical reasons it was possible to pass a floating-point variable to a function where a reference to an integer variable was required. This resulted in the function having an integer variable that was actually floating point and could hold non-integer values. We have prevented this in version 6.03, but provide this checkbox (for now) so that any old scripts that actually use this will still work. Please, if you find you have to use this option, change the scripts in question as well!

# View menu

This menu controls the appearance of the Signal data and XY windows, other menu commands are used to control the appearance of text-based views.

## Toolbar, Edit bar, Status bar

These three commands enable and disable the display of the application toolbar, the edit toolbar and the status bar. The toolbar is the array of buttons normally displayed below the Signal menu bar. The edit bar contains buttons for special functions valuable with text views. The status bar is always at the bottom of the application window and displays information about the current (highlighted) view. These items display a check mark if the corresponding bar is displayed, click on the item to toggle the display state.
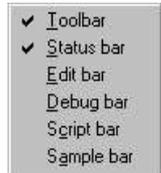
These bars can also be shown and hidden by right-clicking on an unused area of the application window to open a pop-up menu. This menu can show and hide other types of toolbar. The script and sample bars are described in the Script menu and Sample menu sections. The debug bar is described in the script language documentation.

## Next frame, Previous frame

These commands, together with the buttons at the bottom left of the data window and the shortcut keys `PgUp` and `PgDn`, change the current frame to the next or previous frames in the data document. If the current frame is the first or last frame in the document then the corresponding menu item and button are greyed out. The `Ctrl+PgUp` and `Ctrl+PgDn` shortcuts change to the last or first frame in the document. When sampling is in progress `Ctrl+PgUp` switches to always showing the last frame filed, until the frame is manually changed, using `PgUp` to move to the last frame does not cause this effect.

## Goto frame

This command (shortcut `Ctrl+G`) opens a dialog to allow you to enter a frame number to move to directly.

## Show buffer

This command (shortcut `Ctrl+B`) toggles between showing the frame buffer and the current frame. When the buffer is shown, the menu item is shown checked, and the view title is modified to show that the buffer is visible. The frame buffer is a separate frame of data 'behind' each open CFS file which is provided by Signal. See *Analysis menu* for more details of the frame buffer.

## Overdraw frames

This command (shortcut `Ctrl+D`) switches the display between normal mode, displaying the current frame only, and overdraw mode, which displays all of the frames in the frame display list as well, either behind one another or using a 3-dimensional representation. If the frame display list is empty at the time when overdrawing is enabled it is set to 'all frames'. When overdraw mode is enabled, this menu item is shown checked.

The frame display list defines frames to display in addition to the current frame, it is set using the Overdraw settings... option. These additional frames use different colours from the current frame trace. They can either all have the same colour, set using the View menu Change Colours command, or each frame can have a different colour.

# Add frame to list

This adds the currently displayed frame to the frame display list. If overdraw mode is enabled, all frames in the frame display list are displayed along with the current frame. If the current frame is in the display list, this command becomes Remove frame from list. Adding or removing a frame from the display list in this way will destroy any dynamic behaviour of the list. For example: if you have requested to overdraw all tagged frames and then add another frame using this menu item, the display list will then remain fixed even if you subsequently tag or un-tag a frame.

# Overdraw settings...

This command (shortcut `Ctrl+Shift+D`) opens a dialog that sets the frame display list and also controls the manner in which overdrawn frames are displayed. The upper part of the dialog is used to enable frame list overdrawing and to define the frames to be overdrawn, the lower part controls the optional 3D overdraw mode.

At the top of the dialog there are check boxes to turn on frame overdrawing and to enable 3D overdrawing. There is another check box at the bottom of the dialog which causes immediate display updates as you change fields in the dialog; very useful for getting a display just right.

*Frame selection*
The next items select the data frames to be overdrawn. Set the frames either by entering a list of frame numbers directly or by using the drop-down list to select a category of frames from: All frames, Current frame, Buffer, Tagged frames, Un-tagged frames, Frame state = xxx and Last n frames. A subsidiary field below allows a subset of these frames to be defined.
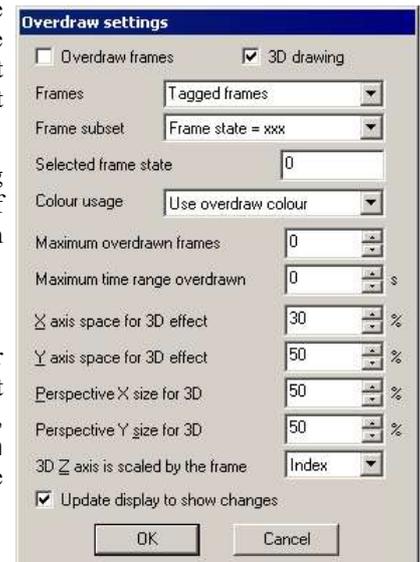
When used during sampling an extra Sampled frames option is available; this enables a minimum-delay overdrawing mode where the previously sampled frame data is not erased and new data is simply drawn on top of it. See **Overdrawing online** below for more about this mode and overdrawing online in general.

If you select Frame state = xxx the Selected frame state field appears, use this to select the state number to be displayed. Similarly if you select Last n frames an extra field appears for the number of frames wanted, this selects the number of frames before the current frame when used offline, and the number of frames most recently filed to disk when used online.

The frame list is dynamic so that, for example, if you request all tagged frames and subsequently tag a frame it will be added to the overdrawn frames.

*Colour usage*
This item selects the colours used for overdrawn frames. You can choose between the following options:

| | |
|---|---|
| Use overdraw colour | Draw all overdrawn (non-current) frames using the Overdraw colour defined in the colour settings. |
| Use colour cycling | Draw overdrawn frames using the Signal colour cycling table. Each frame is drawn using the next colour in the table, wrapping round when the end of the table is reached. |
| Draw at half intensity | Draw overdrawn frames using a colour halfway between the channel trace colour and the channel background. |
| No colour variation | Draw overdrawn frames using the same colour as the current frame – the standard trace colour. |
| Fade to background | Draw overdrawn frames in a colour that fades from the trace colour to the channel background colour. |
| Fade to overdraw colour | Draw overdrawn frames in a colour that fades from the trace colour to the defined overdraw colour. |
| Fade to secondary colour | Draw overdrawn frames in a colour that fades from the trace colour to the channel secondary colour. |

| Colour by frame state | Draw overdrawn frames using a colour taken from the colour cycling table, each frame is drawn using a table colour indexed by the frame state, wrapping round the end of the table as necessary. |
|---|---|

*Frame limits*

The next two items, labelled Maximum overdrawn frames and Maximum time range overdrawn, can be used to set limits to prevent overdrawing from taking too long. Leave these fields set to zero if you do not want to apply such limits. If applied, these limits will restrict overdrawing to those frames closest in number or frame start time to the current frame.

**3-D overdrawing**

If you enable 3D overdrawing using the check box at the top of the dialog, data is drawn as a 3-dimensional 'stack' of frames, with the current frame at the front and older (lower-numbered) frames behind (you cannot show frames after the current frame – we found it made things impossibly confusing). To create the 3D effect, each frame is drawn inside a rectangle, and the position and size of the rectangle depends on a notional z axis. The z axis can be based on the frame's position in the frame display list, the frame number or the trigger time of the frame.
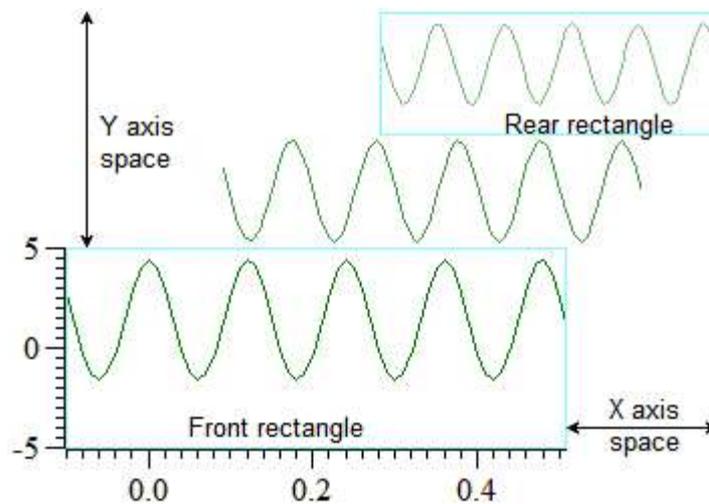


*3D drawing rectangles*

In this example, there are three frames of data. The Front rectangle, used for the current frame, is always positioned at the bottom left of the channel area. The Rear rectangle, used for the oldest (lowest-numbered), is always positioned at the top right of the channel area. The middle frame also has a rectangle that is calculated by a linear interpolation between the front and rear rectangles based on the z axis position of the frame.

The dialog fields concerned with 3-D overdrawing control the positioning of the front and rear rectangles:

*X axis space for 3D effect*

This field sets the percentage of the width of the channel area to use for the 3D effect. The larger the value, the smaller the width of the front rectangle. Set this and the Perspective X size fields to 0 to make all frames draw vertically aligned.

*Y axis space for 3D effect*

This field sets the percentage of the entire view vertical space to share between all the channels to generate the 3D effect. All channels are given exactly the same space so that the 3D effect is the same for all channels. The larger the value, the smaller the height of the front rectangle.

*Perspective X size*

This sets the width of the rear rectangle as a percentage of the width of the front rectangle in the range 0 to 100. Setting a value less than 100 gives a perspective effect.

*Perspective Y size*

This sets the height of the rear rectangle as a percentage of the height of the front rectangle in the range 0 to 100. Setting a value less than 100 gives a perspective effect.

*Z axis is scaled by the frame*

The position of each frame of data (between the Front rectangle and the Rear rectangle) can be set by the frame index within the display list (giving equally spaced frames), the frame number within the file, or by the frame trigger time (as shown for the current frame in the status bar). Choose from Index, Number or Time.

**Notes**

The 3D display mode behaves in exactly the same way as ordinary overdrawing except that the overdrawn frames are offset and scaled. There is one other difference; only frames from earlier in the file than the current frame will be overdrawn, ensuring that the current frame is at the front and the oldest overdrawn frame is at the back.

The 3D display mode makes no difference to any measurements you may make with cursors, the X and Y axes that are displayed match the current frame that is displayed at the front.

**Overdrawing online**

When 3D overdrawing is used on data while it is being sampled the special frame zero behaves as if it is after all the other frames in the file – where it will be if it is written to disk.

The Sampled frames option is only available during sampling; it enables special behaviour where the previous sweep's data is not erased at the start of the next sampling sweep but merely redrawn in the overdraw colour so that the display accumulates sampled traces. This overdrawing method is very fast so as to avoid delays during sampling, but it is also rather limited by the way it operates and as a result changes to the X or Y axis ranges (amongst other things) will cause the previously overdrawn data to be erased and overdrawing to start again from an empty display. The Edit menu Clear option can be used to manually clear the view of Sampled frames overdrawn data if required. Sampled frames mode is automatically converted to All frames mode when sampling finishes.

The Last n frames option is renamed Last n frames filed when used online. Overdraw modes that will cause a lot of frame overdrawing (such as All frames) should be avoided online unless you use the frame limits; the time taken to redraw many frames will seriously impact on sampling performance.

# Enlarge view, Reduce view

These commands, the two buttons at the lower left of data windows and the keyboard shortcuts Ctrl+Right and Ctrl+Left expand and contract the displayed x axis area. The enlarge command zooms out by doubling the data region spanned by the x axis. The reduce command zooms in by halving the data region. The left hand edge of the screen is fixed unless the expand operation would display data beyond the end of the frame, in which case the start of the displayed area is moved backwards. If the result of expanding would display more data than exists in the frame, all of the frame data is displayed.

# X Axis range

This command, double-clicking the x axis of a data or XY view and the shortcut key Ctrl+X open the X Axis range dialog, which sets the region of the view to display. The dialog also gives you the option of setting the X axis style and tick spacing.

### Left, Right and Width

The Left and Right fields set the window start and end using the preferred time units. You can type in new positions or select values from a drop down list. Each drop down list contains the initial field value, cursor positions, the minimum and maximum allowed values and the left and right edges of the window (XLow() and XHigh()). The Width field sets the window width if the box is checked. Click the Draw button to apply changes without closing the dialog.

Show All expands the x axis to display all the data and closes the dialog. Cancel undoes changes made with the dialog and closes it. Close accepts any changes and closes the dialog.

### Tick spacing

Normally, you will let Signal organise the axis style; it tries to adapt to the displayed time range to give you an axis that looks decent. However, when preparing data for publication you may wish to have more control. You can control the Large tick spacing (this also sets the scale bar size) and the number of Tick subdivisions by ticking the boxes. Your settings are ignored if they would produce an illegible axis. Changes to these fields take effect
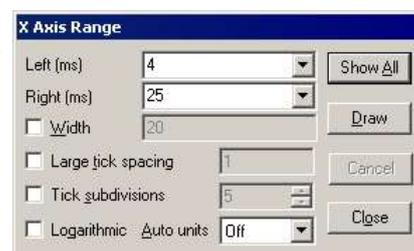
immediately; there is no need to use the Draw button. If you prefer a scale bar to an axis, you can select this in the Customise display dialog.

### Logarithmic

Checking the Logarithmic option will switch the axis from linear to logarithmic; modifying the displayed range only if it included negative values. With a logarithmic axis, the large tick spacing value is now a ratio between tick values, so if it is set to 2 each large tick will be at a value twice the size of the previous value, 10 gives you a power-of-10 axis. In logarithmic mode the Auto units selector vanishes and another check box: Show powers will appear, this allows the big ticks to be labelled with powers of the big tick spacing rather than the simple values. As with the tick spacing options these changes take place immediately with no need to press Draw.

## Auto units

The Auto units field is present when the Logarithmic box is not checked. There are two options:

Off   The axis behaves normally, using the standard X axis units.

x1000s  If the axis range becomes numerically very large or very small and scaling makes sense, the axis units are multiplied by a power of a thousand and the X axis units are displayed with the power of ten after the units.

Any change made to an axis by this field is purely cosmetic; it has no effect on the values seen by Signal or by a script. It changes the visible axis values and the values displayed on screen by cursors and by horizontal measurements made by holding down the Alt key and clicking and dragging in a channel.

### Script language equivalents to this dialog

The XAxisAttrib() script function is the equivalent of the Logarithmic and Show Powers check boxes and the Auto units selector. The XAxisStyle() function sets the Units and the tick spacings and number of subdivisions of major ticks. The XRange() and Draw() functions set the displayed range. You can show and hide axis features with XAxisMode().

### Short cut keys

Short cut keys that control the X axis are: Left and Right arrow scroll the axis left and right by half the screen width and Ctrl+Left and Ctrl+Right decrease and increase the X axis width respectively. The mouse wheel will also scroll the X axis one pixel at a time; add Shift or Ctrl to scroll by larger numbers of pixels.

# Y Axis range

This command, double-clicking a y axis or using the `Ctrl+Y` shortcut open the Y Axis range dialog. This sets the Y axis range and style for data or XY view channels. The Channel field chooses one, all, all visible or the selected channels. If more than one channel is selected, the settings shown are from the first channel.

### Optimise and Show All
Optimise draws the visible data scaled to fill the display, if the channel(s) in question are part of a group of overdrawn channels with locked axes then all of the grouped channels are used to determine the optimal Y axis range. Show All sets the y axis to display the maximum possible range for waveform channels, for channels without a range limit the available data range is shown. Both buttons close the dialog. To optimise without opening the dialog, right-click a channel and select the optimise option from the context menu.
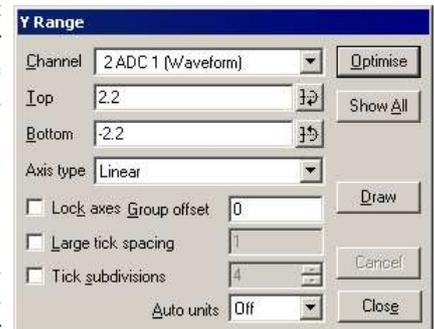
### Top and Bottom
The Top and Bottom fields are used to directly enter the limits to the displayed data. The buttons to the right of these fields are used to set up a symmetrical axis - clicking on one copies the current setting, negated, into the other field.

### Axis type
The axis type can be Linear, Logarithmic or Square root. The values shown on the axis are equally spaced for a Linear axis, have a constant ratio (the large tick spacing) between adjacent values for a Logarithmic axis (so the log of the axis values are equally spaced) and for a Square root axis the large tick spacing sets the difference between the square root of the values shown on the axis.

### Lock Axes and Group offset
The Lock axes and Group offset controls are only visible when the chosen channel shares a y axis with other channels, and only enabled when the channel is the first in the group. If you check Lock axes, the grouped channels not only share the same space, they also share the y axis of the first channel in the group. The Group offset field sets a per-channel vertical display offset for each locked channel to space out channels within the Y axis.

### Tick spacing
Normally, you will let Signal organise the axis style; it tries to adapt to the displayed time range to give you an axis that looks decent. However, when preparing data for publication you may wish to have more control. You can control the Large tick spacing (which also sets the scale bar size) and the number of Tick subdivisions by ticking the boxes. Your settings are ignored if they would produce an illegible axis. Changes to these fields take effect immediately; there is no need to use the Draw button. If you prefer a scale bar to an axis, you can select this in the Customise display dialog.

## Auto units
The Auto units field is present when the Logarithmic box is not checked. There are three options:

Off    The axis behaves normally, using the standard Y axis units.

x1000s If the axis range becomes numerically very large or very small and scaling makes sense, the axis units are multiplied by a power of a thousand and the Y axis units are displayed with the power of ten after the units.

SI prefix The same as x1000s, except that the y axis units are prefixed by a SI scaling prefix (M, k, m, u, ...). If the units already start with an SI prefix, this is removed (and taken into account) before display.

Any change made to an axis by this field is purely cosmetic; it has no effect on the values seen by Signal or by a script. It changes the visible axis values and the values displayed on screen by horizontal cursors and by vertical measurements made by holding down the `Alt` key and clicking and dragging in a channel.

Draw applies changes to the Top, Bottom, Lock axes and Group offset fields. The axis type can be Linear, Logarithmic or Square root. The Auto units selector can be used to select between the units displayed on the axis

being static, switching to multiples of powers of 1000 in order to keep the figures sensible when zoomed well in or well out, or you can select the SI Prefix option which does the same thing by changing the units using SI prefixes, for example by converting `V` to `mV` when you are zoomed in.

Cancel undoes any changes and closes the dialog. Close accepts changes and closes the dialog.

### Script language equivalents to this dialog

The `YAxisAttrib()` script function is the equivalent of the Axis type and the Auto units selectors. The `YAxisStyle()` function sets the Units and the tick spacings and number of subdivisions of major ticks. The `YAxisLock()` function controls locked axes and the group offset. The `YRange()` function sets the displayed range. You can show and hide axis features with `YAxisMode()`.

### Short cut keys

Short cut keys that control the Y axis all operate on all selected or visible channels. The keys are: Up and Down arrow scroll the axis up and down and Ctrl+Up and Ctrl+Down decrease and increase the axis range respectively. Home shows all the Y range, while End optimises the Y range.

## Autoscale Axes

This option is for XY views only and opens the XY view Autoscale dialog. This dialog determines how the x and y axes of an XY view behave when new data points are added to the XY view. You can use this to have the XY view automatically track your data, displaying either all of it, or concentrating on the most recently added points.

There are two common methods used to add data points to XY views:

1. Using the Analysis menu Measurements->XY view command.

2. Adding points using the script language.

If you are adding data using the Measurements to XY view command, the associated Process dialog has its own method for Optimising the axes, which will take precedence over any automatic scaling you set in this dialog. If you enable both methods, the Process dialog optimise will win, and the time spent calculating the Autoscale will be wasted.
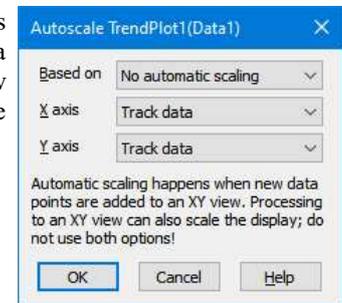
### What does Autoscale do?

An XY view holds multiple channels of data. Each channel is a list of (x,y) data points that can be drawn in a variety of ways. From the script language you can both add data to a channel and delete data points from a channel. The Measurements to XY view process can also add (and remove) data points.

Each time data is added to an XY view from whatever source, Signal gives the XY view the opportunity to rescale the x and y axes. This routine knows the both the co-ordinate rectangle that encloses all the visible data points (including those just added), and the screen rectangle that encloses the points that were added (usually a smaller rectangle, though it could be the same). The XY view rescales the axes based on the settings you make in this dialog.

The script equivalent of this dialog is `XYDrawMode(-1, 5, autoScale%);` you can record any changes you make in this dialog as a script.

### Based on

This field determines which set of data points are passed to the x and y axis optimise routines. You have the following options:

| | |
|---|---|
| No automatic scaling | This turns off automatic scaling. The x and y axes are not changed by adding data to the XY view. |
| New points + visible Y | The screen rectangle used for optimising holds the full range of visible data points in the y direction, but only the newly added points in the x direction. |
| New points + visible X | The screen rectangle used for optimising holds the full range of visible points in the x direction, but only the newly added points in the y direction. |
| All visible points | All the visible points in the XY view (added and existing points). |

New points only            Only the points that were added this time.

The most common settings will be No automatic scaling and All visible points. To use the other settings you will usually have an axis setting that has a fixed size, or that has a minimum size.

**X axis, Y axis**

These two fields have identical options and determine how the x and y axes change using the x and y dimensions of the rectangle generated by the Based on field. You can choose from:

Expand only                If any point in the data passed is not visible, the axis range is increased so that it is visible.

Track data                 The axis is set to display the entire range of the data selected by the Based on field. If the range has no size (for example there is only one point), the axis scale is unchanged and the axis is centred on the new point. You would normally use this with All visible points mode.

Fixed size following data   The length of the displayed axis is saved the very first time data is added and the axis is kept at this length. The axis is shifted so that the mid point of the data selected by Based on is in the middle of the axis. You can use this with the New points options to track the most recently added data.

Track with zoom in limit    The length of the displayed axis is saved the very first time data is added. This then behaves as the Track data option except that the range of the axis is not allowed to be less than the saved size.

No automatic scaling        The axis is not changed by the data.

**Cancel and OK**

Changes made in this dialog are applied immediately you make them so that you can see the result. If you use the Cancel button, any changes made are removed. If you use OK and have made any change, the original state is added to the Undo queue and if you are recording your actions, the changes are recorded.

# File Information

This command (shortcut `Ctrl+I`) displays information about a data document. The dialog has two pages, one showing the overall file information and one showing information about the currently displayed frame. Each shows a selection of useful information plus a list of either the file or frame variables, for each giving the variable number, type, name, value and units. Not all of the file or frame variables are displayed in this dialog so you will see gaps in the variable numbers, the variables that are not shown are either reserved for internal Signal use or unused.

| Information for memb_test.cfs, frame 7 | | | | |
|---|---|---|---|---|

File information | Frame information

| | |
|---|---|
| File name | C:\Users\Tim\Documents\Signal6\memb_test.cfs |
| Created | 20 May 2015 at 17:48:15 |
| Size | 574 KBytes |
| Frames | 7 |
| Channels | 3 (2 waveform, 1 marker class) |
| Data sources | 3 channels in CFS file, 0 memory, 0 virtual |
| Errors | 0 channels have errors |
| Comments | \<blank> |
| | \<blank> |
| | \<blank> |
| | \<blank> |
| | \<blank> |
| X axis range | From 0 to 1 s over all frames |
| Clamping | 1 clamping set detected |
| Variables | 91 (25 file, 66 frame) |

File variables

| # | Type | Name | Value | Units |
|---|---|---|---|---|
| 0 | Int2 | CED Signal Program | 603 | Signal |

# Channel Information

Use this command to view and edit data view channel information for the current frame. You can open it by double-clicking a channel title, from the View menu or by right-clicking the channel to open the context menu. You can edit the Title of the channel set by the Channel field. The remaining fields are hidden or displayed depending on the channel type. For waveform channels the Units may be changed, and the Full scale and Zero values may also be edited. These last two are the same as the values set in the sampling configuration when the file was sampled originally and can be changed to re-calibrate the data. For real marker channels the Units may be changed. For an idealised trace channel the Units may also be changed as well as the −3 dB frequency used for drawing the trace convolution and fitting the trace to the raw data. The Reset, Apply and OK buttons are disabled until you make a change to one of these fields. The Close button closes the dialog and does not apply any changes.
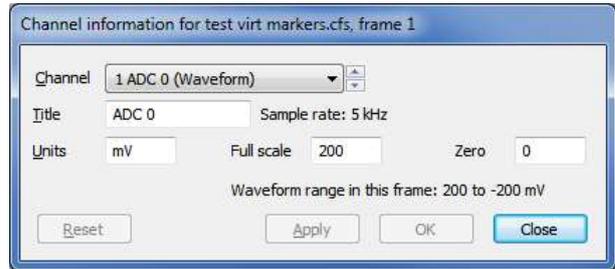
# Channel Information (XY view)

Use this dialog to modify the XY channel titles and the X and Y axis titles and units. You can open the dialog from the View menu, by double-clicking on the axis title and units area or from the XY view context menu. The OK button is disabled until you make a change. Changes are not applied until you click OK, at which point all the changes you have made are applied. If you click the Cancel button, no changes are applied. Changes you make in this dialog can be recorded and can be undone.
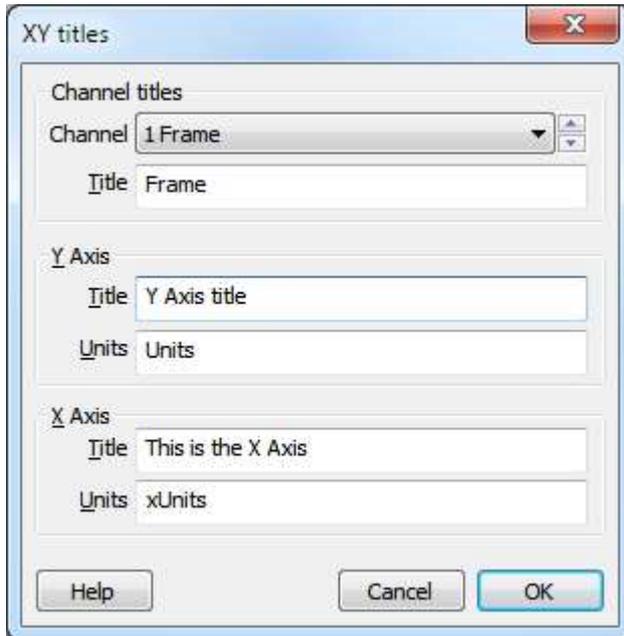
### Channel titles

Each channel in the XY view can be given a title. If a channel does not have a title, it is listed as Channel n, where n is the channel number.  You can select the channel from the drop down list, or by using the spinner control. All changes you make are saved in the dialog until you click OK or press the Enter key. We allow you to enter up to 50 characters.

### Y Axis

An XY view draws multiple traces using the same set of x and y axes. The displayed y axis title and units do not belong to any particular channel, so (unlike Time and Result views), the y axis title and and units are independent of the channels. You can set titles and units of up to 50 characters, though no more than 10 will display. When a

result view is saved, the text strings for the x and y axis titles and units have to fit in a restricted space (for reasons of backwards compatibiity). There is sufficient space for 50 ASCII characters, but if you use non-ASCII characters you will have room for fewer. For example, there is typically space for 17 Japanese or Chinese characters.

**X Axis**

You have a free choice of text for the x axis title and units in an XY view. The same comments about the length of the text apply as for the Y Axis.

# Info windows

Use the Create New Window sub-command to this command to create a new information window that can display text, times and other information using a very large font for extra visibility. The newly created information window is attached to the current time, memory or XY view at the time that the command is used; where necessary it takes information from its attached parent view. Other sub-commands can be used to select an already existing information window.

The text displayed in the information window is generated using a string entered by the user using a settings dialog, with special text fields specifying values that will be taken from the parent window. Any percent symbols (%) in the text string are interpreted along with the following character(s) and replaced with measurement values taken from the attached parent window. For an attached time window the available interpreted text items are:

| | |
|---|---|
| \n | This item acts as a 'new line' indicator, causing any further text to be displayed on the next line in the window. Note that, unlike all of the other special fields, this does not use the percent symbol. |
| %l | This item is replaced by the local time taken directly from the system clock, displayed as HH:MM:SS. The clock can be started and stopped using the Start/Stop button. Note when re-started it will not resume counting from whatever time was reached but rather will update to the current local time. |
| %f | This item is replaced by the number of the frame currently being displayed. |
| %m | This item is replaced by the maximum frame number in the attached parent window. |
| %t | This item is replaced by the current timer value. There are a number of settings controlling how the timer behaves and its value displayed; see below. |
| %s | This item is replaced by the state number for the frame currently being displayed. |

For an attached memory view all of the above items are available, in addition there is:

| | |
|---|---|
| %w | This item is replaced by the number of sweeps that have been processed to produce the frame currently being displayed. |

With an attached the XY view the settings dialog is the same but a subsidiary dialog can be used to generate special strings starting with a percent sign that select XY values from the view. In this dialog the Channel number field selects the XY view channel number, Point number sets the XY point to use and the X coordinate and Y coordinate radio buttons choose between the X and Y part of the selected XY point. You can either enter a specific point number in the Point number field or choose between Latest, Second latest or Difference (between the latest and next to latest points). If you enter a point number that's too large, the dialog will refuse to accept it. When you press OK, the corresponding text string is pasted into the text settings dialog.
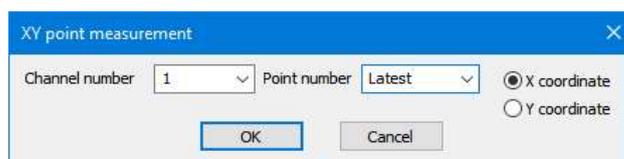
The strings that select an XY point measurement look like %c1px1 – where the number after the c is the channel number, the character after the p is either x or y depending on whether we want the X or Y coordinate value, and the digit after that is the point number.

When used to select the Latest, Second latest and Difference point measurements the strings look like this:

%rx1 - where the r (for recent) selects the latest point, the x (or y) selects the X (or Y) coordinate value and the number at the end is the channel number.

%px1 - where the p (for previous) selects the second latest point, the x (or y) selects the X (or Y) coordinate value and the number at the end is the channel number.

%dx1 - where the d selects the difference, the x (or y) selects the X (or Y) coordinate value and the number at the end is the channel number.

### Settings dialog and timer controls

The info window settings dialog is used to enter the text that controls what is displayed. In addition there are a number of settings controlling how the timer value is displayed and how the timer behaves. The timer can be reset manually (with the Reset button) or automatically during sampling when a new sweep starts (for a file view parent) or when new data is processed into the memory view (for memory or XY view parents). The timer can be started and stopped using the Start/Stop button, or automatically on a reset. The timer controls available in the dialog are:

| | |
|---|---|
| Reset time (s) | This field sets the time in seconds to which the timer will be set when a reset occurs. |
| Start on reset | This checkbox controls if the timer is automatically started (if stopped) when a reset occurs. |
| Count time downwards | This checkbox controls the direction in which the timer counts. |
| Stop at (s) | This checkbox, if set, allows you to set a time at which the timer will automatically stop, the field to the right of the checkbox sets the time (in seconds) at which the timer stops. |
| Update every | Use this selector to choose how often the displayed time is updated and the smallest unit of time that is shown. You can choose between every 0.1 second, Second, Minute or Hour. so, for example, if you update every 0.1 seconds then the time that is displayed will always show tenths of a second, if you update every minute then the seconds value is never shown. |
| Show time as | Use this selector to choose how the time is displayed. You can choose between 0.1 seconds (a simple count of 100 millisecond intervals), Seconds (a simple count of seconds), MM:SS and HH:MM:SS. Note that the Update every selector also affects how the time is displayed - parts of the time display that are shorter than the Update every interval will not be shown. |

The dialog will display a red error message if any of these values contradict each other (e.g. if we're counting down from 0 but want to stop at 1) and the OK and Apply buttons will be greyed out to prevent these settings being used. Entering non-numerical or nonsensical values in the Reset time(s)  and Stop at (s) fields will also display a red error message and refuse to let you accept the settings.

### Using the information window

The information window can be enlarged by dragging the bottom-right corner of the window, the font size will be adjusted according to the window size. Other options are provided by a menu obtained by right-clicking on the window, the menu commands available are:

| | |
|---|---|
| Copy | This command copies the currently displayed information window text to the clipboard. Note that the text does not need to be selected; all of the displayed text is copied. |
| Close | This command closes the information window (handy if the window title is hidden). |
| Start (or Stop) | This command starts (or stops) the clock or timer in the same manner as the Start/Stop button at the top of the window. |

| | |
|---|---|
| Reset | This command resets the or timer in the same manner as the Reset button at the top of the window. |
| Settings ... | This command displays the settings dialog for the information window. |
| Colours... | This command displays the Signal colour selection dialog in a special mode to allow you to change the colours used for this specific information window, the main colour selection dialog can also be used to set the colours for all information windows or all those associated with the current parent view. |
| Font... | This command displays the Signal font selection dialog to allow you to change the font used in the information window. Note that the font size is set automatically according to the size of the window. |
| Image... | This command displays a dialog allowing you to set an image that will be displayed in the information window. This image is displayed 'on top' of the information window background and below the text, the image opacity can be adjusted using the Opacity slider in the dialog while the Browse... button is used to browse the hard disk to look for the image. |
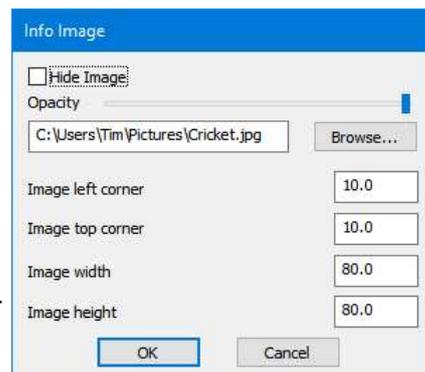
The values for the image left and top corners and for the width and height set the position of the image within the background area of the info window. All of these values are a percentage of the information window height and width, from 0.0 to 100.0.

| | |
|---|---|
| Hide (or Show) title | This command hides (or shows) the window title at the top of the information window, when the title is hidden the title bar shrinks to a thin bar. |
| Hide (or Show) buttons | This command hides (or shows) the standard buttons at the top of the information window. |
| Lock (or Unlock) font size | This command locks (or unlocks) the size of the font so that it does not change when the window size changes. |

# Experimenter's notebook

This command (shortcut Ctrl+A) displays the experimenter's notebook for a data document. The experimenter's notebook is a collection of timed and dated notes that describe the sampling configuration settings used to collect a file, the initial pulse outputs and dynamic clamp model settings, changes made to the pulse outputs and dynamic clamping models during sampling, the application of digital filters to the file and the use of the Multiple frames dialog to modify the file data. Other notes indicate occurrences during sampling such as turning writing to disk on or off. All of these items are automatically generated by Signal to record user actions,
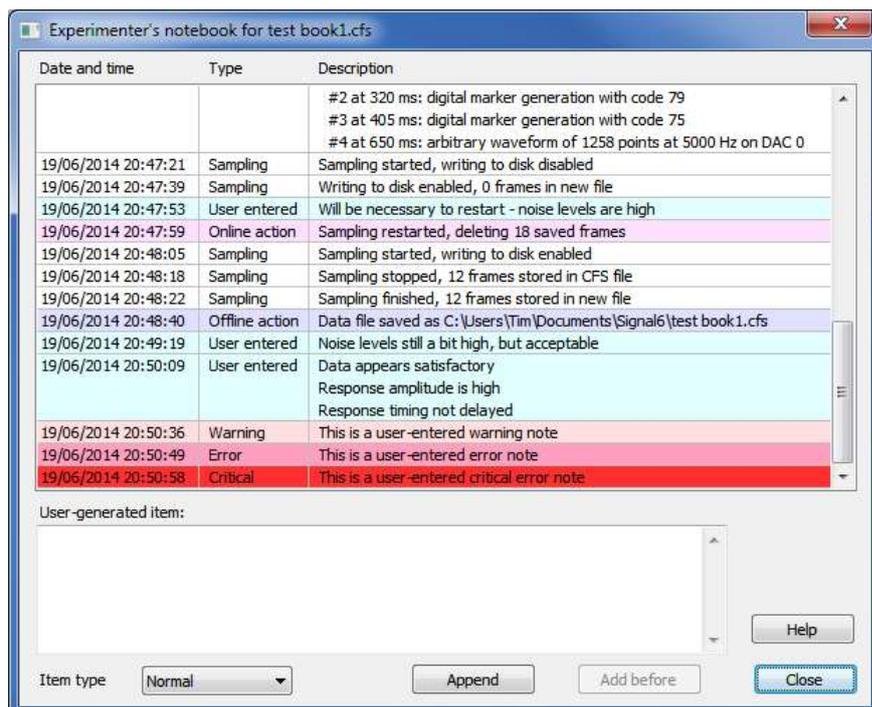
users can also add their own
notes to the notebook both during sampling and afterwards.

The notebook information is held in the .sgrx resources file for each CFS data file so you should take care to keep these two files together if you do not want to lose the notebook information. Versions of Signal before 6.02 did not generate notebook information so files created with these earlier versions will not have information available on the sampling configuration used and other online actions, but any relevant offline actions carried out using later versions of Signal will be recorded.

The notebook display shows the notes as a list, notes are shown along with the time and date on which they were generated and the type of note. Different types of note are drawn using different background colours to help with recognition. The notebook display is 'modeless', which means that you can make full use of Signal while the notebook is being shown, the notebook display will always stay in front of the main Signal windows. Notebook items can be selected and deselected by clicking on them, double-clicking on a note that was generated generated during sampling will cause the data view to go to the frame sampled while or immediately after the time that the note was generated.

Below the list of notebook items are controls used to add user-generated notes to the notebook. The main text area is used to type the note text, below this is a selector for the note type, you can choose between Normal, Warning, Error and Critical types. Warning, Error and Critical notes are drawn using increasingly strong background reds to indicate the presumed increasingly severe nature of the problems being described. The Append button creates a new note using the selected text and severity at the bottom of the notebook. The Add before button adds a new note before the currently selected item, it is only enabled when a single note is selected and sampling is in progress.

Right-clicking inside the notebook window provides a pop-up menu holding additional commands. The first two commands are Add before and Append note, these carry out the same actions as the Append and Add before buttons. The other commands available are:

*Copy*
This command copies all selected notebook items (or all the notebook items if none are selected) to the clipboard as text.

*Spreadsheet copy*
This command is the same as the Copy command, but the text fields are inside quotes to make it easier to paste them into a spreadsheet.

*Select All*
This command selects all of the notebook items. If some notebook items are already selected it changes to Deselect All.

*Print*
This command prints all selected notebook items or all of the notebook items if none are selected.

# Standard Display

This command sets the current data, memory, XY or text view to a standard state. In file and memory views it turns on all channels in their standard display mode and size, ordered as set in the Display Preferences, all special channel colours are reset, x and y axes turned on and all special axis modes and grids off. In an XY view, all channels are made visible, the point display mode is set to dot at the standard size, the points are joined and the x and y axis range is set to span the range of the data.

In a text-based view it removes any maximum line limit except in the Log view, where it applies the line limit set in the Script page in the Preference dialog. It also hides line numbers, displays the gutter and the folding margin (for views that support folding). All the text styles are set to the default values (equivalent to opening the Font dialog and using Reset All) and any zooming is removed.

# Customise display

This command opens a dialog that controls the channels to display in a data or XY view, the display of x and y axes, channel numbers, grids and of the horizontal scroll bar, plus controls for horizontal Y axis labels.

### Channel numbers, Y Axis, X Axis

The three check boxes at the top of the dialog control if channel numbers and the x and y axes are drawn. Channel numbers are never drawn in an XY view as all channels share the same y axis.

### Channel list, All On, All Off

The channel list contains all the channels that exist in the view. The check boxes in the list are ticked if the channel is visible. You can change the state of the check boxes or use the All On and All Off buttons to set the state of all the channels.
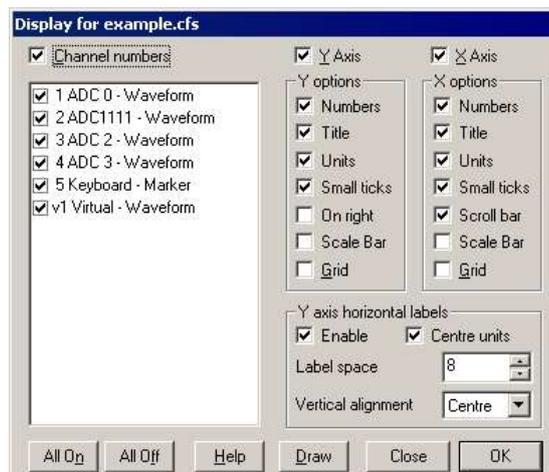
### Draw

Most changes made in this dialog will require the view to redraw, which can take a noticeable time. Because of this, changes are not applied until you click OK or you can apply changes without closing the dialog with the Draw button.

### Y Axis, X Axis

You also have control over the x and y axis appearance. You can hide or display the grid, numbers on the axes, the big and small ticks and the axis title and axis units. You can also choose to show the y axis on the right of the data, rather than on the left and choose if the Scroll bar is visible for the x axis (not in an XY view).

### Scale bar

For publication purposes, it is sometimes preferable to display axes as a scale bar. If you check the Scale only box, a scale bar replaces the selected axis and any grid set for that axis will vanish. You can remove the end caps from the scale bar (leaving a line) by clearing the Small ticks check box. The size of the tick bar can set by the Large tick spacing option in the Y Axis Range or X Axis Range dialogs, or you can let Signal choose a suitable size for you.

A y axis will automatically switch to draw as a scale bar if there is not enough vertical space to display numbers for the tick marks.
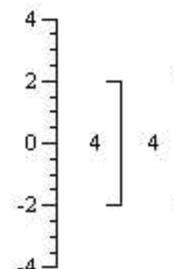
### Y axis horizontal labels

Normally, the title and units text for the y axis is presented vertically unless there is very little vertical space available. If you check Enable, the y axis text is presented horizontally and the other controls in this region become active. The channel title is left aligned when the axis is on the left and right aligned when the axis is on the right. The script language equivalent of this dialog section is YAxisMode(). The other controls are:

*Centre units*
If checked, the units are centred on the channel title. If clear, the units are aligned to match the title.

*Label space*
You can set the horizontal character space to reserve for the channel title and units. This is in terms of a representative character width.

*Vertical alignment*
You can choose to position the text at the top, centre or bottom of the available vertical space.

### Interaction with use of YAxisMode() and XAxisMode()

The YAxisMode() and XAxisMode() script functions can also be used to customise the appearance of the axes. However these functions are also able to disable drawing of the axis large ticks and the line which joins these,

which is not possible using this dialog. Therefore, after using these functions to disable large ticks you will find that this dialog does not recognise that this has been done and any use of the dialog to change axes will also turn on large tick drawing once again.

**Short cut to display a single channel or channel group**

In a data view you can double-click on a channel with a y axis to hide all the other channels; the channel expands to fill the entire window. A second double-click restores the display. If the clicked channel holds a group of channels, all channels in the group expand to use the display area.

# Draw mode

The Draw mode dialog sets the display mode for channels. You can set the mode for a single channel, all channels or any subset of the channels.

The Channel field sets the channels to change. The next field sets the draw mode to use. Click Draw to change the draw mode for the selected channels without closing the dialog, click OK to change the draw mode and close the dialog.
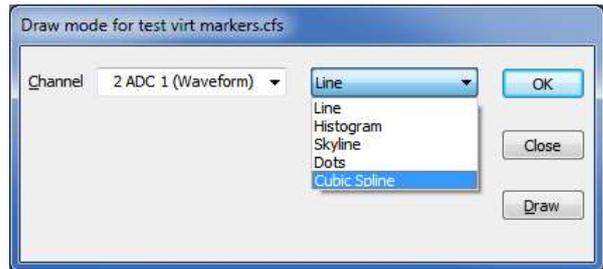
There are a number of other fields in the dialog, these vary with the type of channel and the draw mode that is currentyly selected.

# Waveform draw modes

There are five drawing modes available for waveform channels: Line, Histogram, Skyline, Dots and Cubic Spline. Line mode joins the waveform data points with straight lines. Histogram mode draws the data as a histogram, one bin per data point and the bin height set by the data value. Skyline joints data points with horizontal and vertical lines. Dots draws a dot at each data point, you can choose the dot size (small dots can be very difficult to see on some displays), the size goes up in steps set by the pen width settings for that channel with zero selecting the smallest possible dot: one pixel. Cubic Spline mode joins the points with smooth cubic curves based on the assumption that the first and second derivatives of the data are continuous at the data points.
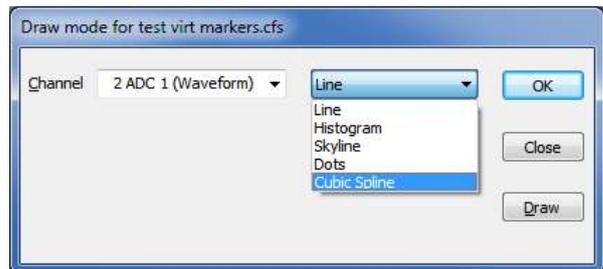
If your view has associated error information, for example a waveform average with error bars enabled, you can set the error display mode as: None, 1 SEM, 2 SEM or SD. Error bars only have meaning if the data points that contribute to the average have a normal distribution about the mean. Given this, 1 SEM shows ±1 standard error of the mean, 2 SEM is ±2 standard errors of the mean and SD is ± 1 standard deviation. If each point of your data can be modeled as a constant "real" value to which is added normally distributed noise with zero mean then you would expect the measured mean value to lie within 1 standard error of the mean (SEM) 68% of the time, or within 2 SEM 95% of the time. The standard deviation represents the width of the normal distribution of the underlying data at each data point.

# Marker draw modes
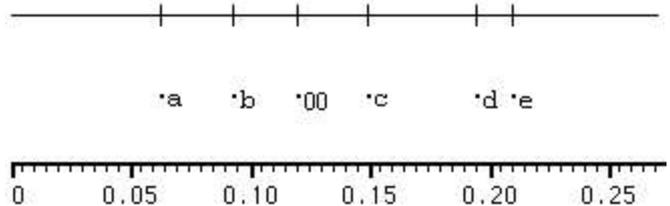
There are three drawing modes available for marker channels; Dots, Lines and Rate. The picture at the right shows the result of Lines and Dots modes.

The simplest method is to draw the marker channel as dots. You can choose the dot size (small dots can be very difficult to see on some displays), the size goes up in steps set by the pen width settings for that channel with zero selecting the smallest possible dot: one pixel.

You can select which marker code byte is displayed with the Use marker code selector, normally marker code byte index 0 is shown but you can select any of the four code bytes. You can also choose to display the marker code values as single characters (where the code value is an ASCII displayable character) or as Hexadecimal only (two base 16 digits) regardless of the code value.

The Show colours for marker codes check box selects drawing the dot using a colour selected by the marker code, the colours used are the overdraw cycling colours as set in the View menu Choose colours dialog. If the check box is clear the dot is drawn in the same colour as the marker codes as set in the colours dialog.

You can also select Lines in place of Dots, which draws a small vertical line at the position of each marker. The picture to the right shows the result of both types of display. If you select lines mode the display of marker values is suppressed. The Centre line check box selects drawing of a horizontal line through the centre of the vertical marker lines. The Show colours for marker codes check box selects drawing the vertical lines using a colour selected by the marker code as for Dots.

The Rate display mode counts how many markers fall in each time period and displays the result as a histogram. The result is not divided by the width of the bin, so strictly speaking it is a count histogram, not a rate. This form of display is especially useful when the marker rate before an operation is to be compared with the rate afterwards.
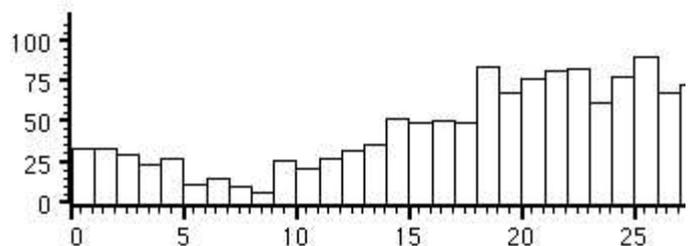
# Real marker draw modes

There are four drawing modes available for real marker channels; Dots, Lines, Rate and Waveform, the first three of these are the same as the normal marker draw modes and have the same options available.

The real marker Waveform drawing mode is rather different. Here the marker channel has a Y axis and each marker is drawn at a position set by the marker time (X) and the marker real value (Y). If the real marker channel has more than one value per data item then the real value that is used can be selected using the Data index field, data indexes start at 0 for the first real value.
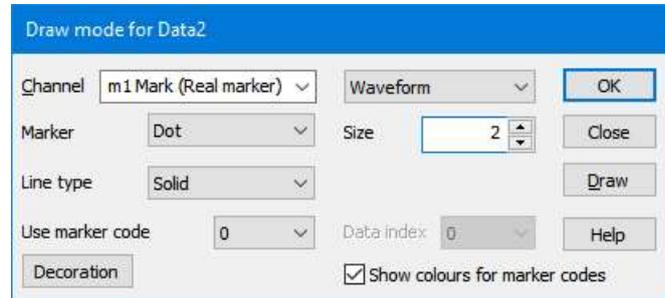
The Marker field sets the symbol drawn at each marker point, you can choose between None, Dot, Box, Plus +, Cross x, Circle o, Triangle, Diamond, Horizontal or Vertical, the Size field sets the marker size in steps set by the pen width for the channel. Similarly, the Line type field sets the type of line connecting the marker points, you can choose between None, Solid, Dotted and Dashed, the line width is set by the channel pen width.

The Show colours for marker code check box enables drawing of the marker symbols in colours set by the selected marker code, the colours used are the overdraw cycling colours as set in the View menu Choose colours dialog. If the check box is clear the marker symbol is drawn in the same colour as any connecting lines as set in the colours dialog.

### *Decoration*

The Decoration button appears for real marker data drawn as a Waveform only. Click the Decoration button to open the Channel Decoration dialog

# Idealised trace draw modes

There are three drawing modes for idealised traces: Basic, Convolution and Both. Basic mode draws the trace as horizontal lines representing event amplitudes separated by vertical lines for the transitions. Closed states are drawn in a different colour. There is an optional Trace offset to allow the trace to be drawn below or above the raw data when displayed with a shared locked axis.
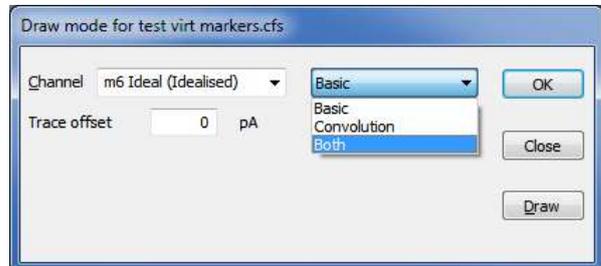
The Convolution is a continuous curve formed by the convolution of the Basic idealised trace and the step response function. The step response function is the error function erf() defined as the integral of the Gaussian function. This is because it is assumed that the raw data was filtered using a close approximation to a Gaussian filter. The cut-off frequency for this filter can be changed by using the channel information dialog. For an idealised trace generated using threshold crossing the cut-off frequency will be set to the Nyquist frequency by default. There is also an optional baseline indicator which will show the level of the baseline for a selected event.

Drawing Both shows both the Basic display and the Convolution at the same time. It is possible to "break" the drawing of an idealised trace if it is taking too long; hit Ctrl+Break and the drawing will be abandoned.

# Channel Decoration dialog

The Decoration button appears in the draw mode dialog for real marker channels that are drawn as a Waveform. Click the button to open this dialog.

A real marker channel holds time markers that each have one or more values associated with them. In a waveform display mode, Signal draws the one of the associated values as a wave (selected in the main Draw Mode dialog Data index field). The Decoration dialog allows you to use other associated values as error bars or to indicate a data range. There are currently three channel decoration modes:
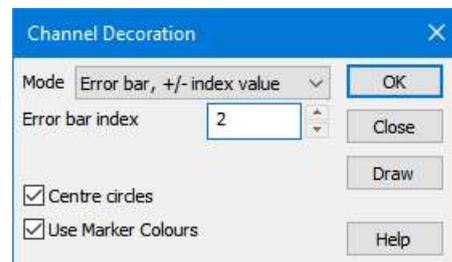
*No decoration*
The channel is displayed without any additional decoration.

*Error bar, +- index value*
You choose one of the associated data values to set the size of an error bar. The bar is drawn from the data value plus the index value to the data value minus the index value.

*Low-high range*
You choose two associated data values to be linked by a vertical bar. These are normally chosen as values that span the low-high range of data, so the waveform data will usually lie between the low and high values, but this is not required.

### Centre circles
If you check this, a circle (filled with the channel background colour) is drawn on the channel data value.

### Use Marker Colours
If you check this box, the bar and any centre circle is drawn in the Marker colour, otherwise the channel secondary colour is used.

### Draw
Update the display to match the current settings.

### Close
Close the dialog.

# Channel Pen Width

The Pen Width dialog in a data or memory view allows you to override the standard pen width set for data in all channels in the Display page in the Preferences dialog. The dialog can be opened as a context menu when right-clicking on a channel, or from the View menu. Changes made with this dialog can be undone and recorded as a script. For XY views, use the XY Draw Mode dialog to set the pen width of a channel.
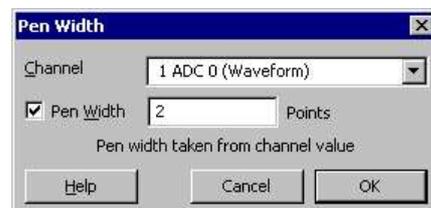
### Channel
You can select a channel from the drop-down list or type in a channel or a list of channels. If more than one channel is set, the displayed information will be for the first channel in the list.

### Pen Width
If you check the box, you can type in a pen width in points and when you click OK, this will be applied to all the channels in the list. If you clear the box, the standard pen width for data that you can set in the Edit menu Preferences Display page will be applied on OK and you will not be able to edit the value. A point is 1/72 of an inch, which is about the same size as a pixel. If you set a size of 0, the pen size will be 1 pixel on all devices (which can be very thin on a printer). You can set a fractional point size, but the output pen will always be at least 1 pixel wide.

# Channel Image

You can set an image file to provide the background to any channel in a data or XY view. This can be very useful in an XY view when the XY view is being used to map activity. For example, when the view is showing where spiking activity occurred in a maze, or when tracking a target. The menu command opens a dialog:

The dialog is used to select a channel and then link the channel to a suitable file on disk. Any channel in the view can be associated with an image. The name of the file appears in the Image file section of the dialog. The image is rendered on top of the channel background colour, but below all other items that are drawn. When channels are overdrawn, the image appears when the background colour of the channel would be drawn. The dialog fields are:
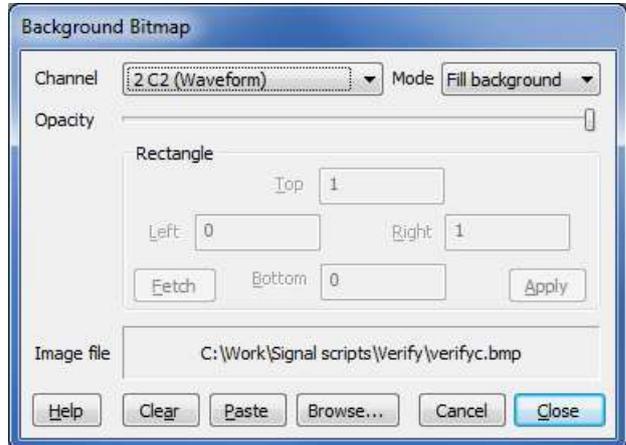
### Channel

You can select any channel in the current view. However, if you are in an XY view, all channels share the background, so the choice of channel makes no difference. When you change channel, any Rectangle region changes that have not been applied will be lost and the current settings for the new channel will appear in the dialog.

### Mode

There are three mode settings: No display, Fill Background and Fill Rectangle.

No display      No image is displayed. This allows you to have an image loaded, ready to display. This will save the time needed to load the image from a disk file.

Fill Background    The image is scaled so that it fills the available channel rectangle. You would use this when you want to display the entire image as a background for the channel.

Fill Rectangle    The image is scaled so that it fills a rectangle defined in channel x and y axis units. You would use this when the image must be aligned with the axes. For example, if the x and y axes represented co-ordinates in a maze, the image could be a picture of the maze.

Changes made to the mode field are applied immediately, so you can see the effect.

### Opacity

This field is a slider control that you can drag to the left to make the image transparent and to the right to make it opaque. Changes made by dragging are applied immediately.

### Rectangle

The fields within this area are disabled unless the mode field is set to Fill Rectangle. The Left, Right, Top and Bottom fields refer to the x axis and y axis co-ordinates of the image. In the current implementation, you cannot invert or reflect the image; if you do it will not appear.

### Fetch

This button sets the Left, Right, Top and Bottom fields to the values that would make the image fill the channel background.

### Apply

This button is enabled when in Fill Rectangle mode and the current rectangle settings are legal number and do not match those of the channel. Click the button to apply the new values.

### Clear

Click this button to clear any image set for the channel. This releases any resources used to hold the bitmap.

## Paste

This button is enabled when the clipboard holds suitable image information, it saves the clipboard information as the background image for the channel, the image file name is set to <CB> to indicate that the background image is taken from the clipboard. The image information is not stored and will not be reliably restored when the file is closed and re-opened, but as the <CB> 'filename' is stored, the background image is refreshed with any suitable data that happens to be on the clipboard when the file is re-opened.

## Browse...

Click this button to open a file dialog in which you can browse for a suitable `.bmp`, `.jpg/jpeg`, `.png` or `.tif/tiff` file.

## Cancel

Click this button to close the dialog and undo any changes made since the last channel change.

## Close

Close the dialog, leaving the associated channel in the current state.

This command is experimental and is implemented in a simplistic manner. If you reuse the same image for multiple channels we do not notice, and store the image multiple times, which could be a problem if you use huge images on many channels.

# XY Draw Mode

This command is used in XY windows to set the drawing style of the XY data channels. Click OK to make changes and close the dialog, click Apply to make changes without closing the dialog. Cancel closes the window and ignores any changes made since the last Apply. The Channel field sets the channel to edit. If you change the channel, the dialog remembers any changes you have made so there is no need to use the Apply button before changing channel unless you want to see the change immediately. The settings available are:

## Join style
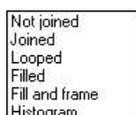
You can set six join styles for a channel. In Not joined style, no lines are drawn between data points. In Joined style, each point is linked to the next by a straight line. In Looped style, the points are joined and the final point is linked back to the first point. In Filled style, the data points are not joined, but they are filled with the XY channel fill colour. In Fill and frame style, the first and last points are joined and linked by a line and the channel is filled. In Histogram style, the XY data is drawn as a histogram with a baseline at zero, each XY point defining the left edge of a histogram bin with the bin extending to the next XY point and the histogram bins filled with the XY channel fill colour. The Line type and Width fields set the kind of line that joins the points in Joined, Fill and Frame and Histogram styles.

## Line type and Width

These two fields set the type of line used to join data points. The Width field determines how wide the line is, in units of half the data line width set in the Display page of the Preferences dialog. If you set a Width of other than 1, the Line type field is ignored and a solid line is drawn.

## Marker and Size

The Size field sets how far, in points, the markers extend around their screen position. A size of 0 makes the markers invisible. There is a wide range of marker styles to choose from, including boxes, circles, triangles and vertical and horizontal bars.

The picture to the right shows a screen dump of all the marker styles in sizes 1 to 10. If you need to tell them apart on screen, sizes below 3 should be avoided. If you have excellent eyesight and a high-resolution printer, size 1 is viable in printed output.

# Options

This command is for XY windows only and opens the XY options dialog. This dialog controls the XY window "key". The key is a small region to identify the data channels that you can drag around within the XY window. For each visible channel it displays the channel name and draws the line and point style for the channel. The dialog also has a check box that controls the automatic expansion of the axes when new data is added. The equivalent script language command is XYKey().

This example (made by the clock.sgs script in the Scripts folder) shows the key. You can choose to make the key background transparent or opaque and choose to draw a border around the key. If you move the mouse pointer over the key, the pointer changes to show that you can drag the key around the picture. Double-click the key to open the Options dialog.

# Font

This command sets the font that is used for the current view and in other windows connected to the current view. The font selection dialog is generated by the operating system, and varies with the version of Windows.

The font size changes the space allocated to data channels in a data view. Smaller fonts give more space to the channels, however fonts need to be large enough to read easily. You can set different fonts in each data or text window.

In a text-based view, this command opens the text editor settings dialog for the current text view type where you can set fonts and text styles for view. This is described in the documentation of the Display page of the Preferences dialog.
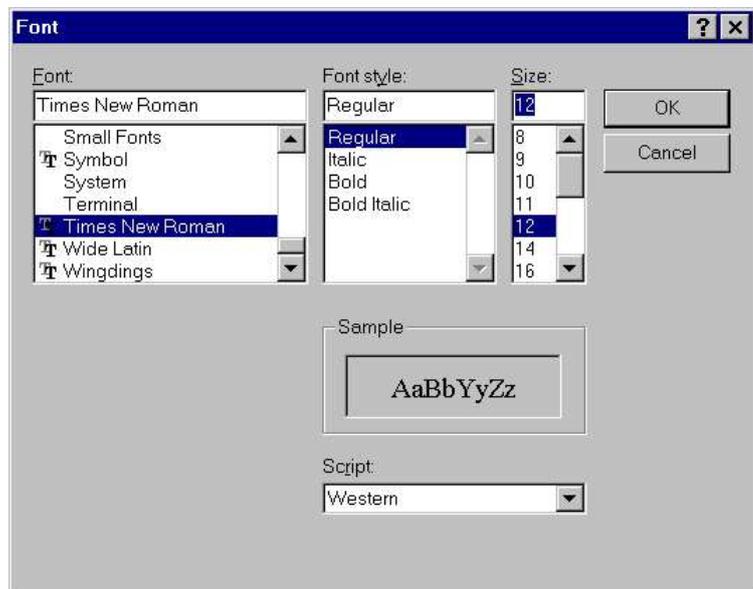
# Use Colour and Use Black And White

If you have a colour monitor, you can choose to display your data files in colour. The Use Colour menu item switches from black and white data displays to colour. If you change to colour, the menu item changes to Use To Black And White. You may prefer to work in monochrome if you have to print the end result in black and white.

# Change Colours

You can choose the colours that are used for almost everything in Signal. The Change Colours dialog controls colours in data and XY views and information windows (use the Font dialog to controls colours in text-based views). To open the Change Colour dialog, use the View menu Change Colours command or click on the palette in the main toolbar. The dialog has multiple sets of colour selections which are selected using the drop-list at the top left, plus a palette of generally useful colours on the right. The colour sets available are: Application colours, Overdraw cycling colours, View colours, Channel primary colour, Channel secondary colour, Channel background colour. The Application and Overdraw cycling colours pages are always available, the others are only available when a data or XY view is active and act to optionally override the application colours for that view only.

*Colour choice overrides*
To avoid your data, axes and other items becoming invisible if you choose a colour for them that is the same (or very similar to - the minimum allowed contrast varies with what is being drawn) the background colour, Signal will

on occasion override your selected colours and replace them with colours that are certain to be visible. This colour choice override can be disabled in the display preferences if necessary.

There are two ways to assign a colour to an item in the currently displayed colour set:

1.  Select an item by clicking on it (or select multiple items by holding down the Control key while clicking) and then click on the palette to the right to assign a palette colour to the list item(s).
2.  Double-click on a list or palette item to open a colour picker that is used to directly choose a colour for that item. You can also right-click on a palette item to edit it without affecting any selected list items.

The colour palette, application colour and overdraw cycling colour tables apply globally to the Signal application and are stored in the Windows registry so that they can vary from user to user. The view, channel and information window colour overrides are specific to a data file and are stored in the .sgrx resources file associated with the data file or in the sampling configuration information for new data files.

### Application colours

The application colour set is a list of data and XY view items to which colours are assigned, these are used throughout Signal unless they are overridden by view, channel-specific or information window-specific colours. You can check the result of your action with the Draw button, while the Reset Page button returns the colour list and the palette to a standard set of colours. A right-click menu is also available with the Reset Page command and also a Reset All command to reset all the pages.

You can set the following (bracketed text means it can be overridden):

| | |
|---|---|
| Channel numbers | The channel number, used to identify and select data channels and drag them. |
| Data view background | Background colour of the entire data view (view background) |
| Waveform as line | Waveform data drawn as lines (primary). |
| Waveform as dots | Waveform data drawn as dots (primary). |
| Waveform as skyline | Waveform data drawn as skyline (primary). |
| Waveform as histogram | Border colour for waveform data drawn as a histogram (primary). |
| Waveform histogram fill | Fill colour for waveform data drawn as a histogram (secondary). |
| Markers as dots | Markers drawn as dots (primary). |
| Markers as lines | Markers drawn as lines (primary). |
| Marker text | Marker code values (secondary). |
| Rate histogram outline | Border colour for markers drawn as a rate histogram (primary). |
| Rate histogram fill | Fill colour for markers drawn as a rate histogram (secondary). |
| Text labels (not used) | |
| Cursors & cursor labels | All data view cursors and cursor labels. |
| Controls (not used) | |
| Data display grid | All axis grids. |
| Axis markings & text labels | All axis lines, ticks and labels. |
| Tagged frames background | The background colour used for tagged frames (view background). |

| | |
|---|---|
| Overdrawn data | Overdrawn frame data with overdraw colour selected and original data in filter dialogs. |
| XY view background | The XY view background colour (view background). |
| Standard deviation/SEM | All data view error bars and envelopes (secondary). |
| Fitted data | Fitted curves (secondary). |
| Convoluted trace | Convoluted idealised trace data that has not yet been fitted (secondary). |
| Closed state | Idealised traces that correspond to the closed state (secondary). |
| XY channel data | XY view data channels (primary). |
| XY channel fill | XY view fill colour (secondary). |
| XY key | XY view key. |
| Fitted convoluted trace | Convoluted idealised trace data that has been fitted (primary). |
| Open state | Idealised traces that correspond to open states (primary). |
| Info window text | The text drawn in information windows. |
| Info window background | The background colour used in information windows. |
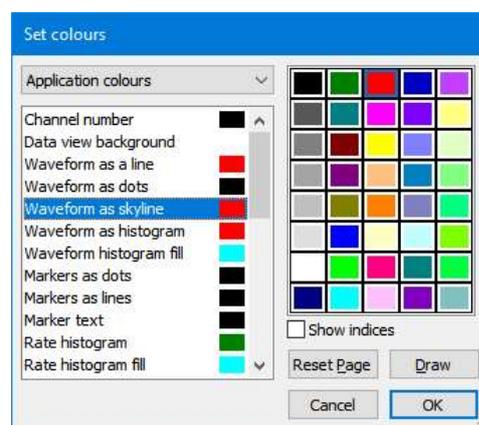
Use the Edit menu Preferences dialog for text view colours. If you set a channel or text colour that is too similar to the background, Signal will change the colour to keep the item visible; there is an option to defeat this in the Edit menu Display Preferences. Script users can set application colours with the `ColourSet()` command.

### Overdraw cycling colours

This colour set is used with overdrawn frames with colour cycling selected in the overdraw settings. In this mode each frame is drawn using a separate colour taken from this colour table, starting again at the beginning when the end of the table is reached. The length of the colour table is variable. These colours are also used when marker or real marker data are drawn using colours set by the marker code values, here marker code zero selects cycling colour 1, code one selects cycling colour 2 and so forth. Again, once the end of the table is reached the colour sequence restarts at the start of the table.

Click the Set... button to change the number of overdraw cycling colours in the range 8 to 100. If you increase the number of colours, the new colours will be set to black. Script users can set the overdraw cycling colours with the `ColourSet()` command.
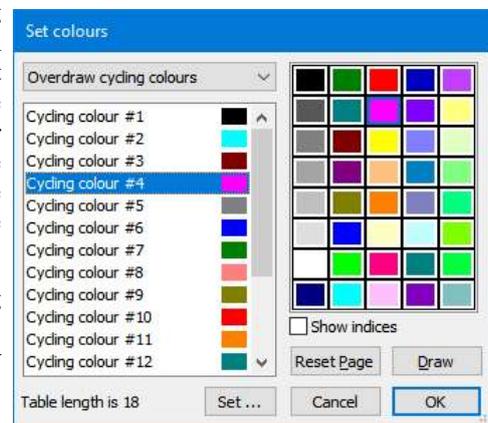
### View colours

This colour set overrides the application colours for the current view and any associated views, you can override the standard view background colour, the information window text colour and the information window background colour. Changes made to the information window colours apply to all information windows associated with this view. The equivalent script command for this is `ViewColourSet()`.

Changes made on this page are applied immediately, so there is no Draw button, the Reset button returns the selected items back to the standard colours as set in the Application colours, Reset Page resets all the colours on the page.

**Channel primary colour**
**Channel secondary colour**
**Channel background colour**

These colour sets assign colours to data channels in the current data or XY view which override any application or view colours. The primary colour is used as the drawing colour for lines, waveforms, and histogram outlines. The secondary colour is for filling histograms and XY channels and for drawing error bars. The channel background colour overrides the view background for the area occupied by the channel data, for an XY view all channels share the same background. An X in a box marks a channel with no colour override.

Changes made on this page are applied immediately, so there is no Draw button. You can Reset the selected channels back to the standard colours set in the Application colours page, Reset Page resets all the colours on the page. Script users can set channel colours with the ChanColourSet() command.
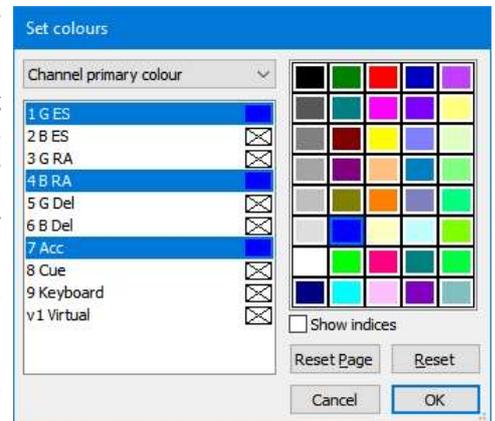
**Information window colours**

The Change colours dialog can also be generated in a special form where it is used to change the colours for a specific information window. When used in this way there is only a single set of colours available (the Information window colours) which has two colours available; the text colour and the background colour. Colour choices set in an individual information window using this dialog will override the information window colours set in both the application colours and in the parent view colours. The equivalent script command for this is ViewColourSet().

**Directly changing a palette or item colour**

To change a palette colour or directly edit an item colour, double click on it to open the colour picker and select a replacement colour.

The first seven palette colours form a grey scale from black to white and cannot be changed. You can replace the palette colour with any standard colour, or you can click the Define Custom Colours... button to select an arbitrary colour. Click OK or Cancel to exit. You can also right-click on a palette colour entry to change the palette colour without also changing any selected colour item on the left.

The colour palette and application colour tables are owned by the Signal application, not by the data file and are stored in the registry. When Signal starts, the palette, application, view, channel and overdraw cycling colours are read from the registry; if there are no colours to read Signal uses defaults.
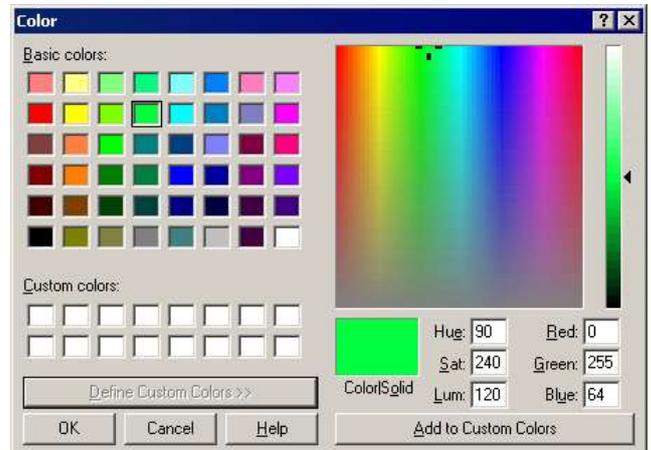
**Show indices**

Check this box to display the colour table index for the application colours (main and colour cycling) and for each colour in the palette - this is a convenience for script programmers.

**Script language**

The script language function ColourSet() controls the palette, application and overdraw cycling colours. The ChanColourSet() function controls the Channel primary, secondary and background colours and The ViewColourSet() command controls the view colours.

# Changes at version 5.02

Prior to version 5.02, the application, view and channel colours were all controlled by the colour palette. All colourable items had an index into the palette and were displayed with whatever colour was at that index. This made sense when Signal was originally written as most systems had 16 colours, some had 4 and some had only 2. It was possible to generate intermediate colours by mixing dots of different hues, but this led to unpleasant screen effects. Apart from limiting the available colour choice to the colours in the palette, using palette indices had the additional effect that a change to a palette colour might change items other than those that were intended.

At version 5.02, all items now have a free choice of colour, defined by RGB (Red, Green and Blue) values in the range 0 to 1. For example, RGB values of 1,0,0 are bright red, values of 0,0,0 generate black, 1, 1, 1 generates white and 0.5, 0.5, 0.5 displays as a mid grey. We have preserved the palette as it is useful to have a set of colours that are quick and easy to apply, but a colour set by clicking in the palette sets the RGB value of the palette colour; the index into the palette is of no consequence.

We have tried very hard to make the colour system fully backwards compatible. The only difference is that if you double click in the palette to change the colour, this only effects the currently selected item in the list on the left of the dialog. Previously, if you changed palette item `n`, this would change the colour of any Application, View or Channel colour that happened to be pointing at palette index `n`.

The changes should make it much easier to set colours using a script. However, we have preserved the old script commands so that old scripts will continue to work. We very much encourage you to avoid the old script commands in new scripts (unless they must also run in older versions of Signal). The new script commands that allow you to set and read back colours using RBG values are: `ColourGet()`, `ColourSet()`, `ChanColourGet()`, `ChanColourSet()`, `ViewColourGet()` and `ViewColourSet()`. You can still use the following commands, but we suggest that your avoid them in new code: `Colour()`, `ChanColour()`, `ViewColour()`, `XYColour()`, `PaletteGet()` and `PaletteSet()`.

We have also separated out the overdraw cycling colours, which can be used used when overdrawing frames. This allows users to customise these colours and also to us to provide more overdraw cycling colours, if needed.

The new system is more open-ended than the old, and allows us to add further colours and colour overrides in the future.

# Folding

This item, which is available for script and sequencer views, allows you to select the style of code folding margin (including disabling code folding), while the Expand All Folds command unfolds all fold points, Collapse All Folds folds all fold points, and Toggle All Folds finds the first fold point in the document, reads its folded or unfolded state and then sets all folds in the document to the opposite state.

# Show Gutter

The gutter is an area to the left of the text in a text view, usually with a grey background, that can be used to select lines of text, except in a script view where clicking on it sets break points. The gutter is also used in script views to display the current line pointer. This menu command shows and hides the gutter, equivalent to the Gutter() script command. The gutter shares the same display background colour as line numbers. The gutter is normally visible.
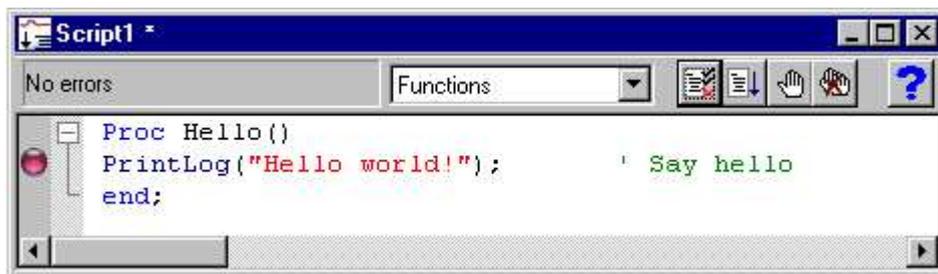


*Script view showing the Gutter, folding margin and selection margin*

If you turn the gutter off, there is a selection margin to the left of the text that can be used to select lines of text by clicking and dragging.

# Show Line numbers

You can choose to display line numbers in any text view. Line numbers usually have space for up to 5 digits, but if you need more than 99,999 lines of text you can use the ViewLineNumbers() script command to increase the displayed digits. This menu command makes the line number area visible if it is invisible and hides it if it is visible. You can change the appearance of the line number region with the View menu Font command. Select the Line number style, and a line number will appear in the example window so you can preview any changes.

If you display line numbers, you will usually want to turn off the gutter, except in a script view, where the gutter is used for break points and to display the current line marker. You can use the line number margin to select complete lines of text.

# Mouse display control

There are a number of ways that you can use the mouse to interactively adjust the display:

The X and Y axes can be shifted and zoomed by dragging on the axis ticks and axis numbers respectively.

You can zoom in or out on the X axis, Y axis or both by dragging rectangles with the cursor. Dragging only in the X direction or across multiple channels zooms in on the X axis to show the range dragged, dragging in the Y direction on a channel zooms the channel Y axis range to show that range and dragging in both zooms in in both directions. If you hold down the Ctrl key while dragging, the display will zoom out instead of in.

If you hold down the Alt key while dragging a rectangle the X and Y size of the area dragged is shown to allow quick measurements to be taken.

You can click on the channel number shown adjacent to a channel Y axis to select or de-select the channel. Holding down the Ctrl key allows you to select multiple channels, holding down Shift allows selection of ranges of channels.

You can drag a channel number to adjust the order of the channels, or drop the channel number on top of another to overdraw channels.

Holding down the Shift key and clicking on a channel data area, then dragging, will adjust the vertical size provided for the channel and its neighbour. Holding down Ctrl as well as Shift allows you to adjust the space allocated to all channels.

Right-clicking on a data or XY view will provide a context menu with generally useful options and options specific to the data channel (& cursor) clicked-upon.

In an XY view the position of the optional key can be adjusted by dragging it using the mouse.

# Keyboard display control

Windows software is usually orientated towards control by means of the mouse and menus, but it is often convenient to use the keyboard instead. For interactive adjustments of the data or display, keyboard control can also be much faster. With this in mind, Signal includes keyboard shortcuts to handle most display manipulation requirements:

| | |
|---|---|
| Scroll data down / up | `Cursor Down` / `Cursor Up` |
| Decrease Y range / increase range | `Ctrl+Down` / `Ctrl+Up` |
| Optimise Y range | `End` |
| Show all Y range | `Home` |
| Y axis dialog | `Ctrl+Y` |
| | |
| Scroll left / right | `Cursor Left` / `Right` |
| Decrease X range / increase range | `Ctrl+Left` / `Ctrl+Right` |
| Show all X range | `Ctrl+Home` |
| X axis range dialog | `Ctrl+X` |
| | |
| Next frame / previous frame | `PgUp` / `PgDn` |
| First frame / last frame | `Ctrl+PgDn` / `Ctrl+PgUp` |
| | |
| Zoom / un-zoom channel | Double-click |
| Hide selected channels | `Del` |

| | |
|---|---|
| Customise display | `Ctrl+Del` |
| Undo changes | `Ctrl+Z` |

Some of these shortcuts are documented with the appropriate menu command, others do not have an equivalent command. All display shortcuts are listed here for convenience. There are more shortcuts provided for data manipulation; see under *Analysis menu*.

# Annotate

This menu item is visible if the current view is a script and is enabled if the script is compiled. Select the option to display the compiled script code below each script line. Select it again to hide the compiled code. We use this annotated view to diagnose script compiler problems.

# Grid view commands

These are the View menu commands that apply to Grid views.

### Optimise Widths

This command scans all the data in the grid and sets each column to be wide enough to display all the data in each cell. The script language equivalent is `GrdColWidth()`.

### Align Column

Each column has an alignment of Left, Centre or Right. The script language equivalent is `GrdAlign()`.

### Show Column/Row Header

You can turn off and on the column and row headers. The script language equivalent is `GrdShow()`.

# Analysis menu

This menu creates memory and XY windows that hold analysed data from data views, fits curves to data, manages the frame buffers and frame manipulation and provides channel data modification mechanisms. It also holds the digital filtering command, and commands for single channel analysis (these are documented separately and will not be visible unless clamping features are enabled).

## New Memory View

This command is enabled when a file view is selected. It opens a pop-up menu in which you can select a memory view generating analysis type. The analysis types available are: Waveform Average, Auto-Average, Amplitude histogram, Power Spectrum and Leak Subtraction (this last only if you have clamping enabled).

Selecting an analysis opens a Settings dialog where you set the analysis parameters and other information needed to construct a memory view to hold the analysis results. This dialog can be recalled from the memory view to change the analysis parameters.

Click New in the Settings dialog box to create a memory view with all data values set to zero and to open the Process dialog, in which you select the source data frames to analyse. The results of analysing different sets of frames can be summed by repeatedly using the Process dialog to select different frames.

## Waveform Average

This analysis averages waveform data channels across multiple frames. The Channels field sets the waveform channels to average. The Width of average field sets the width of the new memory view. The Offset field sets the start time for the data as an offset from the start of the frame. An offset of zero selects data from the start of the frame, regardless of the frame start time. The data from each frame to be analysed starts at Frame start + Offset and runs up to Frame start + Offset + Width. If this data range extends beyond the end of the frame data for any sweep, the sweep is not added into the average and an error message is generated.



If you check Average x axis starts at zero, the X axis of the memory view created by this analysis starts at zero, otherwise the x axis starts at the start time of the first section of data added into the average. This is Frame start + Offset for the first frame analysed.

The Display mean of data check box selects between displaying the mean data value or the sum of all sweeps added into the average. It can be changed after the data has been generated.

If you check the Error bars check box, extra information is saved with the averaged data so that Signal can display the standard deviation and standard error of the mean of the resulting data. The Waveform Draw Mode dialog controls the display of the error information.

The New button (or Change if the memory view already exists) closes the dialog, creates the new memory view and opens the Process dialog, described here.
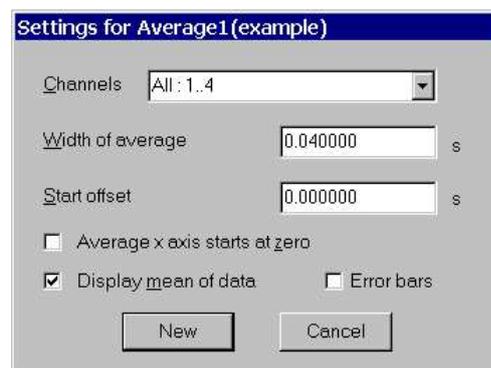
## Auto Average

This form of analysis averages waveform channels in the same manner as standard waveform average processing, but automatically produces a memory view with multiple frames, each frame holding a separate average. Memory view frames can be generated from a preset pattern of groups of source frames, or they can be generated according to the source frame state. When using a pattern of source frames, the groups of frames used can overlap, be adjacent, or can have gaps of unused data between them.

The Settings dialog holds fields for the channels, the width of the average, the data start offset and the number of frames and frame separation per average. The Channels, Width of average and Offset fields are all the same as for standard waveform average processing, as are the Average x axis starts at zero, Display mean of data and Error bars check boxes at the bottom of the dialog.

The Auto-average mode selector chooses how Signal works out which destination frame receives the data for a given source frame, you can choose between Standard or By State. In Standard mode source frames are grouped sequentially and each group is added into the next destination frame. In By State mode the source frame state number determines the destination frame; frames with state number 0 are averaged into destination frame 1, state 1 into destination frame 2 and so on.

The Maximum generated frames item sets an optional limit on the number of frames (holding averaged data) that will be produced. In Standard mode once the limit is reached the averaging wraps round frame 1 again and adds further data to the existing averages, if Maximum generated frames is zero then no limit is imposed and the averaged data file grows larger forever. In By State mode this item becomes Maximum state number and any source frames with state numbers higher than this limit are discarded, again a value of zero is treated as 'no limit'.

The Frames per average item sets the number of source frames that are used to make up each average in Standard mode, the Frames between averages item sets the number of frames in the source between the start of one average and the start of the next, again for Standard mode only.

Thus if you want 4 frames to make up each average you would set Frames per average to 4. If you set Frames between averages to 4, Signal will use frames 1, 2, 3 and 4 for the first average, 5, 6, 7 and 8 for the second average and so on. Setting Frames between averages to 6 would mean that frames 1, 2, 3 and 4 generate the average frame 1 as before, but now frames 7, 8, 9 and 10 are used for the next average and frames 5 and 6 are not used. Then again, if you set Frames between averages to less than Frames per average then the data for each average will overlap, with some frames being included in more than one average.

The Count excluded frames check box (Standard mode only) controls whether the frames used are determined only on frame numbers or if they take into account the subset of frames selected for processing. For example, with 4 frames per average, 4 frames between averages and processing tagged frames only, if Count excluded frames is not checked then the first 4 tagged frames will make up the first average, the next 4 tagged frames the next average and so on. Alternatively if Count excluded frames is checked then those of frames 1 to 4 that are tagged will make up the first average, any tagged frames from frames 5 to 8 make up the second average and so on. In this case some of the averages would be formed from fewer than four frames, any averages formed from no frames are left set to zero.

The New button (or Change if from the Process Settings command) closes the dialog, creates the new memory view and opens the standard Process dialog, described here.
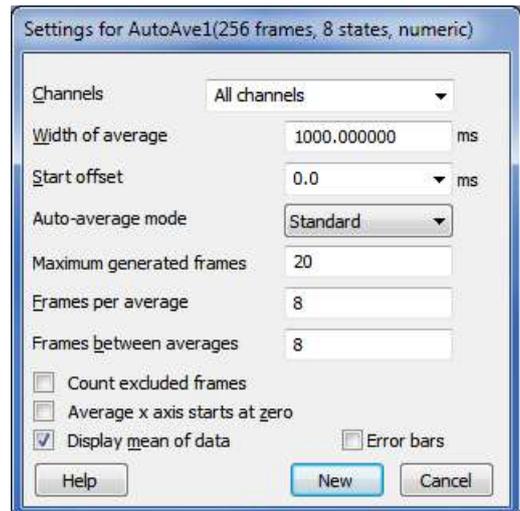
# Amplitude Histogram

This analysis creates a memory view showing a histogram of the number of points of a waveform channel that lie within particular amplitude ranges. This analysis option is not available for log-binned data. The processing reads data for the channel being analysed that covers a set time range, and for each data point read the waveform level is used to calculate the corresponding histogram bin. If this bin number exists within the histogram then the bin data is incremented. This process is repeated for every data point read from every frame, the resulting histogram shows the relative frequency of occurrence of the different waveform levels. For example, if your data waveforms were generally at one of two levels then the amplitude histogram of the data would show two peaks with (probably) Gaussian distributions.
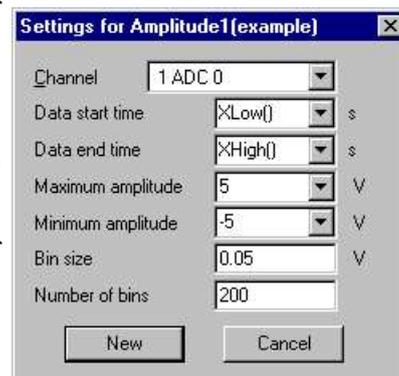
The Channel item sets the waveform channel to be analysed, only a single channel can be selected. The Data start time and Data end time items define the time range over which the source data will be analysed. The Maximum amplitude and Minimum amplitude items set the range of X values that the histogram will cover. These items are linked to the Bin size and Number of bins items, both of which set the number of histogram bins. If either of the amplitude range items are changed the Bin size will be adjusted to fill the new range and keep the Number of bins constant, this also happens automatically if the amplitude depends upon the drawn Y range and this drawn Y range changes. If the Bin size is changed then the Number of bins is adjusted to keep the amplitude range constant. Finally if the Number of bins is changed then the Bin size changes, again keeping the amplitude range constant.

# Power Spectrum

This analysis creates a memory view that holds the power spectrum of a section or sections of waveform data. The option will not appear for log-binned data. If multiple sections are processed the result is an averaged power spectrum. The results of the analysis are scaled to RMS power (where power is amplitude squared), so they can be converted to energy by multiplying by the time over which the transform was done. Signal uses a Fast Fourier Transform (FFT) to convert the waveform data into a power spectrum.

The channels and offset items in the settings dialog behave the same way as they do in the Waveform Average settings dialog.

The FFT is a mathematical device that transforms data between a waveform and an equivalent representation as a set of cosine waves each with an amplitude and relative phase angle. The version of the FFT that we use limits the size of the blocks to be transformed to a power of 2 points in the range 16 to 262144. You set the FFT block size from a drop down list in the dialog, the dialog will automatically update to show the time range that this block size covers. The way the maths works out, the resulting memory view data ends up with half as many bins as the FFT block size. As for waveform averaging, if the block of data starting at the offset specified runs past the end of the frame the sweep is discarded and no analysis is done.

The data in the memory view spans a frequency range from 0 to half the sampling rate of the source waveform channel. The width of each bin is given by the waveform channel sampling rate divided by the FFT block size. Thus the resolution in frequency improves as you increase the block size. However, the resolution in time decreases as you increase the block size as the larger the block, the longer it lasts.

### Windowing of data

The mathematics behind the FFT assumes that the input waveform repeats cyclically. This means that the maths treats the block of data as though it was taken from an input consisting only of that block, repeated over and over again. In most waveforms this is far from the case; if the block was spliced end to end there would be sharp discontinuities between the end of one block and the start of the next. Unless something is done to prevent it, these sharp discontinuities cause additional frequency components in the result.

The standard solution to this problem is to taper the start and end of each data block to zero so they join smoothly. This is known as *windowing* and the mathematical function used to taper the data is the *window function*. The use of a window function causes smearing of the data, and also loss of power in the result.

You can find all sorts of windows discussed in the literature, each with its own advantages and disadvantages; windows shaped to have the smallest side-lobes spread the peak out the most. By reducing the side-lobes you decrease the certainty of where any frequency peak actually is (or the ability to separate two peaks that are close together). Signal implements the following windows:

No Window   Use this if there is one sine wave, or if more than one, they all have similar amplitude. This has the sharpest spectral peaks, but the worst side-lobes.

Hanning     This is a good, general purpose, reasonable compromise window. However, it does throw away a lot of the signal. It is sometimes called a "raised cosine" and is zero at the ends. If you are unsure about which window would be best for your application, try this one first.

Hamming     This preserves more of the original signal than a Hanning window, but at the price of unpleasant side-lobes.

Kaiser      These are a family of windows calculated to have known maximum side-lobe amplitude relative to the peak. Of course, the smaller the side-lobe, the more signal is lost and the wider the peak. We provide a range of windows with side-lobes that are from 30 to 90 dB less than the peak.

# Leak Subtraction

This analysis creates a multi-frame memory view by carrying out a leak subtraction analysis on the source data. Leak subtraction is a specialised analysis used by voltage and patch clamp researchers, this analysis will not be available unless clamping support has been enabled in the Clamp page of the Preferences dialog.

The leak subtraction method used in Signal is based upon the use of a small stimulus, one that does not cause the cell membrane ion channels to turn on, and measuring the current flow through the membrane made up from resistive and capacitive components caused by this stimulus. If possible a number of these low amplitude stimulus frames are recorded and averaged together to minimise the effect of noise. This gives us a current waveform representing the resistive and capacitive response of the cell to the low amplitude stimulus. This leak measurement is then scaled by the stimulus amplitude used in following frames to generate the leakage currents that will be generated by these stimuli and this scaled leak current is subtracted from the recorded traces to leave only the ion-channel effects.

This technique is very satisfactory in many circumstances because it measures the actual response of the cell to stimuli, which in particular allows it to compensate for whatever capacitive behaviour the cell exhibits without making any particular assumptions about what this capacitive behaviour is. However the analysis does make a number of assumptions which you should bear in mind before using it:

1.  The times and duration of the stimulus pulse(s) are the same in all frames, because if the pulse times in the averaged low-stimulus data are not the same as the times in the data that is to be modified then the subtraction of the scaled low amplitude response cannot work correctly. So this mechanism is not suitable for protocols using stimuli that start at varying times or varying duration stimuli.

2.  The stimuli are simple square pulses. The stimulus amplitude is measured by averaging over a specified time range but this measurement will not work correctly if the stimulus is not all at this measured level. So this mechanism is not suitable for protocols using arbitrary waveform stimuli and would have to be used with great care with ramps and sine waves - you would have to make sure that the measured amplitudes gave an accurate estimation of the relative scale of the stimuli.

3.  There are only two stimulus levels in each frame; the baseline or holding potential and the stimulus potential. This is because of the scaling, which measures the stimulus amplitude used and scales the low amplitude response accordingly, obviously cannot scale by two different values at once. This restriction gives the most difficulty as it makes it very hard if not impossible to use leak subtraction with paired pulse protocols. In order to use this mechanism with multiple pulses that are at different amplitudes (if they are all at the same amplitude there is no problem) you have to ensure that you are measuring the amplitude of the pulse giving the response that you are interested in, and that this measured amplitude and the measured baseline level correctly describe the start and end transitions of the stimulus in question.
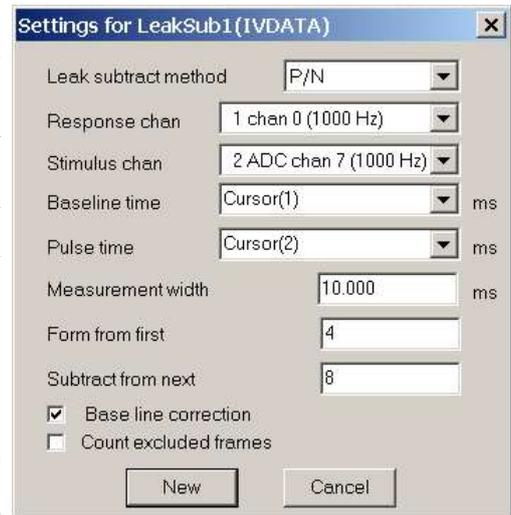
So a paired-pulse protocol with two pulses that are separated by a section of baseline and where you are only interested in the response to the second pulse will be OK as long as you set the pulse time setting to the time of the second pulse (and as long as you realise that the analysis will mess up or ruin the responses to the first pulse and are OK with that). However if there was no gap between the two pulses the scaling of the low amplitude stimulus recording would not work as it would not give a correct estimate of the capacitive transient at the start of the second pulse (because it did not start from the baseline level and therefore is not scaling relative to the pulse amplitude measured from the baseline).

Leak subtraction makes special use of two channels; the stimulus channel which is used to measure the amplitude of the stimulus pulse so that the leak can be scaled (but which is not modified by the analysis) and the response channel which is the only channel modified by the leak subtraction process. All other channels are ignored and copied into the memory view unchanged.

The Settings dialog holds fields for the leak subtraction mode and to define the channels for the stimulus and the response signals. Baseline time defines a time within the sweep where there will be no stimulus and the Pulse time is a time where there is a stimulus. These two times are used to measure the stimulus amplitude from the relevant channel; the measurements are averaged over the Measurement width (using the time range from time-width/2 to time+width/2).

The following two edit fields are used to specify the frames used to generate the leak measurement and the frames from which the leak is subtracted. These fields are interpreted in different ways depending on the leak subtraction method (see below). If Base line correction is on, the corrected response will also have a DC offset removed so that at the baseline time the response level will be unchanged.

The Leak subtraction method can be set to Basic, P/N, or States. These three modes are very similar, the only real difference being how frames used to generate the leak trace are selected:

Basic    here the leak measurement is generated from a fixed contiguous set of frames regardless of which frames are being processed, these frames are set using the First frame for leak and Last frame for leak items. All of the frames processed use this single leak measurement, frames contributing to the leak data are automatically skipped during processing.

P/N    here the leak measurements are extracted from the frames that are being processed. The first n frames processed are used to make the leak data, then the next m frames are processed using this leak data. Then the next n frames are used to make a new leak data set and the following m frames are processed using this new leak data, this cycle continues throughout the frames being processed. The values for n and m are entered in the settings dialog using the Form from first and Subtract from next items respectively.

States    here the leak data is assembled from all frames within the file with a given state, the state number is entered in the settings dialog. All the frames processed use this set of leak data, frames contributing to the leak data are automatically skipped.

As in Auto-Average processing if, for instance, we only process un-tagged frames and there are some tagged frames in the file, the process will continue to search for un-tagged frames until the required number have been found to complete the formation or subtraction of the leak. If Count excluded frames is turned on, however, then even frames that are excluded from the process will still be counted as part of the frames used.

The New button (or Change if this is used from the Process Settings command) closes the dialog, creates the new memory view and opens the Process dialog, described here. Leak subtraction processing uses the same process dialog, but because leak subtraction creates a set of frames in the memory view the behaviour is subtly different; the Clear bins check box, if set, clears out the entire memory view (deleting all frames apart from the first one) and if not set does not accumulate more data into the existing memory view frames but rather appends more frames to the view. For similar reasons the Analysis menu Append Frame command does not create a second set of process parameters; all frames use the same process parameters.

# Interval Histogram

This analysis creates a memory view holding a histogram showing the intervals between markers on one or more channels, either simple markers or real markers can be analysed. The processing reads data for the channels being analysed and for each marker item calculates the interval since the previous item and calculates the corresponding histogram bin. If this bin number exists within the histogram then the bin data is incremented. This process is repeated for every marker from every frame that is processed, the resulting histogram shows the relative frequency of occurrence of different intervals between markers. For example, if your data generally has marker intervals of 25 milliseconds histogram of the data would show a peak at about this point with (probably) a Gaussian distribution.

The analysis normally ignores the first marker in each frame as there is no previous marker, but you can optionally measure intervals that cross frame boundaries.

The Channel item in the settings dialog sets the marker channels to be analysed, each analysed channel generates a separate channel in the memory view that is generated. The Width, Bin size and Minimum interval items define the histogram that will be generated; the number of histogram bins is Width / Bin size, while the range of marker intervals covered runs from Minimum interval to Minimum interval + Width.

The Use intervals that cross frame boundaries check box allows interval generation for the first marker in each frame by remembering the time of the last marker seen in the previously processed frame. The calculation makes use of the absolute frame start time recorded by Signal, this value is completely accurate for data files recorded using Gap-free mode but is rounded down to the previous outputs timing interval for other sampling modes and may therefore lead to slight inaccuracies in the histogram data. More seriously, with non-gap free sweeps markers falling in the gap are not logged, which would give an incorrect large interval. For these reasons the use of this option with non-gap free data is not recommended.
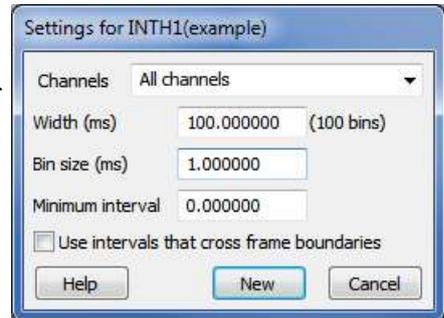
# Process

This command is available when a memory view created using the New Memory View command or an XY created using New XY view is the current view, or when a data view containing a data channel created by measurement processing is current. When you use it a dialog prompts you to select the frames of the source data document to be processed. The Process dialog is also provided automatically when you use the New or Change button from the Process Settings dialog to create or rebuild a memory view.

The dialog contains a number of items used to select the source data frames that will be processed, plus check boxes controlling what happens when data is processed.
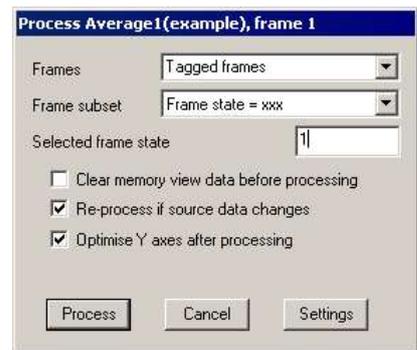
The Frames and Frame subset items are the main controls used to select source frames. The simplest way to use these is to type in a frame list directly. You can also select the current frame, all frames, tagged or untagged frames or frames with a given state code. If you choose the state code option, the dialog also displays a field into which you can enter the state code to use. The Frame subset selector can be used to further qualify the frames which are wanted.

The frame list values are evaluated when the Process button is used, then the frames are processed and the results added into the memory view data.

If you check the Clear memory view before process check box, the memory or XY view or destination channel data is cleared before the results of the processing are added. The Reprocess if source data changes check box enables automatic re-processing. Automatic re-processing is optimised to try to prevent unnecessary work, but can still slow Signal significantly on occasion, particularly with large files. If you check the Optimise Y axis after process check box, the memory or XY view or destination channel y axes will be re-scaled after the new results are added into the destination so as to display the data as well as possible. If you are processing data to generate points in an XY view, an extra check box provides automatic X axis optimisation.

If the Settings button is pressed, the process dialog is removed and replaced by the settings dialog.

### Breaking out of Process

Processing operations can take quite a time, especially in large data documents. You can stop a processing operation early with the `Esc` key.

### Multiple frame processes

For Waveform average, Amplitude histogram and Power spectrum processing, you can use the Append frame command to add more frames to the memory view created by processing. This new frame will have the same

process settings as the original frame, but will have its own separate Process dialog so that you can analyse a different set of source frames for each memory view frame.

# Process command with a new file

The Process dialog is slightly different when used on data from a sampling document. The dialog is also activated automatically when you create a new memory view from a sampling document or when you press the Change button in the Process settings dialog for a similar memory view.

This form of the Process dialog gives you control over when and how the destination memory view is updated during sampling. The Frames field contains extra items that are suitable for processing sampled data: Sampled frames and Last n frames filed. The contents of the dialog change depending upon which frame option is selected. In addition there is a new field: Frames between updates.
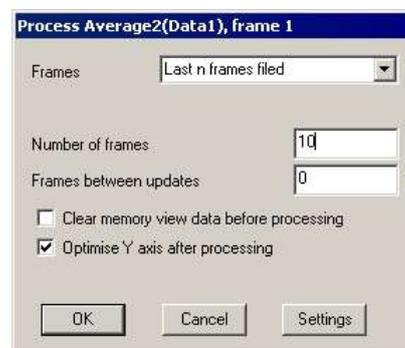
| | |
|---|---|
| Sampled frames | all frames that are sampled will be processed. This option is not available in Fast triggers, Fast fixed interval or Gap free sampling modes because unwritten frames will be discarded before Signal has a chance to process them. |
| All filed frames | all sampled frames that are saved to disk are processed. |
| Last n frames filed | process the most recent frames saved to disk; the dialog displays a field in which you can enter the number of frames required. The Clear memory view check box is ignored as the memory view is always cleared before processing. |

The Frames between updates field sets how often the processing of filed frames occurs. Set this to zero to process as often as possible. If you are processing Sampled frames, then this field is ignored and the memory view is updated for each frame.

*Processing to a data channel*

When online processing is generating measurements to a data channel the dialog has a different item: Leeway for processing, which replaces Frames between updates. This is used to avoid processing right up to the latest data, as you may want to take a measurement that includes data after cursor 0 or position active cursors after the latest feature found by cursor 0 iteration and if the data has not yet been sampled the cursors will not position correctly. Set the leeway value to the amount of data you want to be present after the cursor 0 position.

# Process settings

This menu command re-opens the analysis settings dialog for the current memory or XY view. This is the same dialog as the one used to define and create the new view except that the New button is now a Change button. The Change button accepts the changed settings, if the changes are significant enough to make any previous analysis incompatible the destination memory view will be cleared and re-initialised.

# Measurements

This command, analogous to the New Memory View command, is available when a file or memory view is selected. It provides a pop-up menu from which you can select a measurement-generating analysis type, the types of analysis available are XY View, Data Channel and XY View (Trend plot). The major difference between these types of analysis is that XY View and Data Channel analysis use cursor zero iteration through points of interest to generate one or more measurements per data file frame, while XY View(Trend plot) analysis does not iterate and generates just one measurement per frame. Selecting an analysis opens a Settings dialog where you set the analysis parameters and other information needed to construct the XY view or data channel to hold the analysis results. This dialog can be re-opened from the XY view, right-clck context menu or analysis menu to change the analysis parameters.

Click New in the Settings dialog box to create the empty XY view or data channel and open the Process dialog, in which you select the data file frames to analyse. The results of analysing different sets of frames can be summed by repeatedly using the Process dialog to select different frames.

# XY View

Measurement generation to an XY view generates multiple measurements from each data frame by using a special cursor, cursor 0, which is used to iterate through each frame searching for features of interest. Multiple XY channels (up to 32) can be created, a separate XY data point is generated for each channel for each feature found by the cursor 0 search. The X and Y measurements are defined separately, allowing great flexibility of operation.

When Measurements to XY View analysis is used Signal provides a Settings dialog used to define the various analysis parameters. This dialog has three separate sections controlling Cursor 0 stepping, X measurements and Y measurements plus a few extra controls.

## Cursor 0 stepping

The Cursor 0 stepping section of the dialog is at the top of the dialog and contains items that define how cursor zero searches through each processed frame to find features in the data. When a frame is processed, cursor zero is initially positioned at the Start at time. Then (except for Expression mode, where a measurement is generated at the initial position) the feature is searched-for, other cursors are positioned and a measurement is taken if the search has been successful and the other active cursors have also succeeded in their searches (are valid). This search-and-measure process is repeated until the search fails or the End at time is reached. An XY data point is generated for each feature found; there may be no features found in a given frame of data and in that case no measurements will be generated for that frame. The available cursor 0 stepping methods are:

| | |
|---|---|
| Peak find | local maxima in the data are found. The data must rise and fall again by the set amplitude, a maximum allowed width for the peak can be set. |
| Trough find | local minima in the data are found. The data must fall and then rise again by the set amplitude, a maximum allowed width can be set. |
| Rising threshold | upwards crossings through a threshold level are found. The data must first fall far enough below the threshold level as set by a hysteresis value, a delay value sets how long it must remain above the threshold. |
| Falling threshold | downwards crossings through a threshold level are found. The data must first rise far enough above the threshold level as set by a hysteresis value, a delay value sets how long it must remain below the threshold. |
| Outside dual thresholds | transitions from within a pair of levels to outside the levels are found.. The data must first of all lie sufficiently far within the levels as set by a hysteresis value, a delay value sets how long it must remain outside the levels. |
| Within dual thresholds | transitions from outside a pair of levels to within the levels are found.. The data must first of all lie sufficiently far outside the levels as set by a hysteresis value, a delay value sets how long it must remain within the levels. |
| Slope peak | local maxima in the slope of the data (steeply rising data values) are found. The slope measured must first rise and then fall again by the set amplitude, a width value sets the time range over which the slope is measured. |
| Slope trough | local minima in the slope of the data (steeply falling data values) are found. The slope measured must first fall and then rise again by the set amplitude, a width value sets the time range over which the slope is measured. |
| +ve slope threshold | upwards transitions of the slope through a threshold level are found. The slope value must first be sufficiently below the threshold level as set by a hysteresis value, a width value sets the time range over which the slope is measured. |
| -ve slope threshold | downwards transitions of the slope through a threshold level are found. The slope value must first be sufficiently above the threshold level as set by a hysteresis value, a width value sets the time range over which the slope is measured. |

| Turning point | points at which the slope changes sign in either direction (peaks and troughs in the data) are found. A width value sets the time range over which the slope is measured. |
|---|---|
| Data points | a specified number of data points or markers. |
| Expression | a string specifying a time such as "Cursor(0)+0.1" is evaluated and cursor 0 set to the position generated. This is most useful for stepping cursor 0 through the data by a fixed amount, but other effects are possible. Iteration stops when the cursor passes the End at location, if the expression used does not move cursor zero onwards through the data then the analysis will hang. |

These stepping methods and nearly all of the parameters controlling them are identical to the active cursor modes for cursor zero, see the active cursor documentation under *Cursor menu* for complete details of these. If cursor zero is already set up as an active cursor when a Measurements analysis is created the current active cursor parameters are copied into the settings dialog.

The Skip if parameter can contain a string such as "Cursor(1) < Cursor(2)" that is evaluated after each cursor zero iteration and other cursor placement. If the result of evaluation is non-zero then no measurement will be taken for this cursor zero position. The User check position check box, if set, allows the user to check each cursor positioning and accept or reject them individually, or cancel further measurements.

## XY Channel management

Below the cursor 0 stepping settings is a region holding controls to manage the XY channels generated. At the left is the Plot Channel control which selects the currently viewed channel, you can set the XY view channel title by typing into the Plot Channel selector and create additional channels by using the Add Channel button. The Delete Channel button deletes the current channel; you cannot delete the last remaining channel.

## X and Y measurements

The X measurements and Y measurements sections are the same as each other. Both hold a selector for the type of measurement plus a variety of other parameters used by the measurement process; another for the channel to take measurements from, items for the one or two time values needed, a measurement width and a fit coefficient number. These additional parameters are shown and hidden according to the type of measurement selected. The types of measurement available are:

| Value at point | the value on the channel specified at the time specified, measured using the set width. A non-zero Width will give a measurement averaged over Time–Width/2 to Time+Width/2, a zero width reads the value of the nearest available waveform data point. |
|---|---|
| Value difference | the difference between the value on the channel at the time specified and the value at the reference time, both measurements using Width as for Value at point. |
| Value above baseline | the difference between the channel value at the time specified and the value at the reference time. Only the reference time measurement uses the specified width, the other is always a single-point measurement. |
| Value ratio | the channel value at the time specified divided by the value at the reference time, both using the specified width. |
| Value product | the product of the channel value at the time specified and the value at the reference time, both using the specified width. |
| Time at point | the time specified. This can be a cursor position; if that cursor's mode is set to move to a feature, it measures the feature position in each frame. |
| Time difference | the difference between the time specified and the reference time. Either or both of these can be a cursor position that can vary. |
| Frame number | the frame number. This is often used as the X measurement to give a plot of 'measurement against frame'. |
| Absolute frame time | the absolute start time of the frame. Often used as an alternative to the frame number to give a 'measurement against time'. |
| Frame state value | the frame state value for the frame in question. |
| Fit coefficients | the value of a coefficient generated by fitting on the specified channel, you should select the coefficient you want from the list provided. If the channel selected has no fit the list shows coefficients by number (a0, a1 and so forth). |
| User entered value | a value entered by the user. A dialog opens to read the value. |
| Iteration count | the count of cursor 0 positions that were found. When used with averaged measurements (which only produce a single measurement per frame), this generates a count of how many features the cursor 0 stepping found in the frame. |

| Expression | a string entered by the user that is evaluated and the result used. Note that, unlike the expression used for cursor 0 iteration, this expression can include all of the dialog expression strings that generate a channel Y value. |
|---|---|
| Cursor regions | any measurement available in the cursor regions window can be used. Not all types of measurement will operate on all channel types; all of them are available on waveform data. |
| Frame variables | a range of frame variables can be selected, some variables are hidden because they are reserved for system purposes. Many of the available variables may not hold useful information but they include the user frame variables accessible from the script language, any membrane measurements results generated by the clamping system and (if in use) Magstim settings. |

The channel selector and time entry fields are as standard for Signal and should be familiar to you and easy to use. Don't forget that, in addition to entering a time value directly or selecting an item such as "Cursor(2)" or "XLow()", you can apply an offset to these selected value allowing you to enter "Cursor(2)-0.1" or "XLow() +1".

**Other options**

The Average measurements within frame check box at the bottom of the dialog applies to all measurement channels. If set then only a single XY data point is generated per frame, this being the average of the data values measured. So, for example, you could use this to produce a plot of mean response amplitude against frame number.

The All channels use same X check box below disables the X measurement for all XY view channels apart from the first one, and uses this X measurement for all XY channels. If appropriate this makes the setup quicker and easier and when the XY data is converted to text only a single column of X values is created which makes it easier to handle in spreadsheets.

The Points item allows you to specify the maximum number of points the currently selected XY channel can contain before old points are deleted to make way for new data. Set this field to zero if you want all data points to be retained.

The New button (or Change if this is used from the Process Settings command) closes the dialog, creates the new XY view and opens the Process dialog, described here. Processing for measurements is very similar to standard memory view processing; the frames specified are used to generate measurements which are added to the view data. If the Clear XY view data before processing check box is checked, all of the data points in the XY view will be deleted first. In addition to the Y axis optimisation control, there is also an extra check box for view X axis optimisation after processing.

# Data Channel

Measurement generation to a data channel generates a single memory channel, either a marker channel or a real marker channel can be created. Multiple measurements are taken from each frame using cursor 0 iteration in the same manner as measurement generation to an XY view. A separate data point is generated for each feature found by the cursor 0 search. The analysis is extremely similar to measurement generation to an XY view, only the differing aspects of this analysis will be described here.

When **Measurements to Data Channel** analysis is used Signal provides a **Settings** dialog used to define the various analysis parameters. This is very similar to the Measurements to XY View settings.

### Channel settings

The most important difference in the settings dialog is the addition of **Channel type** and **Channel name** items at the top of the dialog. The channel type item sets the type of channel to be generated, you can choose between marker or real marker data. This measurement process only creates real marker data that holds a single real data value. The channel name item allows you to set the title of the generated channel, the channel units is automatically generated if necessary. The **Overdraw Y source channel** check box below the channel name is enabled for real marker data when using a suitable Y measurement. If set it causes the newly created real marker channel to be drawn on top of the channel used to generate the Y value measurement.
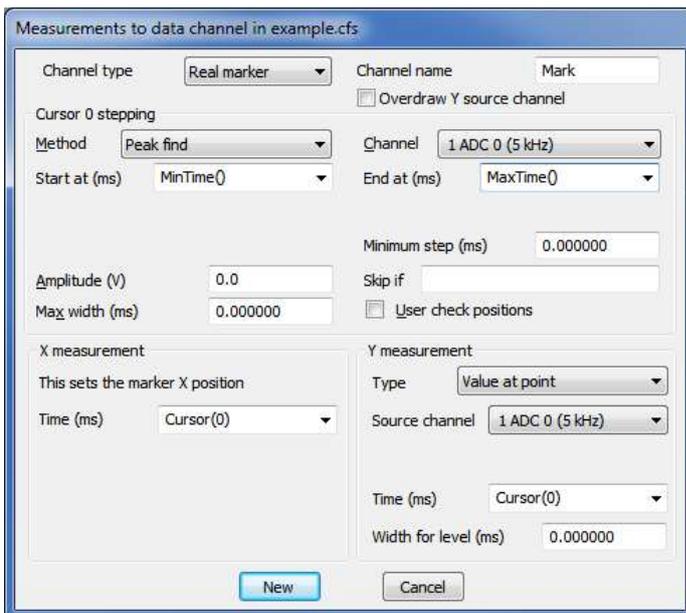
### Cursor 0 stepping

The **Cursor 0 stepping** section of the dialog is identical to the same section in the **Measurement to XY View** settings dialog.

### X and Y measurements

The X measurement part of the settings is much simplified as it generates the time for each marker and therefore contains only a single field, which defines the time at which each marker is generated. For most purposes the default value of `"Cursor(0)"` will be appropriate but any other time calculation such as `"Cursor(1)-25ms"` could be used.

The Y measurement part of the dialog is only available when real marker data is selected as the new channel data type and specifies the measurement used to generate the real value that is held in the real marker. It is almost identical to the **Measurement to XY View** Y measurement options but the **Iteration count** measurement type is not provided.

The **New** button (or **Change** if this is used from the **Process Settings** command) closes the dialog, creates the new memory channel and opens the **Process** dialog, described here. Processing for measurements is very similar to standard memory view processing; the frames specified are used to generate measurements which are added to the channel data. The options to clear channel data and to optimise after processing are not provided.

# XY View(Trend plot)

Measurement generation to an XY view for trend plot analysis generates only a single measurements from each data frame. Separate X and Y measurements are defined, the measurements available are the same as for the other types of measurement analysis. Multiple XY channels (up to 32) can be created, a separate XY data point is generated for each channel for each frame analysed.

Selecting Trend plot analysis provides a Settings dialog where you define the analysis parameters, this dialog is also available later on to change the analysis parameters. The dialog is similar to the lower part of the Measurements to XY View settings dialog; as only one measurement is generated per frame no cursor 0 stepping settings are needed.

**XY Channel management**

At the top of the dialog are controls to manage the XY channels generated. At the left is the Plot Channel control which selects the currently viewed channel, you can set the XY view channel title by typing into the Plot Channel selector and create additional channels by using the Add Channel button. The Delete Channel button deletes the current channel; you cannot delete the last remaining channel.

**X and Y measurements**
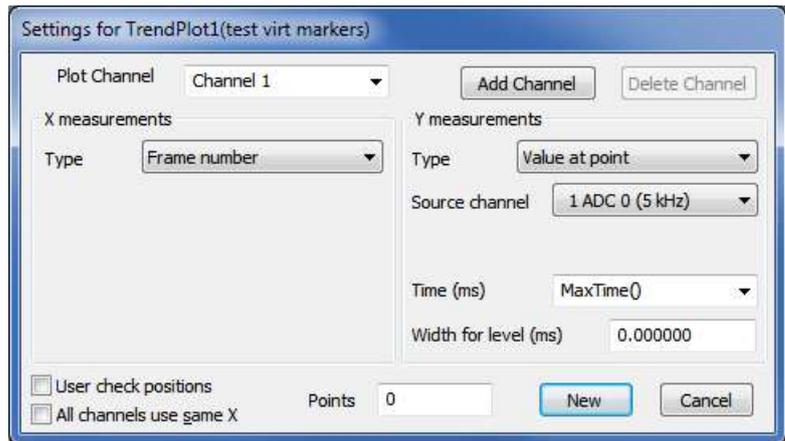
The X measurements and Y measurements sections are the same as the Measurement to XY View measurement options except that the Iteration count measurement type is not provided. Both hold a selector for the type of measurement plus a variety of other parameters used by the measurement process; a channel to take measurements from, items for the one or two time values needed, a measurement width and a fit coefficient number. These additional parameters are shown and hidden according to the type of measurement selected.

**Other options**

The User check positions check box below the X measurements will, when checked, cause Signal to pause on each frame after positioning any active cursors. This allows you to check that the cursor positions are correct and to skip individual measurements or cancel all further measurements.

The All channels use same X check box below disables the X measurement for all XY view channels apart from the first one, and uses this X measurement for all XY channels. If appropriate this makes the setup quicker and easier and when the XY data is converted to text only a single column of X values is created which makes it easier to handle in spreadsheets.

The Points item below the X and Y measurements allows you to specify the number of points the currently selected XY channel can contain before old points are deleted to make way for new data. Set this field to zero if you want all data points to be retained.

The New button (or Change if this is used from the Process Settings command) closes the dialog, creates the new XY view and opens the Process dialog, described here. Processing for trend plots is very similar to standard memory view processing; the frames specified are used to generate measurements which are added to the view data. If the Clear XY view data before processing check box is checked, all of the data points in the XY view will be deleted first. In addition to the Y axis optimisation control, there is also an extra check box for view X axis optimisation after processing.

# Measurements and active cursors

All forms of measurement analysis can be made a great deal more powerful by the use of active cursors. Active cursors can be set up to move to features in the data such as a maximum or a threshold crossing, so that they are automatically aligned to significant features in each data frame. By using the cursor position to specify a time or time range for a measurement, many useful effects can be achieved. For example, with cursor 1 set to find the maximum value between a preset time range within the frame, you could have the X measurement being the frame number and the Y measurement being Time at point. With the time field set to "Cursor(1)", you would get a graph of time for the maximum versus the frame number. See under *Cursor menu* for complete details of active cursors. Active horizontal cursors are also supported by measurement processing.

# Measurement processing online

The Process dialog for measurement processing is slightly different when used on a sampled data view or on a memory view which is itself being created by online processing from a sampled data view. There are a number of different situations, except where specified the various types of measurement processing will behave in the same manner:

*Processing a sampled data file*
The Process dialog is basically the same as when processing to a memory view online, the Frames between updates field controls how often the measurement data is updated. Each frame sampled and processed generates a new trend plot point or set of measurements. For measurements to a data channel the process dialog shows the extra leeway item.

*Processing a memory view generated by online Auto Average analysis*
The Process dialog is as above, the measurements are taken when a new averaged frame is completed - when the required number of source frames have been averaged into the frame.

*Processing a memory view created by online Leak Subtraction analysis*
The Process dialog is as above, the measurements are taken whenever a new frame is added into the leak subtracted data. This is essentially the same as when processing a sampled data file.

*Processing a memory view created by other online analyses*
The Process dialog is as above, all of the available memory view frames are re-processed in a manner very similar to offline analysis whenever any of the memory view data is changed so that the XY view or memory channel data automatically updates to reflect the changing memory view contents.

# Fit data

This command opens a tabbed dialog from which you can fit mathematical functions to channels in a file, memory or XY view. If you fit data to a channel in a file or memory view and error bars are displayed, the fit minimises the chi-squared value, otherwise the fit minimises the sum of squares of the errors between the data and the fitted curve.

In addition to best-fit coefficients and an estimate of how much confidence to place in them, you also get an estimate of how likely it is that the model you have fitted to your data can explain the size of the chi-squared value or sum of errors squared. If your data does not have error bars, these estimates are based on the assumption that all data points have the same, normally distributed error statistics.

The dialog has three pages accessed using the tabs at the top of the dialog:

Fit settings    Set the fit type and range of data to fit and range to display
Coefficients    Set the starting point for your fit and optionally fix coefficients
Results         Display the fitting results and residual errors

The three buttons at the bottom of the dialog are common to all pages. The Help and Close buttons do what they say. Do Fit attempts to fit with the current fit settings.

# Fit settings

This page of the Fit Data dialog controls the type of fit, the data to fit and what to display. The area at the bottom of the window gives a synopsis of the current fit state. Fields are:

### Channel

You can select a single channel from the current view. If this is a file or memory view, the channel must have a y axis. If you change the display mode of a marker-based channel, any fit associated with the channel will most likely become invalid.

### Fit

The fit to use is defined by its name and the order of the fit (a number). For example, an exponential fit allows single exponents or double exponents. The window at the top of the dialog displays the mathematical formula for the fitting function. The following fits are currently supported (N is the maximum order allowed):

| Name | N | Comments |
|------|---|----------|
| Exponential | 2 | This fit includes an offset. You can force a zero offset in the coefficients page. Set a local reference point for the fit, otherwise the even-numbered coefficients may become too large to be useful. |
| Polynomial | 5 | These fits do not require starting values for the coefficients. |
| Gaussian | 2 | If you attempt to fit two overlapping peaks you may need to manually adjust the guesses for the peak centres to get convergence. |
| Sine | 1 | You can fit a single sinusoid with an offset. If the frequency guess (in radians) is not reasonably close, the fit may not converge. |
| Sigmoid | 1 | A single sigmoid may be fitted. |

### Range

You fit data over a defined x axis range, set by the between and and fields. You can choose values from the drop down list or type in simple expressions, for example `Cursor(1)+1`. There must be at least as many data points to fit as there are coefficients. For example, to fit a double exponential, which has 5 coefficients, you need at least 5 points. Most fits will use many more points than coefficients.

### Reference

This is the x axis position to use as the zero value of *x* in the fitting function. The most common value for this would be the start point of the fit. However, in some cases you may want this to be elsewhere. For example, in exponential fitting, you may want to calculate the likely amplitude of a trace at some position. Making this position the reference point makes it easy to calculate the amplitude (it is the sum of the even-numbered coefficients).

### Maximum iterations

All fits except the polynomial are done by an iterative process. Each iteration attempts to improve the coefficient values. The iterating stops when improvements in the fit become insignificant, the iteration count is exceeded, the mathematics of the fitting process suggests that the fit is not going to improve or there is a mathematical problem. This field sets the maximum number of iterations to try before giving up.

### for frames

It is possible to have the fit run automatically across several frames at once. Choose the frames whose data you wish to fit here. If you select Frame state = xxx, an extra field is provided for entry of the state code value.
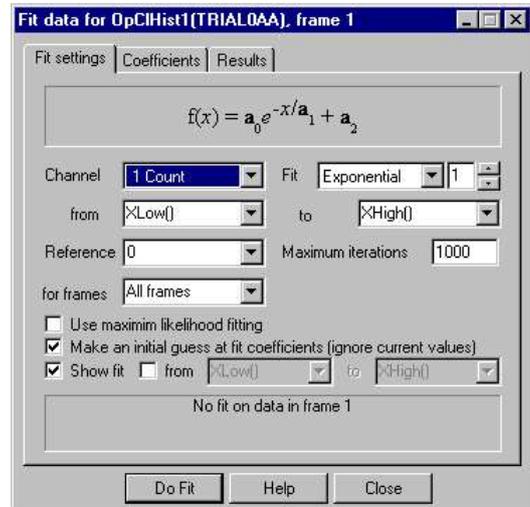
### Use maximum likelihood fitting

This option is only available when fitting exponential curves to data derived from a single-channel idealised trace by generating open/closed time or burst duration histograms and when the processing link between the original trace data and the binned histogram data being fitted is still present. When these conditions apply Signal can use maximum likelihood fitting techniques (which require access to the original, unbinned, trace data). The details of maximum likelihood fitting are a topic beyond the remit of this manual; suffice it to say that by using the original idealised trace data you avoid mathematical complexities and errors caused by the histogram binning process.

### Make an initial guess

The iterative fits need a starting point. There are built-in guessing functions that usually generate a starting point near enough to the solution that the fitting process can converge. If you check this box, these guessing functions are used each time you click the Do Fit button. Otherwise, each fit starts with the current values.

### Show fit

Check this box to display the current fit for the current channel. If the *From* box is checked, you can also choose the range over which to display the fitted data. If this box is not checked, the fit is displayed over the range that the data was fitted.

## Coefficients

This page of the Fit Data dialog lets you set the starting values for iterative fits. You can also use this page to hold some of the coefficients to fixed values and you can set the allowed range of values for fitting.

If you know the value of one or more of the coefficients, type the value in and check the Hold box next to it. For example, in an exponential fit you may know that the final coefficient (the offset) is zero.

The limit values are applied after each iteration. The fit may have to follow a convoluted path before it converges on a solution, so do not set the fit limits too close to an expected solution as this may prevent convergence.

The Estimate values button can be used to guess initial values for fitting based on the raw data. The Clear fit button removes the fit from the channel.

# Results

The results page of the Fit Data dialog holds information about the last successful fit done with the dialog. The page has three regions: coefficient values at the top, a message area at the bottom, and a plot of the residuals (differences between the fit and the data) in the middle. The residuals are displayed immediately after a fit but will not be displayed if you close the dialog and reopen it.
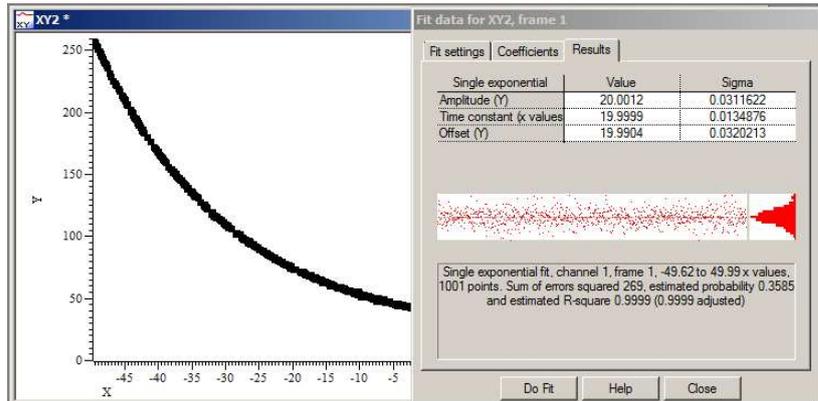


### Coefficient values

The Value column holds the fitted value that minimises the chi-squared or sum of squares error for the fit. The Sigma column is an estimate of how the errors between the fitted curve and the original data translates into uncertainty in the fit coefficients given that the model fits the data and that the errors in the original data are normally distributed. If a coefficient is held, the Sigma value will be 0. For Sigmoid and Sine fits, extra information derived from the fitted coefficients are also displayed. The *Testing the fit* section gives more information on the derivation of these values and how to interpret them. You can select rows, columns or individual cells in this area, the use `Ctrl+C` to copy them to the clipboard. This also copies a bitmap image of the page to the clipboard.

### Residuals

This section of the page displays the differences between the data points and the fitted curve in the large rectangle and a histogram of the error distribution on the right. The error plot is self-scaling based on the distribution of errors; the plot extends from +3 at the top to –3 at the bottom times the RMS (root mean square) error. The grey line across the middle of the plot indicates an error of zero.

In the case that the data can be modelled by the fitting function plus normally-distributed noise, you would expect to see residuals distributed randomly around the 0 error line and the histogram on the right should resemble a normal curve.



If the data cannot be modelled in this way, you would expect to see evidence of this in the residuals. In this example (generated by fitting a cubic to data that was actually a double exponential), you can see that there are clear trends in the errors.



In extreme cases, the error due to the wrong model being used becomes much larger than the errors due to uncertainty in the data values, and you get a residual plot like this one.



### Message area

This area displays a summary of the fit information that you can select with the mouse and copy to the clipboard. The first line holds the type of the fit, the channel number, the ordinate range and the number of points in this range. For example: "Double exponential fit, channel 1, 0 to 50, 50 points".

The contents of the second line depend on the source of the data. If you are fitting a result view channel that has error information displayed, the second line displays the chi-squared error value for the fit, the probability that you would get a chi-squared value of at least that size if the function fits the data and the errors are normally distributed and the . For example: "Chi-square value 58.6 with probability 0.5867 and R-square 0.998 (0.96 adjusted)".

In all other cases, the second line displays the sum of the squares of the errors between the data and the fitted function and an estimate of the probability that you would get a sum of squares of errors of at least this size based on the assumptions that the errors in the original data had a normal distribution that was the same for all points. For example: "Sum of errors squared 1.22, estimated probability 0.8553 and estimated R-square 0.998 (0.96 adjusted)".

If the probability value is very low or very high, there are extra lines of information warning that the fitted function plus normally-distributed noise is unlikely to model the data, or that the errors in the original data have probably been over-estimated. The R-square value is a measure (or estimate) of the proportion of the variation in the data is explained by the fitted equation, the adjusted R-square value has been adjusted for the degrees of freedom available in the data and is normally a more useful form of this value.

The R-square value is the standard "goodness of fit" value, which is the proportion of the variance of the data that can be explained by the fit, so a larger value is better. If you want to compare fitting different orders of polynomials or exponents you should use the adjusted values as these discount the improvement you would expect just because you increase the number of coefficient.

### Context menu

If you right click on this page you are offered a context menu that contains Copy, Log and Log Titles commands. The Copy command copies selected sections of the results, or all the results if there is no selection to the clipboard as text. It also copies the page as a bitmap. The Log command prints a one-line synopsis of the current fit to the log window. The Log Titles command copies a suitable set of titles for the logged data.

# Testing the fit

When you fit a model to measured data to obtain the best-fit coefficients, there are two questions you would like answered:

1.  How well does this model fit the data? Put another way, how likely is it that this model plus some degree of random variation can explain my data set?
2.  Given that the model does fit the data, how much confidence can I place in each of the fitted coefficient values?

When we talk about fitting curves to data, we are making the implicit assumption that you took measurements from some process that follows a model, and that this model can be expressed as a mathematical function with adjustable parameters, which are our fitting coefficients. Further, we assume that the measurements you make are not perfect; they have random variations with a known probability distribution about the correct value. To allow us to calculate likelihoods, we assume that this probability distribution is a normal (Gaussian) distribution. In the real world, or course, only some of this may apply. You may have no a priori knowledge of the distribution of errors in your original data, and this distribution may be anything but normal.

### (Estimated) probability

This value is an attempt to say how likely the model you have chosen plus normally distributed noise is to explain the measured data. If the residuals of your data show other than a random distribution around the zero line, the probability will be 0 (or close to it).

### R-square (R2 or R2 or r2) value

This is a simple measure of the "goodness of fit" and describes what proportion of the variance of the original data is explained by the model. One would expect values in the range 0 (meaning that the model accounts for none of the variation) to 1 (the model accounts for everything). With non-linear fits, it is possible to get a negative result, meaning that a horizontal line through the mean of the data is a better fit than the model.

We also give an adjusted R-square value. This is useful when you have a model that allows you to choose the order of the fit and you are not sure what order is appropriate. For example, when fitting polynomials you have a choice of order 1 through 5. When you increase the order, you make it easier for the model to fit random changes in the data, so the R-square value will improve as the order increases. The adjusted R-square value is less than the R-square value by an amount that quantifies how much we would expect the R-square value to increase by chance due to having more fitting coefficients.

You can read more about R-square here.

### Chi-square fits

In the ideal case, where you know the standard deviations of each data point, the fitting minimises the chi-squared value, which is the sum of the squares of the differences between the model and the data points divided by the standard deviation of data point values. Given a chi-squared value and the number of points it was measured from, we can calculate that probability of getting a chi-squared value at least this large, due to random variations in the

data. This is the value given in the Results section. Ideally, you would like to see a value around 0.5, meaning that you were equally likely to get a larger value as a smaller one. Values very close to 1 mean that, given the errors in each point, the data is too close to the model. Either the error estimates are too large, or the data has been "improved". Although you can hope for probabilities in the range 0.1 to 0.9, values down to 0.01 may occur for acceptable fits, and even smaller values can occur if your error distribution is not as normal as you thought.

Very low fit probabilities will occur if your data contains variations that are significant compared to the errors in the input values and that are not included in the model. For example, if you are fitting exponents to a sampled waveform that includes perceptible mains interference, you can get a good fit (by eye) to the exponential data, but with a probability of 0.0000 as far as the mathematics is concerned because the model does not include the mains hum and cannot explain why the chi-squared value is so high.
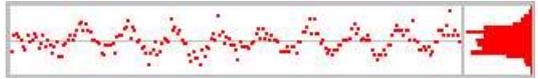
If we assume that the model fits the data, we can make an estimate of the standard deviation of the fitted coefficients. This means, that if we re-ran the experiment many times and fitted the data to each set of results, what would be the likely variation in the fitted coefficients. This is presented as the Sigma value in the results page.

### Least-square fits

If there is no error information for each point, we assume that all the points have the same, normal error distribution and the fit minimises the sum of squares of errors between the model and the data. Because there is no independent estimate of the likely spread of the errors in the original data, strictly speaking, there is no way to give a probability of getting an error of at least this size.

However, we can say (though statisticians may shudder), *"Given that the model does fit the data, and that the errors all have the same, normal distribution, then the differences between consecutive errors should also be normally distributed with twice the variance of the errors"*. We use this to estimate the standard deviation of the data and then we apply the probability test. We label this as *estimated probability*. The same comments about likely values apply as for the Chi-square fits, except that very small values may just mean that our estimation process fails for your data.

The coefficient Sigma values are calculated on the assumption that the model fits the data, that all the original points have the same standard deviation, and that the standard deviation of the original data can be deduced from the residual sum of squares errors.

# Memory Channels

Memory channels hold data that is not part of the CFS data file. While this does not seem very useful at first sight it allows these channels to be more flexible; it is easy to add or delete memory channels and to add or remove data items from them. Memory channel data is held in the resources file associated with the CFS data file, any changes made to memory channels are automatically saved in the resources file when the file is closed.

Memory channels can hold marker or real marker data, waveform data is not supported because of the excessive amount of space this would use up in the resources file. While marker data can also be held in a CFS file, currently real marker data can only be held in memory channels. Idealised trace data is also held in memory channels and can be manipulated by these methods but this is a very specialised form of data and is generally best handled by special analysis menu commands. Memory channels use a separate range of channel numbers but generally you do not need to use these as you can use 'm' followed by a number to indicate them thus: "m1" or "m1..m3".

There are Analysis menu commands to create a new memory channel, to add items to a memory channel, to delete items from a memory channel and to create memory channel data by importing from other channels.

Memory channels can also be created by processing or by using the Signal script language.

# Create New Channel

The create new memory channel dialog is used to create a new memory channel in the current data file. The new channel will be created using the next free memory channel number, if required the memory channel import channel dialog can be automatically provided.

The fields in the dialog change depending on the type of memory channel you choose to create. The example shown is for a real marker channel. The fields are:

*Channel type*
You can create a channel of type: Marker, Real marker or Idealised trace.

*Channel title*
This sets the title of the new channel.

*Units*
This sets the units of the new channel, it is not shown if you have selected the marker channel type.

*Number of reals*
This field is present for real marker data only where it sets the number of real values to store with each marker.

Press the Create button to create the new memory channel, press the Import button to create the new channel and move directly to the memory channel import channel dialog.

# Add Items

This command opens a dialog used to add data items to a memory channel. As many items as are desired can be added to marker-type channels but only one idealised trace event can be added as otherwise the idealised trace channel could easily become corrupted and unusable. The dialog is modeless, allowing you to (for example) alter cursor positions while the dialog is in use. The items shown in the dialog vary according to the channel type.

*Channel*
This chooses the memory channel to be modified.

*Data time*
This sets the time at which the new data will be created

*Marker codes*
This sets the four code bytes for new marker data

*Real index*
This selects between the various real marker real values, it is disabled if there is only one real value per marker. The first real value is at index 0.

*Data*
This sets the value of the selected real marker channel real value.

### Idealised trace data

The dialog shows very different information for an idealised trace channel, the fields here are the event baseline value, the event amplitude and the event period.
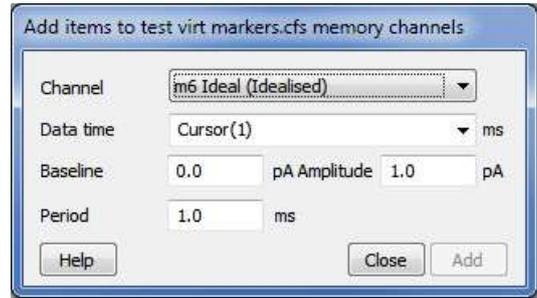
# Delete Items

This command opens a dialog in which you can delete one or more data items or a time range from a memory buffer. The dialog is modeless, allowing you to (for example) alter cursor positions while the dialog is in use. The Delete button removes one or more items as set in the dialog. The Empty buffer button deletes all data for this channel (it does not delete the channel). Close removes the dialog. The MemDeleteTime() script command has the same functions.
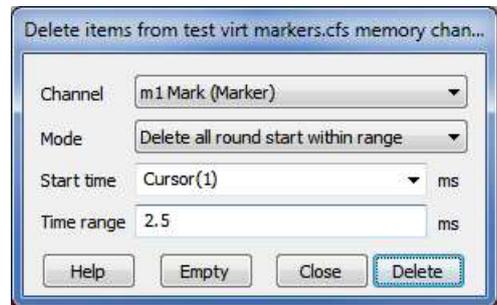
*Channel*
The memory channel to use.

*Mode*
There are four modes: Delete nearest to Start within Range, Delete all round Start within range, Delete first in range Start to End, Delete all from Start to End. The first two modes delete one or more data items around a time, the other two modes delete the first or all data items in a time range.

*Start time*
The time range used is Start time – Time range to Start time + Time range in the first two modes, and Start time to End time for the last two modes.

*End time*
The end of the time range used. This field appears in the second two modes.

*Time range*
The time range around the Start time to search for data to delete. This field appears in the first two modes. This field is usually set to a small value so that you can delete events close to the position of a cursor.

### Idealised trace data

The dialog behaves differently for an idealised trace channel, the modes here are Delete the first event before time, Delete the nearest event to time and Delete the first event after time. Only a single time value is shown.

# Import Channel

You can import data into a memory buffer selected by the Memory field from a channel set by the Source field (you may not import data from a memory buffer to the same memory buffer). The Start time and End time fields set the time region to import data from.

The Frames and Frame subset items are the main controls used to select frames for the import operation. The simplest way to use these is to type in a frame list directly. You can also select the current frame, all frames, tagged or untagged frames or frames with a given state code. If you choose the state code option, the dialog also displays a field into which you can enter the state code to use. The Frame subset selector can be used to further qualify the frames which are wanted.

During data import, values that cannot be extracted from the source are set to 0. For example, when importing a marker channel into a real marker channel, there are real values in the source so these are set to 0.

When importing to a marker or real marker channel from a marker or real marker channel, or when importing to an idealised trace channel from an idealided trace channel, a simple copy operation is carried out. When importing to a marker or real marker channel from a waveform channel a more complex feature detection process is carried out (see below).

The Apply button imports data and leaves the dialog open. The OK button imports data and closes the dialog. The Close button closes the dialog without importing. The Help button or using the F1 key displays this page of help.
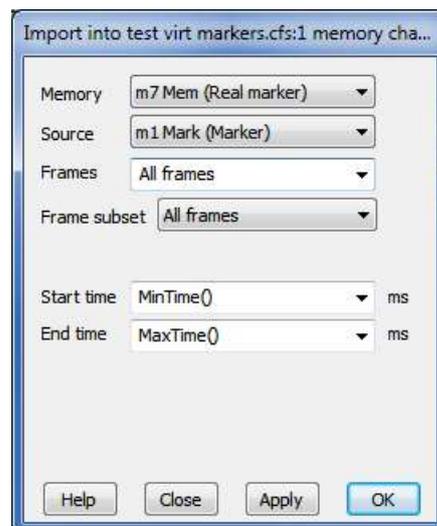
### Extract markers from Waveform channels
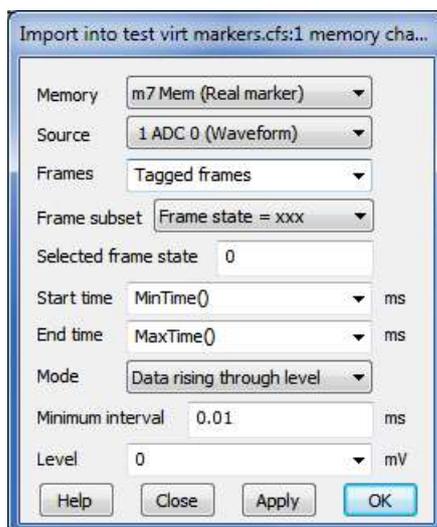
You can also extract marker data from waveform channels. The Mode field is present if the source is a waveform channel and sets the method used to extract marker times from the waveform, there are four modes available: Peaks, Troughs, Data rising through level and Data falling through level. Detected times are added to the memory channel as markers with codes 2 (Peaks), 3 (Troughs), 4 (rising data) or 5 (falling data).

These modes use the Minimum interval field as the minimum separation of detected times, to filter out times caused by noise in the input waveform. Set this to 0.0 if you do not want to set a minimum period between detected times. The peak search mode looks for a peak followed by a fall of at least the Size field. The trough search mode looks for a minimum, followed by a rise of at least the Size field. The rising and falling level modes detect events when the signal crosses the Level in the selected direction.

When importing to a real marker channel each first real marker value is set to the source data value at the marker time.

# Virtual Channels

Virtual channels hold waveform data derived using a mathematical expression from existing waveform channels (including other virtual channels as long as they have a lower channel number) and built-in function generators. No data is stored on disk; the virtual channel data is recalculated as required. You can match the sample interval and data alignment to an existing channel, or type in your own settings. Channel sample intervals and alignments are matched by cubic splining of the source waveforms. The script language equivalent of this command is `VirtualChan()`.

You can use virtual channels to carry out a wide variety of channel arithmetic (for example sums and differences of channels), to linearise non-linear transducers, to process channel data in various ways (for example DC offset removal or smoothing), to generate waveform data from scratch (for example, to copy into a sampling configuration as a waveform stimulus) and to convert marker channels into waveforms proportional to the marker

rate. Virtual channels use a separate range of channel numbers but generally you do not need to use these as you can use 'v' followed by a number thus: "v1" or "v1..v3".

There are Analysis menu commands to create a new virtual channel and to edit the settings of existing virtual channels. Both commands open the Virtual channel dialog:

### Virtual channel

Use this field to select the virtual channel to work with when you have more than one virtual channel. The New Virtual Channel button creates a new channel.
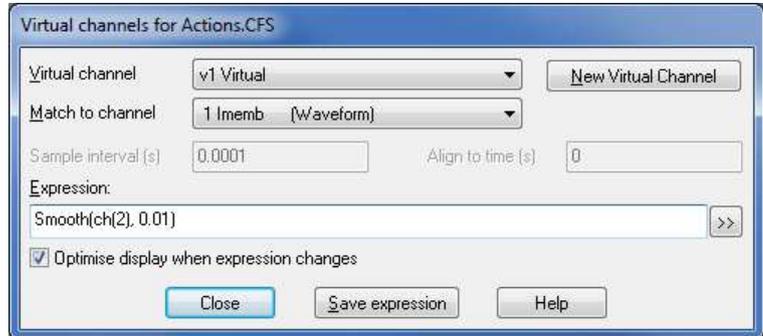
### Match to channel

You can select an existing waveform-based channel (but not a virtual channel) from which to copy the sample interval and data alignment, the virtual channel first point time, sample interval and point count will all be set to match the designated channel.

Alternatively, you can select Use manual settings and type in the interval and alignment yourself.

### Sample Interval

This field sets the sample interval between data points in the virtual channel in seconds. You can edit the sample interval if you select Manual Settings in the Match to channel field. This field accepts time interval expressions; for example, to set 27 Hz you can type `1/27`. The number of data points in the virtual channel will be set to the frame width divided by the interval.

### Align to time

This field sets the time of a data point in the virtual channel. The time of any point and the sample interval completely defines all the sample times for the channel - the first point for the channel will be at the time `Align mod Interval`. You can edit the alignment if you select Manual Settings in the Match to channel field.

### Expression

This field holds an expression that defines the virtual channel. Expressions are composed of scalars, vectors, operators, channel functions, spectral functions, marker kernel functions, waveform generation, mathematical functions, channel processes and frame information functions.

A scalar is a number, such as 4.6 or `Sqrt(2)`. A vector is a list of data values that are derived from a channel or generated by a waveform function. A function can be applied to a vector or a scalar to yield a result that is a vector or a scalar. An operator combines vector and scalars, for example 1 + 2 combines the scalars 1 and 2 to generate the scale 3. `Ch(1)+1` combines the vector `Ch(1)` with the scalar 1 to give a vector that is the waveform from channel 1 increased by 1. Select one of the following for more information:

*Example expressions*

If channels 1 to 3 hold waveforms, then `Ch(2) - 2*Ch(1)` displays the difference between channel 2 and twice channel 1.

`Sqrt(Sqr(Ch(1))+Sqr(Ch(2)+Sqr(ch(3)))` displays the square root of the sum of squares of three channels. You could use this to display the magnitude of the resultant of three perpendicular forces or movements.

`Sqr(Ch(1))` is the same as `Ch(1)*Ch(1)`, but `Sqr()` is faster. To generate a polynomial function of the input it is much quicker to use `Poly()` than to use `Ch()`, `Squ()`, `Cub()` and so on to generate a power series.

### >>

Click this button to open the Build expression drop-down list, which helps you to construct expressions interactively without needing to know the virtual channel expression syntax.

**Optimise display when expression changes**

If this button is checked then the virtual channel Y axis range will be optimised every time the expression changes to make it easier to see what the virtual channel expression is doing.

**Save expression**

There is a list of saved expressions in the Build expression drop-down list. Valid expressions are added to the list when you click the Save expression button, when you change to a different virtual channel and when you close the dialog with the Close button. The expressions are stored in the system registry.

# Operators

*Arithmetic operators*

You can use the four standard arithmetic operators plus (+), minus (−), multiply (*) and divide (/) and comparison operators together with numbers, round brackets and some mathematic and channel functions. The result of combining a vector and scalar with an operator is a vector, for example, the expression `Ch(1)+1` is a vector, being the data points of channel 1 with 1.0 added to each of them.

Dividing by a scalar value of zero is an error. Dividing by a vector holding zeros is not an error and generates special floating-point numbers for positive and negative infinities. These can cause problems in subsequent calculations (and they are difficult to display).

*Comparison operators*

You can also use comparison operators less than (<), less than or equal (<=), equal (=), not equal (<>), greater than or equal (>=) and greater than (>); the result of these is 1.0 if the comparison is true and 0.0 if it is false. In the expression `a op b`, where `a` and `b` are vectors and/or scalars and `op` is a comparison operator, if either `a` or `b` is a vector, the result is a vector with value 1.0 at points where the comparison is true and 0.0 where it is false. For example `Ch(1)>1` has the value `1.0` where channel 1 is greater than 1.0 and 0.0 elsewhere. `Ch(1)<>Ch(2)` is 1 wherever channels 1 and 2 are not the same. Be cautious using <> (not equals) and = (equals) as we are dealing with floating point numbers and exact equality may be compromised by arithmetic rounding.

*Operator precedence*

Multiply and divide have a higher precedence than all the other operators which all have the same precedence. You can use round brackets to force other evaluation orders. Apart from that, evaluation is from left to right, so `a>b*c+d` is interpreted as `(a>(b*c))+d`.

You can insert spaces between operators and numbers and between round brackets and the items within them. You may not insert spaces between a function name and the opening bracket that follows it.

# Waveform from channel

The following functions take a channel of data (including virtual channels with a lower channel number than the channel being configured) and create a vector holding waveform data at the specified sample interval and alignment.

| | |
|---|---|
| `Ch(n)` | n is a waveform or lower-numbered virtual channel. Copy data from channel `n` into the virtual channel. |
| `If(n)` | n is a a marker channel to convert to a waveform by linear interpolation of the instantaneous frequency. |
| `Ifc(n)` | The same as `If()` except that cubic spline interpolation is used if there are at least three marker points available. |
| `Rm(n,i)` | n is a real marker channel to convert to a waveform by linear interpolation of the real data values, i is the real data item to use (the first item is 0). You can omit i, omitted values are treated as 0. |
| `Rmc(n,i)` | The same as `Rm()` except that cubic spline interpolation is used. |

You can use channel expressions like m1, v2, 1a in places where a channel number is expected.

**Use of virtual channel numbers in an expression**

To prevent recursive references, you can only use a virtual channel in an expression if it refers to a lower-numbered virtual channel than the channel being defined. Although this can sometimes make things easier to visualise and allows common sub-expressions to be used by other virtual channels, this does not save you any calculation time (any may even be slower) than writing out the full expression. This is because the virtual channel is calculated whenever it is needed; it is not cached.

# Marker kernel functions

The `Ec(n,w,r)`, `Et(n,w,r)`, `Es(n,w,r)`, `Eg(n,w,r)` and `Ee(n,w,r)` functions convert data from marker channel `n` into a virtual waveform and can be used anywhere in an expression that you can use the `Ch()` function. They replace each marker by a kernel (shape), centred on the marker time. For a marker at time `t`, the kernel extends from `t-w` to `t+r` seconds except for `Ee()`, which extends from `t-8w` to `t+8w` seconds. Normally you will omit `r`, in which case the shapes are symmetrical with r set equal to `w`. The resulting waveform is the sum of the kernels for all the markers. The area of each kernel is unity, so the area under the waveform between any two times is the number of markers within that time interval.
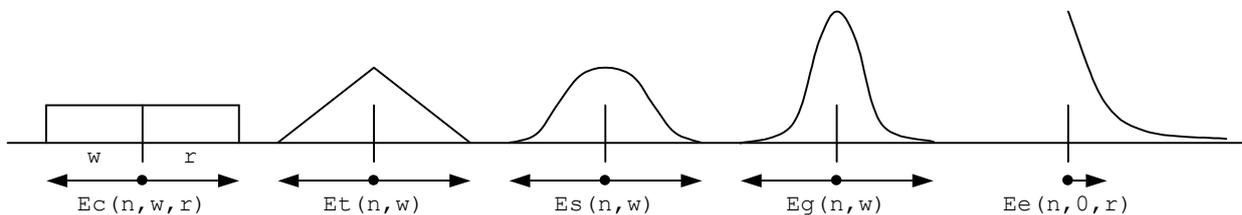


| | |
|---|---|
| `Ec(n,w{,r})` | The replacement shape is a rectangle, think Event Count, running from `w` to the left to `r` to the right. |
| `Et(n,w{,r})` | The replacement shape is a Triangle running from `w` to the left to `r` to the right. |
| `Es(n,w{,r})` | The replacement shape is a Sinusoid running from `w` to `r` to the right. |
| `Eg(n,w{,r})` | The replacement shape is a Gaussian with a sigma (standard deviation) of `w/4` to the left and `r/4` to the right. |
| `Ee(n,w{,r})` | The replacement is an exponential function with a time constant of `w` to the left of each event and `r` to the right. The kernel extends to 8 times the time constant in each direction. |

The `Ec()` function simply counts the markers in the time range. This is the fastest analysis method, but produces the most jagged output. The `Et()` function weights the markers with a triangle function. This is slower than `Ec()`, but much faster than `Es()` and `Eg()`. The `Es()` function weights the markers with a raised cosine. The `Eg()` function weights the markers with a Gaussian curve extending to 4 sigmas.

The `Ee()` function is rather different from the others as it is not suitable for use as a smoothing function and is more likely to be used in a single-sided form. It weights the markers with an exponent `exp(-t/r)` to the right and `exp(-t/w)` to the left (t stands for the time difference between the point and the time). The exponent extends to `8w` to the left and to `8r` to the right.

# Spectral functions

The following functions all calculate the power spectrum of an input channel and generate a waveform illustrating how some feature of the power changes with time. These functions are particularly useful in EEG and EMG analysis. The power spectrum is calculated using the Fast Fourier Transform (FFT), applied many times to span the data. These functions can take some time to calculate. Arguments in common to all spectral function commands are:

| | |
|---|---|
| `n` | The input waveform channel. The maximum frequency for use in bands available to any of the commands is half the sample rate of this channel. Missing data values (gaps) in the input channel are treated as zero values. |
| `f` | The desired frequency resolution of the output. For a resolution of `f` Hz, the FFT used to calculate the power must span at least `1/f` seconds of the input data. So 0.1 Hz resolution implies an FFT spanning |

10 or more seconds of input data. The FFT we implement uses a power of 2 data points, which constrains the available resolutions; we round `f` down to the nearest available value. There is a tradeoff between frequency and timing resolution. If the virtual channel sample rate is more than 8 times the frequency resolution we calculate the output for 8 times the resolution and interpolate the result. If you set `f` to 0, the FFT size is 256 points.

`l`    The low edge of a frequency range, in Hz. If this is an optional argument and you omit it, 0 is used.

`h`    The high edge of a frequency range, in Hz. if this is an optional argument and you omit it, half the sampling rate of channel `n` is used.

The available functions are:

| | |
|---|---|
| `Pw(n,f,l,h)` | Calculate the power in a frequency band from `l` to `h` Hz. There is a dialog to help you build the expression. |
| `Pw(n,f,l,h,r{,s})` | Calculate the power in the frequency band from `l` to `h` Hz divided by the power in the band from `r` to `s` Hz. If `s` is omitted, the band extends from `r` to half the sample rate of the input channel. There is a dialog to help you build the expression. |
| `SpE(n,f,p{,l{,h}})` | Calculate the frequency at which `p` percent of the power in the band from `l` to `h` Hz lies below. If `l` is omitted, 0 Hz is used, if `h` is omitted, half the sample rate of the input is used. There is a dialog to help you build the expression. |
| `MF(n,f{l{,h}})` | Calculate the mean frequency in the band from `l` to `h`. If `l` is omitted, 0 Hz is used, if `h` is omitted, half the sample rate of the input is used. The mean frequency is sum of products of frequency times power divided by the sum of the power. There is a dialog to help you build the expression. |
| `DF(n,f{,s{,l{,h}}})` | Calculate the Dominant Frequency. We search the band from `l` to `h` Hz for the frequency region with the maximum power. `s` sets the distance in Hz to smooth the data either size of each. There is a dialog to help you to build the expression. |

# Waveform generation

These functions generate waveforms without the need for an input channel, they all produce output from the start to to the end of the frame. All the arguments are in units of seconds, except the sine wave frequency, which is in Hz. All these waveforms have unit amplitude. You can use the standard maths operators `*/+` and `-` to scale and shift the output to different values.



*Generated waveforms and their alignment points*
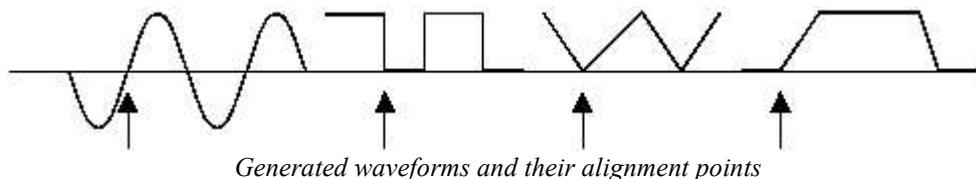
If you attempt to create a cyclic waveform with a frequency above half the channel sample rate, no output will be generated. You can generate the following outputs:

| | |
|---|---|
| `WLev(v)` | A constant level of value `v` running from the start of the frame up to the end. |
| `WSin(f,a)` | Sine wave of frequency `f` Hz running from the start to the end of the frame and aligned so that phase 0 (the point where the rising sinusoid crosses 0) is at time `a` seconds. The amplitude of the output runs from -1.0 to +1.0. The sinusoid is most accurate at the start of a frame; the accuracy falls off as the number of cycles increases (this is highly unlikely to be noticeable). |
| `WSqu(l,h,a)` | Square wave with low period `l` seconds and high period `h` seconds aligned so that a low period starts at time `a` seconds. Both `l` and `h` must be greater than 0 seconds. The output level of the low section is 0, the output level of the high section is 1. |
| `WTri(r,f,a)` | Triangle wave with a rise time of `r` seconds and a fall time of `f` seconds aligned so that a rise starts at time `a` seconds. Either of `r` or `f` may be zero, but not both. The triangle output level is from 0 to 1. |

| `WEnv(r,h,f,a)` | Envelope with a rise time of `r` seconds, a hold time of `h` seconds, a fall time of `f` seconds with the rise starting at time `a` seconds. The output waveform is 0 before time `a` and after time `a+r+h+f` and is 1 during the hold time. At least one of `r`, `h` or `f` must be non-zero. |
|---|---|
| `WPoly(s,e,r,L)` | Polynomial in time from `s` to `e` seconds, the data is 0 before **s** and after `e`. The value at time t is a function of (t-`r`) where `r` is a reference time. `L` is a list of 1 to 6 coefficients. `WPoly(f,t,r,a,b,c,d,e,f)` is: |

$$y(t) = a + b*(t\text{-}r) + c*(t\text{-}r)^\wedge 2 + d*(t\text{-}r)^\wedge 3 + e*(t\text{-}r)^\wedge 4 + f*(t\text{-}r)^\wedge 5$$

| `WT(s,e)` | Ramp of time starting at time `s` and running up to up to time `e`. The ramp value at time t is (t-`s`) with value 0 before `s` and from `e` onwards. Omit `e` to run to the end of the frame, omit both `e` and `s` to start from the beginning and run to the end. |
|---|---|
| `WExpD(s,e,r,tc)` | Decaying exponential curve starting at time s and ending at time e, the data is 0 before s and from `e` onwards. The decay curve data is 1.0 at the reference time `r` and decays to zero with the time constant `tc`. |
| `WGaus(s,e,c,sd)` | Gaussian curve starting at time s and ending at time e, the data is 0 before `s` and from `e` onwards. The gaussian curve data is at 1.0 at the centre time `c` and has a standard deviation `sd`. |

# Channel process functions

The channel process functions all have the form `Func(x)` where `x` can generally only be a vector and `Func()` is the operation. The result is a vector. Many of these functions use a time constant to specify a range of data values that will be use to produce the result, the first and last frame points are duplicated to provide the extra data needed at the edges of the frame. This duplication gives reasonable behaviour but can give odd effects if the first or last frame point is atypical; you should if possible avoid using the resulting data within one time constant of the ends of the frame. Note that the vector that is operated on will have the sample interval and offset set for the virtual channel so the data points that the `Int()`, `Dif()`, `Smth3()` and `Smth5()` functions operate on match the resulting virtual channel, not the source for the vector x.

| `Abs(x)` | The absolute value of `x` (negative values are replaced by positive values of the same size). This can also be used to generate the absolute value of a scalar. |
|---|---|
| `Hwr(x)` | Half wave rectify `x` (negative values are replaced by zeros). This function can also be used on a scalar quantity. |
| `Int(x)` | The integral of `x`, where the value at position `p` is replaced by the sum of all values from the start of vector x up to and including p, multiplied by the sample interval. This gives the total area from the start of the data. |
| `Dif(x)` | The slope of x calculated by differences, where the value at position `p` is replaced by the difference between the current value and the preceding value in vector x divided by the sample interval. |
| `Smth3(x)` | A 3-point smoothing of `x`, where the value at position `p` is replaced by the average of the 3 points in vector x around that position. |
| `Smth5(x)` | A 5-point smoothing of `x`, where the value at position `p` is replaced by the average of the 5 points in vector x around that position. |
| `Smooth(x, tc)` | Smoothing of `x` with time constant `tc`. The value at position `p` is replaced by the mean of the data in vector x over the range `p-tc` to `p+tc`. |
| `DCRem(x, tc)` | DC offset removal of `x` with time constant `tc`. The value at position `p` is replaced by the value at `p` minus the mean of the data in vector x over the range `p-tc` to `p+tc`. |
| `Slope(x, tc)` | Slope measurement of `x` with time constant `tc`. The value at position `p` is replaced by the slope measured using least-squares fitting over the range `p-tc` to `p+tc`. |
| `RMSAmp(x, tc)` | RMS amplitude of `x` with time constant `tc`. The value at position `p` is replaced by the RMS value of the data over the range `p-tc` to `p+tc`. |
| `Median(x, tc)` | Median filter of `x` with time constant `tc`. The value at position `p` is replaced by the median value (the middle point after the data has been sorted into order) of the data over the range `p-tc` to `p+tc`. |

# Mathematical functions

The mathematical functions all have the form `Func(x)`, where x can be either a scalar or a vector and `Func()` is the operation. If x is a vector, the result is a vector with the function applied to each value otherwise the result is a scalar.

| | |
|---|---|
| `Max(x,y)` | The maximum of x and y. This can be used to set a lower limit, for example `Max(Ch(1),1)` is a vector with minimum value 1. |
| `Min(x,y)` | The minimum of x and y. This can be used to set an upper limit, for example `Min(Ch(1),Ch(2))` can be thought of as the value of channel 1 with the maximum value limited by the value of channel 2 (or vice versa). |
| `Sqr(x)` | This calculates the square of x. `Sqr(x)` is the same as `x*x`, but is faster, particularly when x is a vector. |
| `Sqrt(x)` | This calculates the square root of x. When x is a vector, negative values are set to 0. If x is a scalar, negative values cause an error. |
| `Cub(x)` | This calculates the cube of x. `Cub(x)` is the same as `x*x*x` but is much faster, particularly when x is a vector. |
| `Poly(x,L)` | Replace x with a polynomial in x of order 1 to 5; L is a list of 1 to 6 coefficients. Use this to apply a non-linear calibration to a signal. For example: `Poly(x,a,b,c,d) = a + b*x + c*x*x + d*x*x*x` |
| `Sin(x)` | Calculates the sine of x (x is in radians). |
| `Cos(x)` | Calculates the cosine of x (x is in radians). |
| `Tan(x)` | Calculates the tangent of x (x is in radians). The result can be infinite. |
| `ATan(x)` | The arc tangent of x. The result is in radians in the range `-pi/2` to `pi/2`. |
| `ATan(s,c)` | Both s and c must be vectors or both must be scalars. The result is the arc tangent of `s/c` with the quadrant set by the signs of s and c. The result is in the range `-pi` to `pi`. |
| `Ln(x)` | Natural logarithm of x. Negative or zero x values generate -100 when x is a vector and an error when x is a number. |
| `Exp(x)` | Exponential of x. If the result overflows, this is an error when x is a number and sets the largest allowed result when x is an array. |

# Frame, tag and state functions

These functions all generate a scalar quantity based upon an aspect of the current frame, which makes is possible for the virtual channel behaviour to vary with the frame.

| | |
|---|---|
| `Frm()` | The result of this function is the current frame number. |
| `Tag()` | The result of this function is 1 if the current frame is tagged, 0 if it is not. |
| `State()` | The result of this function is the state code number (0 to 255) of the current frame. |

# Build expression

There is no need to remember all the expression functions; just click the >> button and choose from a list of possible items to add. All the commands in this section generate a text string that is inserted at the caret position (replacing or including where appropriate the current selection) in the Expression field. Simple commands insert the text immediately; more complex commands open a dialog to build the expression. Commands that open dialogs display the expression at the lower left corner. You can choose from:

| | |
|---|---|
| Waveform from channel | Use this to generate commands that create a waveform from an existing channel. |
| Spectral functions | Select this to generate commands that create waveforms based on the spectral content of a waveform. You can choose from Power in band |

or ratio of bands, Spectral edge, Mean frequency and Dominant frequency.

| | |
|---|---|
| Generate waveform | Create a waveform independently of any channel, for example a sinusoid or a triangle wave or an envelope function. |
| Channel processes | These commands carry out various processes upon channel data, for example rectifying, smoothing and slope measurement. Many of these operate over a specified time range, in which case a dialog is provided to set the time constant. |
| Mathematical functions | These commands insert general mathematical functions (Sqrt(), Sin(), Cos() etc and also the Poly() command which generates a poylnomial of a vector or scalar and that opens a new dialog. |
| Frame, tag and state functions | These commands provide single values derived from the current frame information. |
| Mathematical operators | Select one of these to insert the operator to replace the selection.. |

### Previous virtual channel expressions

This option lists expressions that you have used previously. Valid expressions are added to the list when you click the Save expression button, when you change to a different virtual channel and when you close the dialog with the Close button. The expressions are stored in the system registry. The most recently used expression is at the top of the list.

# Waveform from channel

Select one of these items to generate the commands that create a waveform from an existing channel. You build the commands using a dialog, all dialogs display the equivalent command in their lower left corner. The result from the dialog replaces the selection in the Expression field.

The commands in the Waveform from Channel section of the Virtual channel dialog expression field generate waveforms directly from data channels. The commands all open a dialog to select the source channel and set the parameters.

### Copy waveform

This dialog builds the Ch() command, allowing you to copy data from a waveform channel.
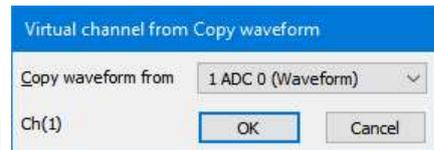
### Marker channel using kernel

This dialog lets you choose between the various Marker kernel functions: Rectangular, Triangular, Raised Sinusoid, Gaussian and Exponential. Normally you will leave the Asymmetric kernel box unchecked so that the shape that replaces each event is symmetric about the event time. The shape is scaled so that the area under it is unity (time measured in seconds).
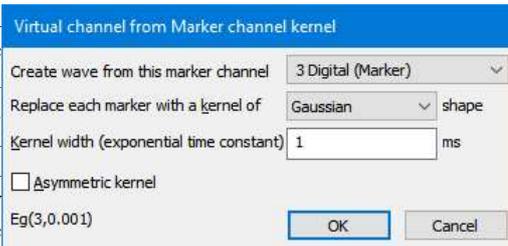
**Marker instantaneous frequency**

This dialog generates the `If()` and `Ifc()` commands, which convert a channel of marker times into a waveform of the channel instantaneous frequency. The frequency points at the markers can be linked by a straight line (linear) or by a cubic spline.
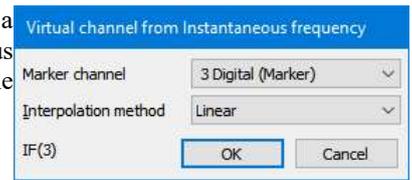
**Real marker data item**

This dialog generates the `Rm()` and `Rmc()` commands, which convert a real marker channel into a waveform using the specified real marker value. The values at the marker times can be linked by a straight line (linear) or by a cubic spline.

# Power in Band

This dialog (accessed from the Virtual Channel build expression dialog) generates a waveform channel representing either the power in a frequency band, or the ratio of power in a band to the power in another band. The fields are:

### Input waveform or RealWave Channel

Use this to select a suitable input channel to be analysed. The maximum values you can set for the frequency band is half the sample rate of this channel. If you change the channel, the frequency resolution field and the frequency range are reset to default values.

### Frequency resolution should be no worse than ... Hz

When you open the dialog or change the channel, this value is set to the frequency resolution you would get for a FFT size of 256 points (which is usually a sensible value). Follow this link for more information about the frequency resolution. The actual frequency resolution and FFT size that will be used to achieve it are displayed in the dialog.

### Frequency range for Power from ... to ... Hz
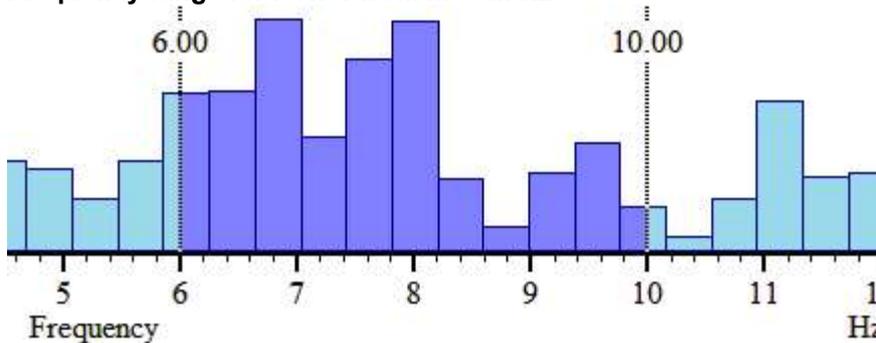
These fields set a frequency range where the power is to be calculated. The lower frequency must be less than half the sample rate of the source channel, the upper frequency must be greater than the lower. The results of the power calculation are in bins of fixed frequency width so it is unlikely that the frequencies you set will match the edges of the power bands. The power is calculated starting at the lower frequency and extending up to the upper one. If a band starts or ends with a fraction of a bin, then that fraction of the power in the bin is included.

### Divide power by power in band from ... to ... Hz

If you check the box, the result is not power, but is the ratio of the power in two bands. Usually, the second band is set from 0 to half the sampling rate, giving a measure of the proportion of the total power, but you can set any frequency range you like as long as the lower frequency is less than half the sample rate. If the band used for division does not completely overlap the first band there is the possibility of division by zero. If this occurs, the result is set to 1000000 (rather than setting infinity, which is difficult to draw).

If you do not check the box, the output waveform is calibrated in power such that the output is the mean square of the data in the frequency band. That is, if the input data were a sinusoid of amplitude $A$ units of a frequency in the band, the output would be $A^2/2$ units squared. Put another way, if the frequency band was set to include all frequencies, the output would be more or less that same as the input channel squared and then smoothed over a time of the FFT size.

# Spectral Edge

This dialog (accessed from the Virtual Channel build expression dialog) generates a waveform channel holding the frequency at which the sum of the power starting at 0 (or the low edge of a defined band) is a used defined percentage of the total power (or the total power in a defined band). The fields are:

### Input waveform or realwave channel
Use this to select a suitable input channel to be analysed. The maximum values you can set for the frequency band is half the sample rate of this channel. If you change the channel, the frequency resolution field and the frequency range are reset to default values.

### Frequency resolution should be no worse than ... Hz
When you open the dialog or change the channel, this value is set to the frequency resolution you would get for a FFT size of 256 points (which is usually a sensible value). Follow this link for more information about the frequency resolution. The actual frequency resolution and FFT size that will be used to achieve it are displayed in the dialog.

### Spectral edge as a percentage of power (in band) ... %
You set the proportion of the signal as a percentage in the range 0 to 100. If you want the Median frequency as used in some EMG work, set the percentage to 50.

### Limit calculation to band from ... to ... Hz
If you check the box, you can set limits for the band of frequencies that are used in the calculation. The range of bins used is inclusive from the nearest bin to the low frequency in the power spectrum, up to the nearest bin to the high frequency.

# Mean frequency

This dialog (accessed from the Virtual Channel build expression dialog) generates a waveform channel holding the mean frequency, defined as the sum of the product of Power and frequency divided by the sum of the power. You can limit the calculation to a range of frequencies. The fields are:

### Input waveform or RealWave Channel
Use this to select a suitable input channel to be analysed. The maximum values you can set for the frequency band is half the sample rate of this channel. If you change the channel, the frequency resolution field and the frequency range are reset to default values.

### Frequency resolution should be no worse than ... Hz
When you open the dialog or change the channel, this value is set to the frequency resolution you would get for a FFT size of 256 points (which is usually a sensible value). Follow this link for more information about the frequency resolution. The actual frequency resolution and FFT size that will be used to achieve it are displayed in the dialog.

### Limit calculation to band from ... to ... Hz
If you check the box, you can set limits for the band of frequencies that are used in the calculation of mean frequency. You could use this to exclude a DC offset from the calculation.

# Dominant frequency

This dialog (accessed from the Virtual Channel build expression dialog) generates a waveform channel holding the frequency at which the most power was found. You can limit the search to a range of frequencies. The fields are:

### Input waveform or RealWave Channel
Use this to select a suitable input channel to be analysed. The maximum values you can set for the frequency band is half the sample rate of this channel. If you change the channel, the frequency resolution field and the frequency range are reset to default values.

### Frequency resolution should be no worse than ... Hz
When you open the dialog or change the channel, this value is set to the frequency resolution you would get for a FFT size of 256 points (which is usually a sensible value). Follow this link for more information about the frequency resolution. The actual frequency resolution and FFT size that will be used to achieve it are displayed in the dialog.

### Frequency smoothing either side of each power bin ... %
If you leave this value as 0, the result is the power bin with the largest power and the result is quantised to the frequency resolution. The value you set here is divided by the frequency smoothing to give the number of extra bins either side of the each searched bin to include when looking for the maximum. If the extra bins would extend before 0 Hz or after the bin corresponding to half the sampling rate, these non-existing bins are presumed to hold 0 power. When a maximum is found, the result is the weighted mean frequency in the range of bins.

### Limit calculation to band from ... to ... Hz
If you check the box, you can set limits for the band of frequencies that are used as the centre of the search for the dominant frequency. The actual result can be outside this range if the frequency smoothing is set and a large power peak is just outside the range.

# Spectral frequency resolution

The four virtual channel functions that generate spectral functions all have a frequency resolution parameter. To best use this you need to understand how the spectral functions are calculated. Signal converts from a waveform (the time domain) to power at a given frequency (the frequency domain) using a radix 2 Fast Fourier Transform (FFT). The analysis we do takes a sequence of n input data points and generates n/2+1 output points that describe the power, spanning the frequency range 0 to half the waveform sampling rate. The output points are equally spaced in frequency.

The frequency spacing of the output points is thus the input waveform sample rate divided by the number of points, n in the transform. However, the sample rate is also the reciprocal of the sample interval, and n times the sample interval is the time duration of the input n points:

Frequency resolution = sample rate / n = 1 / (n* sample interval) = 1/(input duration)

Notice that this is independent of the sampling rate of the data. When you specify the frequency resolution, you are also specifying the time duration of the FFT transform. This means that if you want the power at time $t$, with a frequency resolution of $f$ Hz, we must transform the data from time $t-1/(2*f)$ to $t+1/(2*f)$ seconds. That is, the more accurately you want to determine frequency, the less accurately the position in time where the frequency occurred can be known.

### The details
The FFT we use does not accept arbitrary lengths of input data; it works with powers of 2 data points. If the number of samples set by a duration of 1/f seconds is not a power of two (as is generally the case), it is rounded up to the next power of two, which means that the value of $f$ used will always be less than or equal to the value that you request.

The actual value of the resolution, the FFT size used and the time period that the FFT spans are all displayed in the dialog so you can make sure that the resolution makes sense. Each waveform section that is transformed is first multiplied by a Hanning window. This weights the input data so that the values nearer to the centre get more weight in the result that those at the edges; it also prevents artefacts in the result.

If the sample interval for the virtual channel is more than half the FFT size in seconds, the result for a particular point is calculated by using overlapped transforms (the overlap factor is at least half the FFT width) of the input data from half an output sample interval before the point to half a sample interval after.

If the sample interval for the virtual channel is half the FFT size down to 1/8 the FFT size, each output point is calculated from one transform with the data centred on the output point.

If the sample interval for the virtual channel is less than 1/8 the FFT size, we calculate the power spectrum for an interval of 1/8 of the FFT size, then interpolate the values.

# Generate Waveform

The commands in the Generate Waveform section of the Virtual channel dialog expression field generate waveforms without any reference to data channels. Note that these commands generate data that coveres the entire width of the data frame as defined by other channels. If you have a completely empty data file you will need to add a channel with some data time you want to generate data for. You can generate:

### Level
This generates a `WLev()` command that extends for the entire width of the frame.

### Sinusoid
This generates a `WSin()` command that extends for the entire width of the frame. You set the frequency in Hz and a point in time at which the sinusoid rises through zero. The generated waveform has mean value of 0 and a peak to peak amplitude of 2. The waveform frequency must be less than twice the sampling rate set for the channel.
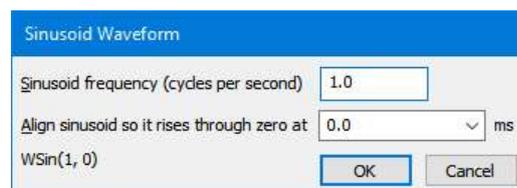
### Square wave
This generate a square wave that extends for the entire width of the frame with the `WSqu()` command. You define the wave by setting the low (waveform value 0) and high (waveform value 1) time periods and an alignment point where the waveform rises from zero. The period of the waveform (low plus high time) must be at least two sample periods of the virtual channel.

### Triangle wave
This generates a triangle wave that extends for the entire width of the frame with the `WTri()` command. The waveform is defined by a rise time and a fall time in seconds and by a time at which a rise starts. The period of the wave must be at least two sample periods of the virtual channel.

### Envelope

An envelope is zero everywhere except in one region where the value rises linearly to 1, then holds at 1, then falls linearly back to 0. It is implemented by the `WEnv()` command. It is defined by the start time, rise time, hold time and fall time, all in seconds.
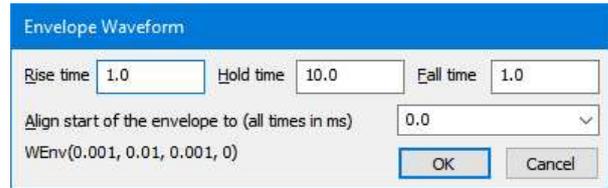
### Polynomial of time

The `WPoly()` command generates polynomials in time relative to a reference time, time being measured in seconds. The polynomial has a value in a defined time range and is zero elsewhere. Such curves can be used to create complex envelopes, or to subtract out a curve generated by the interactive curve fitting routines. The order of the polynomial can be set to: Constant, Linear, Quadratic, Cubic, Quartic or Quintic, representing the highest power of the time used. The Coefficient field selects the coefficient to set in the Value field.
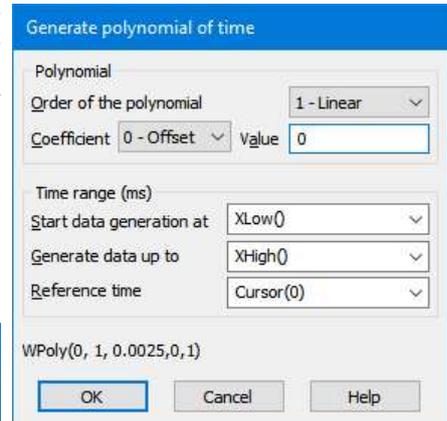
### Ramp of time

The `WT()` command generates a ramp from a start time up to, but not including an end time. The data is zero outside this time range. Within the time range, the ramp value is the current time minus the start time.
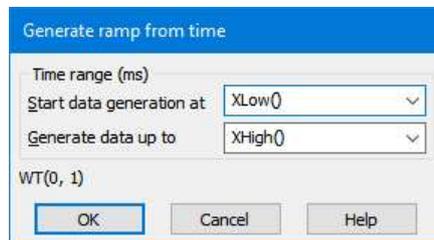
### Decaying exponential

The `WExpD()` command generates a decaying exponential from a start time up to, but not including an end time, the data is zero outside this time range. The curve is 1.0 at the reference time and decays with the specified time constant, note that a negative time constant gives you exponential growth and that the curve can be greater than 1.0 if the reference time is after the start time (or before the end time if the time constant is negative).
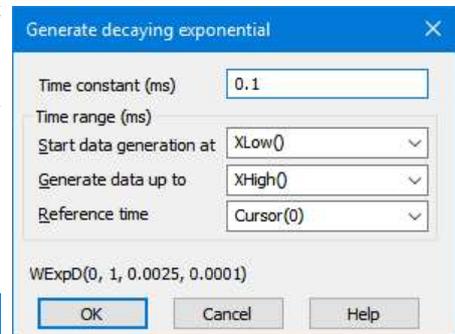
### Gaussian

The `WGaus()` command generates a Gaussian curve from a start time up to but not including an end time, the data is zero outside this time range. The maximum value is 1.0 at the defined centre point and the curve exhibits the specified standard deviation.
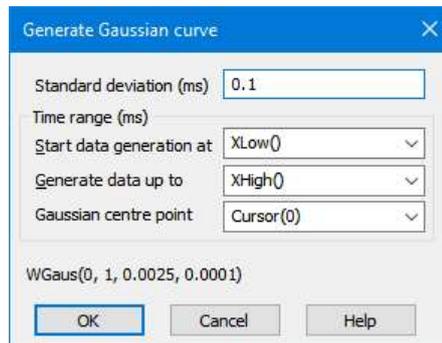
# Channel process

This dialog is provided to help you set up the various channel process functions that use a time constant:

| | |
|---|---|
| `Smooth(,tc)` | Apply smoothing with time constant tc to the selection. |
| `DCRem(,tc)` | Apply DC offset removal with time constant tc to the selection. |
| `Slope(,tc)` | Running slope measurement with time constant tc to the selection. |
| `RMSAmp(,tc)` | Running RMS amplitude measurement with time constant tc to the selection. |
| `Median(,tc)` | Apply a median filter with time constant tc to the selection. |
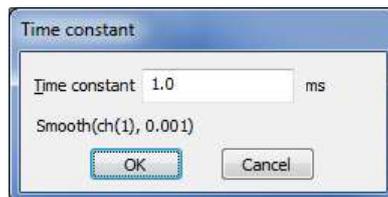
For all of these, the dialog provided allows you to enter the time constant in your preferred X axis units. For example, if the Expression field holds `Ch(1) + Ch(2)` and you want to smooth the channel 1 data before adding it, select `Ch(1)`, click the >> button, select Channel process functions, select Smooth with time constant the selection and enter the time constant required for the smoothing.

# Apply polynomial to data

This dialog is opened from the Virtual channel expression field and builds a `Poly()` expression. The Vector expression field is set to whatever was selected when this dialog was opened, or is set to `Ch(1)` if nothing is selected. This field is not tested for validity. Each element x of the vector is replaced by a polynomial in x. You can set the order of the polynomial (order means the highest power of x used) in the range 1 to 5. The command is:

`Poly(x, a0, a1, a2, a3, a4, a5)`

where *x* is the vector expression (you can also use a scalar, but this is not very useful) and the `a0` to `a5` are the coefficients of the polynomial. This expression generates the quintic:

$$y = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + a_4 x^4 + a_5 x^5$$

For lower order polynomials, omit the coefficients from the right. For example, for a quartic (fourth order polynomial), omit `a5`, for a cubic omit `a5` and `a4`. To set coefficient values in the dialog, use the Coefficient field to select the required coefficient and then set its value.
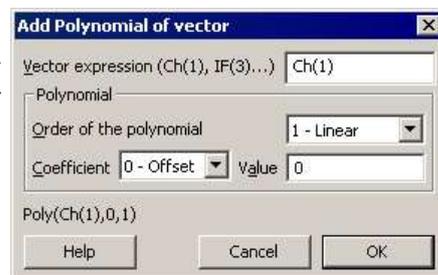
# Append frame

This command appends a new, blank, frame to the end of a file or memory view. This command can be used on a file view to generate an extra frame that will be used to hold processed data; for example a frame containing leak subtraction data that will be subtracted from other frames in the file. The extra frame can be used as part of a script, or it can be manipulated via the frame buffer.

If the command is used to append a frame to a memory view created by processing, the new frame will have its own processing parameters. When the frame is appended, the process dialog is provided to define the new frame's processing parameters. This allows for result views with different frames holding the results of processing different sets of source frames. For example, if you were sampling with multiple states, you might want to produce a multiple frame average with each frame holding the results of averaging source frames with a different state. Multiple-frame memory views, with attached processing parameters, can be saved as part of a Signal sampling configuration.

# Append frame copy

This command appends a new frame, containing a copy of the data in the current frame, to the end of a file or memory view. This command is not available for memory views created by processing.

# Delete frame

This command removes the current frame from the file or memory view. It is only available if the current frame has been appended and not yet written to disk. The last frame in a memory view and file view frames stored on disk may not be deleted.

# Delete channel

This command provides a dialog that can be used to remove a channel permanently from an XY data document or a virtual channel or idealised trace from a file view. Once the channel has been deleted, it cannot be retrieved.

# The frame buffer

The frame buffer is an extra frame of data that is automatically provided by Signal. Every open data file and memory document has a separate frame buffer that is used in conjunction with the document data, this buffer is shared by all of the views of that document. The frame buffer can be used to carry out arithmetic on frames (for example, subtract the average of frame 1 to 4 from all frames), either interactively via the commands described via the links below or by using the multiple frames dialog.

The way to think of the frame buffer is as an extra frame of data that is behind the current frame in the view. When you change to a different current frame, the buffer moves too so that it is always associated with the current frame. Understanding this association is important because all of the frame buffer arithmetic commands work with the current frame. If you are displaying the frame buffer the buffer moves so that it is in front of the current frame, but it is still closely associated with the current frame. When the buffer is shown the view title changes to show that the buffer is visible, the current frame number is still shown in brackets because the user still needs to be aware of which frame is current in order to use the buffer.

# Clear buffer

This command (keyboard shortcut `Ctrl+0`) clears the data in all channels of the frame buffer to zero. This is the initial state of the buffer after a CFS data file has been loaded.

# Copy to buffer

This command (keyboard shortcut `Ins`) copies the data in the current frame into the frame buffer.

# Copy from buffer

This command (keyboard shortcut `Ctrl+Ins`) copies the data in the frame buffer, and any count of sweeps averaged, to the current data frame.

# Exchange buffer

This command (keyboard shortcut `Shift+Ins`) exchanges the data in the frame buffer, and any count of sweeps averaged, with the data in the current frame.

# Add to buffer

This command (keyboard shortcut +) adds the data in the current frame to the data in the frame buffer. There is an alternative form of this command, available only through the `Ctrl++` shortcut and the Multiple frames dialog, which adds the buffer data to the data in the current frame.

# Subtract buffer

This command (keyboard shortcut -) subtracts the data in the frame buffer from the current frame. There is an alternative form of this command, available only through the Ctrl+- shortcut and the Multiple frames dialog, which subtracts the current frame data from the buffer.

# Average into buffer

This command (keyboard shortcut Enter) adds the data in the current frame into an average accumulating in the frame buffer. The addition is carried out in such a way that the buffer holds the average of frames accumulated. If the buffer contains data before averaging starts, this will be included as the first frame of the average. There is an alternative form of this command, available only through the Ctrl+Enter shortcut and the Multiple frames dialog, which removes the current frame data from an average in the buffer - this will not work correctly if the frame was not accumulated into the buffer average in the first place.

If you mix the buffer averaging commands with the normal addition and subtraction operations, you will find that normal addition and subtraction on the buffer 'resets' the average by setting the count of sweeps so far to one. The actual addition and subtraction act as you would expect.

# Multiple frames

This command (keyboard shortcut `Ctrl+M`) provides a dialog that can be used to carry out numerous operations on multiple frames in the document. The dialog contains a selector for the operation to be carried out, a field to enter any operation data required (this is hidden if the operation does not need it) plus a standard set of controls to specify frames in the data document to be used. The dialog can be used repeatedly by pressing the Apply button, it doesn't disappear until Close is clicked.

The operations available from the dialog include all of the frame buffer operations, all of the channel data modification options, plus tag and un-tagging frames and clearing any fits.
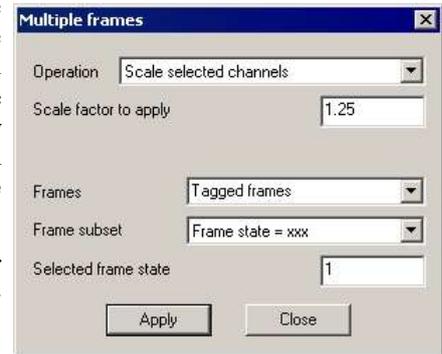
For operations that modify the frame buffer, such as accumulating an average or summing frames, the effect is straightforward. For operations that modify file view data, such as rectifying channels or subtracting buffer data from frames, the changed data must be saved to disk if the action is to have an effect (otherwise the changed file data will be discarded). This is because Signal only holds one frame from a data file in memory at a time; frames are loaded from disk as required and discarded when another frame is wanted. Use the File menu Data update mode option to ensure that changed frame data is saved, either unconditionally or by querying the user. For memory views, all of the document data is held in memory and changes to frame data are always saved.

# Modify channels

This command provides a pop-up menu specifying the data modifications that are available. All of these modifications operate on the selected waveform channels or on all visible waveform channels if none are selected. If the frame buffer is being shown then they operate on frame buffer data. In either case, all data points in the channels are modified regardless of the displayed X range. Most of the modifications are also available via the keyboard shortcuts shown in the menu. As for the frame buffer operations, changes to the frame data will be saved, or not, according to the file data update mode.

The behaviour of the modifications themselves are mostly straightforward. Subtract DC measures the mean value of the channel data, then subtracts this DC offset value from all data points. Normally, the DC level is measured over the visible frame area, the X range for the DC measurement can be set using the Area of DC item. Differentiation replaces each data point with the difference between that point and the previous point and divides the result by the sample interval; the first data point is set to zero. Integration replaces each data point with the sum of all data points up to and including that point multiplied by the sample interval. Neither integration nor differentiation are available for log-binned data. 3-point and 5-point smoothing replace each point with the average of the 3 or 5 points centred on that point. The scale and offset data options provide a dialog in which a numeric value can be entered. Scaling the data multiplies each data point by the number entered, offsetting adds the number entered to each data point.

The shift data option rotates data points by shifting data from one time to another within the frame. Note that this operation rotates; data points that fall off one side of the frame are shifted back in on the other side, so the operation can be reversed without loss of data. There are special shortcuts `Shift+<` and `Shift+>` to shift left and right by one point.

The last four options are for inter-channel arithmetic. A channel will be prompted for and this channel will be applied with the appropriate operand to the other channels on a point by point basis.

# Tag frame

This command (keyboard shortcut `Ctrl+T`) is used to tag or untag the current frame in the current view. When the current frame is tagged, this menu item is shown checked. All data frames in files handled by Signal can be tagged or untagged, the tagged status of a frame is displayed as part of the application status bar and can be interrogated by scripts. Frame tagging can be used for any purpose you require; all commands requiring a frame selection are able to operate on all tagged frames or all untagged frames. The command toggles the tag state of the current frame, changing tagged frames to untagged and vice-versa.

# Digital filters

This option provides the FIR and IIR digital filtering dialogs, which can create filters and apply them to waveform channels. See *Digital filtering* for details of these. This option will be greyed out for log-binned data

# Keyboard analysis control

Windows software is usually orientated towards control by means of the mouse and menus, but it is often convenient to use the keyboard instead. For interactive analysis of the data, using the keyboard can often be much faster. With this in mind, Signal includes keyboard shortcuts designed to handle most common data manipulation requirements:

| Channel arithmetic | Key | Frame buffer operations | Key |
|---|---|---|---|
| Zero channels | Shift+Z | Toggle display of frame buffer | Ctrl+B |
| Negate data | Shift+N | Add frame to buffer (average) | Enter |
| Rectify data | Shift+R | Add frame to buffer | + |
| Subtract DC level | Shift+O | Add buffer data to frame | Ctrl+'+' |
| Differentiate data | Shift+D | Subtract frame from buffer (average) | Ctrl+Enter |
| Integrate data | Shift+I | Subtract buffer data from frame | − |
| 3-point smooth | Shift+3 | Subtract frame from buffer | Ctrl+'-' |
| 5-point smooth | Shift+5 | Copy frame data to buffer | Insert |
| Shift 1 point left | Shift+< | Copy buffer data to frame | Ctrl+Ins |
| Shift 1 point right | Shift+> | Exchange buffer and frame data | Shift+Ins |
| | | Multiple frames dialog | Ctrl+M |

All of these shortcuts are documented with the appropriate menu commands. All analysis shortcuts are listed here for convenience. There are more keyboard shortcuts for data view and text view manipulation, and for control of sampling.

# Single-channel analysis

The Analysis menu for file and memory views contains a separate section, Open/Closed times, that is dedicated to facilities used by patch clamp researchers to analyse single channel data. To avoid confusion among non-clampers, these features will be hidden unless clamping support is enabled in the Clamp page of the Preferences dialog. With clamping support enabled, the Open/Closed times item is visible in the analysis menu and the available commands for analysis and interaction will be displayed in an attached sub-menu.

The currently available single-channel analysis commands are New idealised trace (SCAN), New idealised trace (Threshold), which generate idealised trace data from a waveform channel and Open/Closed time histogram, Open/Closed amplitude histogram and Burst duration histogram, which process previously generated idealised trace data. A final command; View and modify event details, provides the Event details dialog used for interactive generation and editing of idealised traces.

## About idealised traces

An idealised trace represents the opening and closing (assumed to be instantaneous) of an ion channel in a membrane or artificial bilayer. It shows the noise-free underlying current through the membrane that, processed though imperfect transducers and amplifiers, would generate the distinctly non-ideal signal that is actually sampled. Idealised trace data is generated by fitting the trace to the sampled data in such a manner as to most accurately represent the behaviour of the actual ion channel, taking into account such things as the sampling rate, the average level of sections of the data and the frequency response of the signal amplifiers.
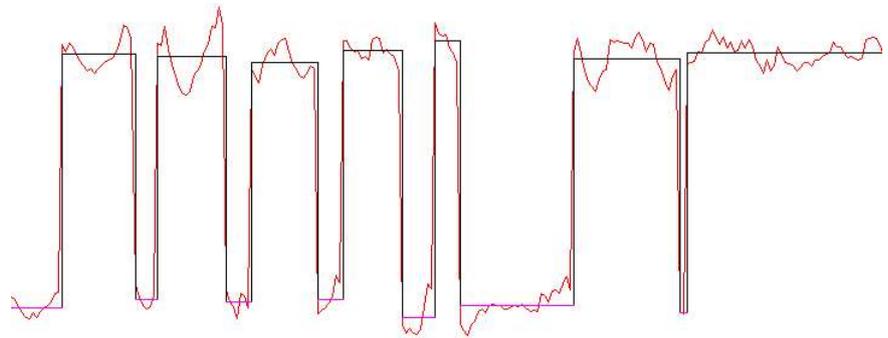
Idealised trace data is very different in a number of ways from other Signal data. Whereas normal waveform data consists of a series of measurements equally spaced through time, an idealised trace is a sequence of levels, called events, of arbitrary duration, each running from the end of the previous level up until the start of the next one. Each event has a start time, an amplitude, a duration and a set of flags to identify the type of event: a closed time; first latency etc. Because this is rather different from a sampled waveform many analyses such as waveform averaging and the cursor regions measurements are not available for idealised trace data. On the other hand other analyses such as open\closed time histograms are possible with idealised traces.

An idealised trace must be generated from a waveform before histograms based upon the trace data can be built. If you set up histograms based on an idealised trace channel where the trace has not yet been generated the histograms will still be created, but will not contain any data. If you are working on-line, it is worth noting that idealised trace data will be generated first for newly sampled data so that other analyses that depend on it will work correctly. Once an idealised trace exists, it is stored in the `.sgrx` file associated with the data file and will be re-loaded each time the data file is opened.

## New idealised trace SCAN

This analysis generates idealised trace data from a waveform using the SCAN technique which was developed by Professor David Colquhoun, to whom we owe thanks for his assistance in implementing this technique within Signal. The settings dialog provided when you use this command holds fields to determine how the trace should be generated.

Note that before you can use the SCAN method you must first perform the Baseline measurements option available in the same sub-menu.

SCAN analysis makes use of an assumption that a Gaussian filter was used to remove noise from the signal to produce a high time resolution guess of what the original unfiltered, noise free waveform was. This is a form of reverse convolution of the idealised trace using the step response function for the amplifier and other electronics. It is worth noting that the usual multiple poles Bessel filter used by most patch clamp amplifiers is a good approximation to a Gaussian filter so produces quite satisfactory results with this technique. The default draw mode for the analysis is to show the convolution over the top of the raw data with the idealised trace drawn below. Once this process has been used to produce a rough idealised trace, the trace will need to be fitted to the data using the Event details dialog to achieve maximum accuracy.
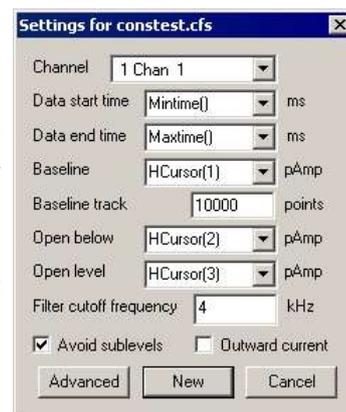
The Channel field sets the waveform channel to be fitted. The time range to be fitted is then defined by the Data start time and Data end time fields. If you are analysing voltage activated channels then you should set these times to be the start and end times of the stimulus. The first event generated by the analysis will be flagged as a first latency so if you are interested in first latency times you will need to make sure that the first event starts when the stimulus does. You should note however, that the transition times calculated will be skewed by the effects of the filter. This does not affect the durations of most of the events as they are all skewed by the same amount but first latencies will be slightly extended. If you are analysing a spontaneously active channel you should set the end time to be XHigh() as you will need to process small chunks of data at a time to keep a check on progress through the file. More details of how to do this are given at the end of this chapter.

The Baseline field is used at the start of the analysis to tell the event detection where the closed state is expected to fall. Using a horizontal cursor here will allow rhe cursor to be moved by the analysis routines to update their positions as the baseline is tracked. Baseline track will keep a running average of points in the closed state in order to correct for baseline drift during the recording.

The next field will be either Open below for an inward current where an opening is downwards on the display or Open above for an Outward current where an opening is upwards on the display. This field should be set to just outside the noise level. It is used to determine when an event is too short to be distinguished from noise.

Open Level defines the full open level. It is used in conjunction with the Advanced parameters to determine what amplitude change is significant enough to constitute a transition.

The Filter cut-off frequency is the –3 dB frequency of the Gaussian filter. If an analogue Bessel filter is used it is worth noting that the –3 dB frequency is often about half the cut-off frequency set on the front panel of the filter which uses a different definition for the cut-off.

Sometimes a transition takes longer than expected for a given filter cut-off. When this happens Signal can either insert an event at a sub-level or insert two full height transitions in opposite directions in order to make the resulting convolution approximate to the raw data. With the Avoid sublevels check box checked then the latter option will be used whenever appropriate though it does not guarantee that no sub-levels will be fitted.

The New button generates the channel to hold the idealised trace and provides the standard process dialog where you can control the processing. The idealised trace data channel is originally positioned in the file view on top of the original waveform channel, the drawing is set up so that the convoluted trace is shown on top of the waveform, with the idealised trace itself offset below.

# Advanced

The algorithm used by the SCAN technique is highly complex. The Advanced button allows various parameters used in the trace formation to be changed. The basic principle of the trace formation is that an average is kept of the point amplitudes of the data. When a number of consecutive points fall outside a critical level then a transition is deemed to have taken place. The critical level is defined as a percentage of the difference between the baseline and the full open level. To begin scanning for the next transition imediately would probably result in another transition being detected straight away when in fact it was just part of the same transition already found. For this reason the scanning jumps ahead by an amount to get past the transition just found. This amount is specified as a percentage of the filter length; the filter length being defined as the time taken for the step response to get from 1% to 99% of the step amplitude. For a Gaussian filter this turns out to be 0.6165062/fc where fc is the –3 dB filter cut-off frequency. Rise time is defined as 0.3321412/fc.

An average of the data point values found while looking for a transition is kept and used as the default amplitude for an event. If this amplitude is less than the critical level from the baseline then the event is flagged as closed. The data skipped over by the transition detection is checked for turning points that might indicate that a transition in the oposite direction to the previously detected one had been missed. If a turning point is found, a transition is inserted at a time calculated from the amplitude of the turning point and an assumption that the original data reached full amplitude or was closed.

Consecutive transitions in the same directions can either mean a sub-conductance level or a pair of transitions have been missed. If the Avoid sublevels box has been checked then by looking for turning points in the first derivative of the raw data the times of transitions can be deduced or their presence eliminated. Any missed events are assumed to be of full amplitude or closed and are flagged as assumed amplitude.

Transitions having less than a certain amplitude are stripped out in a final pass. At the fitting stage in the Event details dialog amplitudes having the assumed amplitude flag set are held fixed then a second pass of fit is made freeing up assumed amplitudes above a certain duration. The Advanced parameters dialog can be accessed from the Event details dialog by clicking the Parameters button.

# New idealised trace Threshold

This analysis generates idealised trace data from a waveform using a threshold crossing technique. Threshold crossing as a method for generating an idealised trace is needed if you have more than one channel in your patch or if you want to make use of a simpler technique and are not concerned with a very high time resolution result.

The first three fields in the setting dialog set the channel to be analysed and the time range to analyse in exactly the same way as for the SCAN method settings.

The next two fields set the thresholds to use for detecting openings. These are labelled Open above and Close below for outward currents or Open below and Close above for inward currents. Having two thresholds in this way provides some protection against false events being detected, which are actually noise. It is assumed that these thresholds will be positions approximately around the half-way mark between the baseline level and the open level, so if the open level was approximately 1.8 pA then the thresholds would be placed around 0.9 pA; at 0.8 and 1.0 pA for example.

The Base level field is mostly used in multiple level analysis, where the thresholds for subsequent levels are calculated by averaging the two thresholds and then doubling the difference between this average and the base level. The result is assumed to be the current per channel opening for subsequent levels and so subsequent thresholds are calculated by adding multiples of this value to the original thresholds. This value also sets the reference (baseline) level for event amplitude measurements when baseline tracking is not in use.

Transition points can be set to a number greater than zero to provide a 'dead time' after a transition before another transition can be detected. This allows for settling of data which has been filtered.

The Baseline track field sets the number of points over which the baseline level is averaged to produce a current baseline level. The current baseline level is used to set the level for closed events and baseline level for all events, plus all thresholds are shifted to match shifts in the current baseline level. Set this field to zero to disable baseline tracking, in which case the level of a closed event is set by the average value of the waveform making up this event and the baseline level for all events is set by the Base level field.

By default an event will start at the first data point found to be across a threshold and will have an amplitude which is the average amplitude of all the data point within the event period. The Interpolation field allows the start time of the event to be calculated by extrapolating the data around the transition to find the exact time that the threshold was crossed. Currently only linear interpolation is available, this assumes that a straight line can represent what happens between sample points.

Select Multiple level if you have more than one channel in your patch or if other circumstances give your preparation more than one open level. Outward current means that a channel opening produces a more positive current; a commonly used convention. An inward current would produce a more negative current when a channel opens.

The New button generates the channel to hold the idealised trace and provides the standard process dialog where you can control the processing. The idealised trace data are originally positioned in the file view on top of the original waveform.

# Open Closed time histogram

This analysis generates a histogram showing the relative frequency of events of various durations, the flags associated with trace events are used to select which events are included in the analysis. The command opens a settings dialog which defines the analysis and histogram parameters.

The Channel field selects the idealised trace channel that will be analysed. The Bin width, Number of bins and Maximum duration fields define the histogram that will be created; the x-axis of the result starts at zero (unless log binning is used) and the Bin width and Number of bins fields are linked so as to remain consistent with Maximum duration.

If the check box marked Use log binning is checked then the Bin width field changes the Minimum duration. Log binning means that the width of each bin increases geometrically so that when the resulting

histogram is drawn with a log x-axis, the bins appear to occupy a constant number of pixels across the histogram. The contents of each bin are divided by the bin width to maintain the overall shape of the histogram as if it were binned normally.

The two sections labelled Include and Exclude represent the flags associated with each event and are used to control which events are included in the histogram and which are not. An event will be included in the histogram if at least one flag set for the event matches the Include set specified and none of the event flags match the Exclude set. The flags are:

| | |
|---|---|
| Open time | An event where the channel is open. |
| Closed time | An event where the channel is closed. |
| First latency | The first idealised trace event in the frame. |
| Truncated | The last idealised trace event in the frame. |
| Assumed amp. | An event where the amplitude is not an average of the raw data points. |
| Bad data | Event flagged as not suitable for analysis. |
| Level n | Six flags, one for each level of multiple level data. The Level 1 flag is set for all closed times as well as the first open level. |
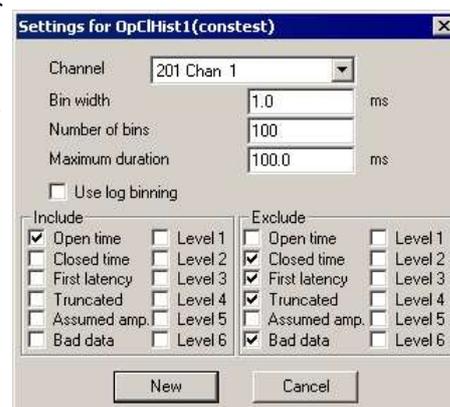
# Open Closed amplitude histogram

This analysis generates a histogram showing the relative frequency of occurrence of events with specific levels, the flags associated with trace events select which events are included in the analysis. This analysis is almost identical in concept to the amplitude histogram for waveform data described previously. The differences are that the y-axis is a count of trace events at a level rather than the time spent at that level and that the analysis can select events for analysis using the Include and Exclude flags for the analysis in the same manner as the Open/Closed time histogram.

The Channel field in the settings dialog selects the idealised trace that will be analysed.

The Maximum amplitude and Minimum amplitude fields set the amplitude range that will be divided into bins and thereby set the x axis range in the memory view holding the histogram. The Bin size and Number of bins fields both set the number of bins that cover the amplitude range, whenever you change one of these items the other one will change to match.

By default, the amplitudes added to the histogram will be absolute amplitudes as measured by Signal. The amplitudes can be measured relative to the baseline by checking the box labelled Amplitudes relative to baseline.

The Include and Exclude sections control which events are included in the analysis in exactly the same manner as for the Open/closed time histogram analysis.

# Burst duration histogram

This analysis generates a histogram showing the relative frequency of occurrence of event bursts of different durations, the flags associated with trace events are used to select which events are part of a burst and which events can terminate a burst. The command opens a settings dialog which defines the analysis and histogram parameters.

The Channel field selects the idealised trace channel to be analysed. The Bin width, Number of bins and Maximum duration fields define the histogram that will be created; the x-axis of the result starts at zero (unless log binning is used) and the Bin width and Number of bins fields are linked so as to remain consistent with the Maximum duration.

If the check box marked Use log binning is checked then the Bin width field changes to Minimum duration. Log binning means that the width of each bin increases geometrically so that when the resulting histogram is drawn with a log x-axis, the bins appear to occupy a constant number of pixels across the histogram. The contents of each bin are divided by the bin width to maintain the overall shape of the histogram as if it were binned normally.

The Critical interval field defines how an event burst can be terminated; a burst is terminated by a non-included event whose duration is longer than the critical interval.

The Include and Exclude sections control which events can start or terminate a burst. An event can start a burst if at least one flag set for the event matches the Include set specified and none of the event flags match the Exclude set. All events that cannot start a burst are capable of terminating it. A burst begins with any suitable event and is terminated by any suitable event whose duration is greater than the critical interval. The burst duration runs from the start of the first event in the burst to the start of the event that terminated it.

# Baseline measurements

The Baseline measurements command must be used before any SCAN analysis can be carried out. When the command is use a dialog is presented asking for a time range over which the baseline is to be measured and the channel on which to carry out measurements. Clicking OK produces a message window showing the mean data point value in the time range; the standard deviation of the measured values and the standard deviation of the first derivative of the measured values. This information is also be written to the log window and saved internally for use by the SCAN analysis.

# View and modify event details

The View and modify event details command is available when the current view contains idealised trace data. It opens a dialog that allows information about individual events to be viewed and changed. Click on a horizontal portion of the idealised trace to select an event or on a vertical section to select the following event. You can also click and drag the idealised trace to change transition times and levels. The event whose details are displayed in the dialog will have a small box drawn at both ends to indicate that it is the current event.

The event details dialog shows information about the current event. The Start time, Duration and Amplitude items and all of the Flags can all be changed manually. Any start time you enter must be between the start times of the neighbours. Changing the start time adjusts the event duration so that the end time remains unchanged. The duration must be a positive value and cannot extend the event beyond the end time of the following event. The Baseline level, normally calculated by averaging points in the closed state, is also displayed. The Resolution item will be shown if the idealised trace has a SCAN process attached to it. It shows the minimum duration of full amplitude opening needed for the data trace to reach the trigger level. Each event also has a number of flags associated with it and these may be set or unset using this dialog. These flags and their uses are explained in the documentation of Open/Closed time histogram generation. The buttons below the event details provide many useful functions:

### Merge
Merge combines the current event with the one to the right to produce a single event with the combined duration of the two events but the attributes of the first.

### Chop
Chop will break the current event in two. If the current event has an amplitude between those before and after then the first and second new events created by the break will be given amplitudes and attributes from the following and preceding events respectively.

### Delete
The Delete button will delete both the current and following event and extend the preceding event to cover the time gap created. This event will have an amplitude adjusted to the weighted average of all the events previously covering the time period.

### Split
Spilt will divide the current event into three separate events all having the same amplitude and duration.

### Fetch Amp
Fetch Amp will scan backwards through the trace to find an event of the same type and copy the amplitude of this previous event to the current event.

### Scroll On and Scroll Back

Scroll On and Scroll Back will do the same thing as Previous and Next but will do so using a smoothly scrolling display. Repeatedly hitting the scroll buttons will double the scroll speed each time. To stop the scrolling either click on the Next, Previous or the other scroll button or simply click on the data window.

### Fit Visible

The Fit Visible button will adjust the time and amplitude of the transitions using the filter cut-off frequency defined during the SCAN process to build a step response function to fit. Idealised traces created using threshold crossings will be fitted using the Nyquist frequency as the cut-off frequency unless this frequency is changed in the Channel information dialog. Amplitudes having the assumed amplitude flag set are held fixed then a second pass of fit is made freeing up assumed amplitudes above a certain duration. This duration is defined in the Advanced Parameters dialog, which can be accessed by clicking the Parameters button. If you do Fit Visible when the end of the display is beyond the least event of the idealised trace then a process will be done to extend the idealised trace before a fit is done. After fitting either the first event to fail to be fitted will be made the current event or the last displayed event will be.

### Undo

Press this button to undo the last operation. Up to 100 operations can be undone.

### Previous and Next

The Previous and Next buttons step the current selection of event through the idealised trace forwards and backwards respectively. Stepping past the end of the idealised trace will cause the display to jump on to the next possible opening ready for processing. If this is done there is then a short "dead" period during which the Next or Scroll On keys do nothing after that pressing either Next or Scroll On will extend any closed time terminating the idealised trace to include the displayed data.

### Parameters

The parameters button displays the advanced parameters dialog for SCAN processing. Though the button and dialog are still available for threshold analysis, the settings do not affect the processing.

### Process

This button causes the process for generating the idealised trace to be called. If the process settings are such as to process `Mintime()` to `XHigh()` then the trace will be extended from the current end time to the end of the displayed data.

# View event list

The View event list command opens a new window showing a list of idealised trace event durations and amplitudes. Up to 10 events are shown (drag the right edge of the window to change the number of events visible) with the currently selected event on the trace being shown highlighted in the list. Information for events in the closed state is shown using the colour set for drawing such events. You can click on an event in the list to select it and this will be reflected in the data trace as well as the Event details dialog.
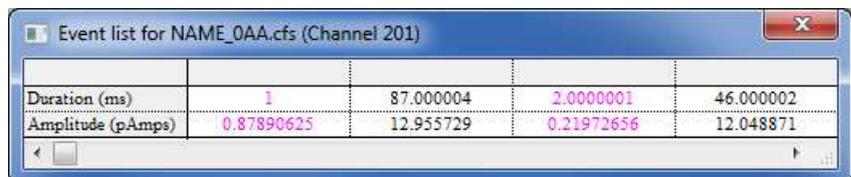
| | | | | |
|---|---|---|---|---|
| Duration (ms) | 1 | 87.000004 | 2.0000001 | 46.000002 |
| Amplitude (pAmps) | 0.87890625 | 12.955729 | 0.21972656 | 12.048871 |

# Export to HJCFIT

This command exports idealised trace data to a `.scn` data file suitable for use by HJCFit and EKDIST. HJCFit is an analysis program used for modelling channel states and calculating the rate constants for each possible state change. EKDIST is a separate program used to build and plot histograms. These applications take an idealised trace as input, this needs to be in a `.scn` format file. HJCFIT and EKDIST are not CED software, they are written

and maintained by Prof. David Colquhoun of UCL. For more information on these programs see http://www.ucl.ac.uk/Pharmacology/dcpr95.html.

## Short cuts

Because of the intensely interactive nature of idealised trace generation using the event details dialog, you can set keyboard shortcuts so that a single keypress is the equivalent of clicking on a button. These shortcuts can be freely adjusted to match your preferences, the settings are saved in the user-specific application data directory so that they can be different for each user. Two standard configurations of shortcuts can also be set using the Numeric Defs and the Letter Defs buttons. Letter Defs will set the standard shortcuts as indicated by the underlines on the event details dialog. Numeric Defs will select a set of shortcuts that uses the numeric keypad. This is to provide the keys close together by the right hand to speed up interaction as much as possible. If you use numeric shortcuts you should remember to have Num Lock on.

## Tips for fitting

Occasionally the fitting routines may produce unexpected results and there can be a number of reasons for this.

1. There is too much level data at the start and end of the fit. Restrict the visible area to only view the transitions you are interested in fitting.

2. The initial guess is too good. When the initial guess for the fit is so close to the "correct" solution the fitting routines can be unable to detect which direction to move the parameters to improve the result. This can be overcome by worsening the initial guess manually first. Dragging the amplitude of a principle event usually works best.

3. The initial guess is too poor. Use the other edit functions to produce a more plausible guess.

4. An amplitude needs fixing. If you flag an event as having an assumed amplitude it will be held fixed for the fit. If it is a very short event the assumed amplitude flag will remain set otherwise the fit routine will clear the flag.

## Strategy for long recordings

Although it is possible to analyse a long single channel recording by simply zooming all the way out and processing the entire recording in one go then hitting Fit Visible to give your result, this is not to be recommended. Subtle changes in conditions throughout the file combined with mistakes in the processing caused by noise conspire to make this an error prone strategy. An incremental approach is needed and here is a suggested approach for doing this.

The SCAN method is by far the superior method for event detection. It copes well with sub-conduction bands and can detect events far shorter and with far higher time resolution than would be possible with threshold crossing. To begin with, select an area of data near the start of the recording with a full opening and an area of baseline showing. Place a horizontal cursor on the baseline, another on the full open level and a third between the two towards the baseline but out of reach of the noise. Once these have been set up you can now evoke the SCAN settings dialog via the analysis menu. Use the drop downs in the dialog to select the relevant horizontal cursors for the different levels. Select a Data Start Time of Mintime() and a Data End Time of XHigh(). Long recordings will normally only be a single frame of data, however, if there are multiple frames you will want to process just the first one to start with. Once the initial process has been done an idealised trace will appear. This is just the initial guess and will now need refining. Zoom in on the first opening or burst so you can see the transitions in more detail.

The convolution of the idealised trace with the step response function for the amplifier will be drawn on top of the data. By default this will be in blue; indicating that the trace has not yet been fitted. At this stage you may decide that the initial guess is poor and edit the trace to something more plausible. Once you are happy with the guess, click on Fit Visible and the convolution should jump to fit the data and turn black; indicating the fit was successful. If it does not then the initial guess may need to be refined further (See Tips for Fitting).
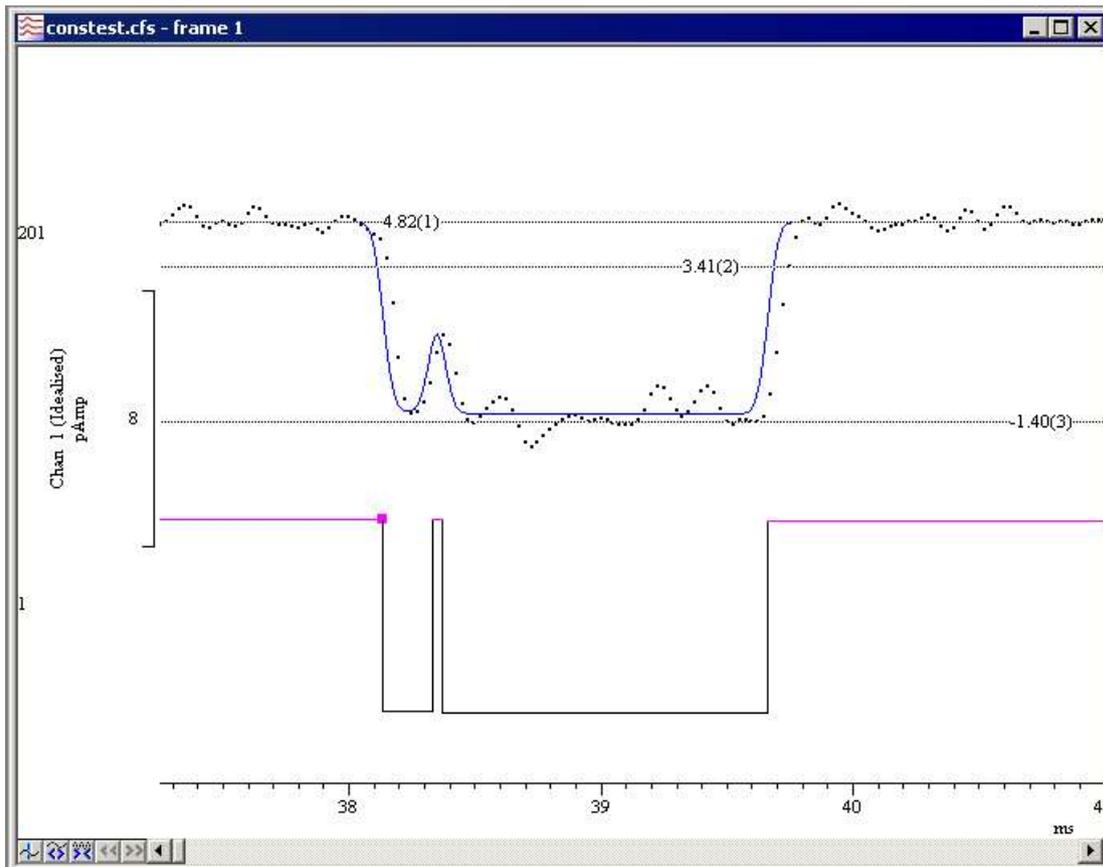
Once you are happy with the idealised trace on the screen, step forward using either the Next or Scroll On buttons. These will take you either to the next part of the idealised trace to work on or to the next possible opening past the end of the idealised trace. If it is the latter you can then decide if this possible opening is real or just an artefact. If it is an artefact you can step past it using either the Next or Scroll On buttons, optionally setting the Bad data flag. If a possible opening extends past the end of the display you may like to adjust the display range to include the full open time. The incremental analysis will work better if breaks in the middle of openings are avoided though if this is not possible you can still edit any discontinuity that appears as a result and re-run the fit. A fit done on a section of data with no idealised trace fitted will generate a guess first. Closing the data file at any time will save the idealised trace and process settings to disk allowing the analysis to be spread over several sessions.

# Cursor menu

The cursor menu creates and destroys vertical and horizontal cursors, changes the display to make them visible, changes their labelling mode and obtains the values of channels where they cross vertical cursors and between vertical cursors. Up to 10 cursors of each type, numbered 1 to 10, can be present in each data view, there is also an extra vertical cursor, number 0, which is used for special purposes. Cursors can be dragged over and past each other using the mouse. Cursors in separate windows are independent of each other. When a window is duplicated, the cursors are also duplicated.

Cursors can also be active so that they automatically position themselves by searching or measuring the channel data.

Options within the cursor menu allow you to take measurements at the cursor positions or between pairs of cursors.

Cursors and their labels can be dragged using the mouse, right-clicking on a cursor will provide a popup menu offering useful options including label control and setting the cursor position.

Mechanisms for keyboard control of the cursors are also available:

New vertical cursor       `Ctrl+|`
Show cursor number n     `Ctrl+n (0 to 9)`

## Vertical cursors

The following commands in the Cursor menu are used to control and interact with the vertical cursors. There are also vertical cursor context commands available by right-clicking on a vertical cursor.

## New Cursor

This menu command (keyboard shortcut `Ctrl+|`) duplicates the action of the new cursor button at the bottom left of data views. The command is available when a data view is the current window and there are less than ten cursors already active in the view. A new vertical cursor is added at the centre of the window. The cursor is given the lowest available cursor number and is labelled with the cursor label style for the window.

## Delete

This command opens a pop-up menu in which you select a cursor to remove, or you can delete all cursors. The cursors are listed with their number and position as an aid to identification. Deleting a cursor removes it from the view; other cursors are not affected.

## Fetch

This opens a pop-up menu where you select a cursor to place in the centre of the x axis.

## Move To

This command activates a pop-up menu from which you can select the cursor to move to. The cursors are listed with their number and position as an aid to identification. The window scrolls to show the cursor in the screen centre, or as close to it as possible.

## Position Cursor

This command opens a pop-up menu to select a cursor and then opens a dialog in which you can type or select a cursor position. You can also activate this dialog by right clicking on a vertical cursor and selecting Set Position from the cursor context menu.

# Display All

This command has no effect if there are no cursors. If there is a single cursor, the command behaves as though you had used the Move To command and selected it. When there are multiple cursors, the window is scrolled and scaled such that the earliest cursor is at the left-hand edge of the window and the latest is at the right-hand edge.

# Label Mode
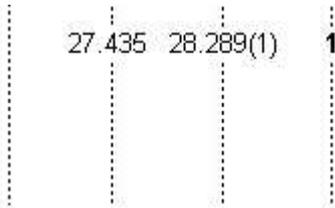
27.435  28.289(1)     **1**

Each cursor has an optional label used to identify it. You can drag the cursor labels up and down the cursor with the mouse to suit the data. There are five cursor label modes: None, Position, Position and Number, Number and User-defined. Select the most appropriate mode for your purposes from the pop-up menu. To avoid confusion between the cursor number and the position, the number is displayed in **bold** type when it appears alone and bracketed with the position. The style applies to all the cursors in the window. You can drag the cursor labels up and down the cursor with the mouse to suit the data.

Each cursor stores its own mode and label, and the view has a mode that is applied to new cursors. The first four items in the menu set the view mode and the mode of all cursors. You can also set a user-defined label for cursors (but not for the view) with the Set Label command.

# Set Label

You can open the cursor label dialog from the Label Mode and Horizontal Label Mode cursor menu commands or by right-clicking on a cursor and using the Set Label command from the cursor pop-up menu.

From this dialog you can set the cursor label mode for one or all cursors. The Cursor field can be set to the number of any cursor in the view, or All. If you choose All, any change applies to all cursors and sets the view mode except in User-defined mode, where the view mode does not change.

The Label Mode field lets you choose one of None, Position, Number, Position and Number, and User-defined as the cursor mode. If you select User-defined, the Label field appears together with the >> button and you can set a label of your choosing.

### User-defined labels

User-defined labels display the text you type, except that text sequences introduced by `%` are replaced:

| | |
|---|---|
| `%p` | Replaced by the cursor position |
| `%u` | Replaced by the position units as displayed on the axis |
| `%q` | Equivalent to `%p %u` (position followed by units) |
| `%n` | Replaced by the cursor number |
| `%v(n)` | Replaced by the value of channel n where it is crossed by the cursor. This is not valid for horizontal cursors or in an XY view. |
| `%w(n)` | The same as `%v(n)` but followed by the units of the value. |

You can also stipulate the width (W) and the number of decimal places (D) used for the position and value by using `%W.Dp` and `%W.Dv(n)`. For example: `%n at %6.4p %u, %v(2)` might display: `1 at 2.2346 s, 87.128756` if cursor 1 was at 2.2346 seconds and channel 2 had the value 87.128756 at this point. The `%v(n)` option is not allowed for horizontal cursors or for vertical cursors in an XY view. The value returned by `%v(n)` is the same value as displayed on the axis for that cursor and channel.

The >> button pops up the list of replacements, and if you choose one, it replaces any selection in the Label field. If you choose the `%v(n)` option, you are prompted to select a channel to measure.

We allow you to type in quite long labels. However, when you close a data file, only the first 19 characters of a user-defined label are saved and long labels look messy, so it is usually a good idea to keep labels short.

# Renumber

This command renumbers vertical cursors 1 to 10 by position, with cursor 1 on the left.

# Horizontal cursors

In many respects, horizontal cursors are similar to vertical cursors. However, they differ significantly in that each horizontal cursor is attached to a channel, or more accurately, to the y axis of a channel. If you change the drawing mode of an event channel with a horizontal cursor to a mode that has no y axis, the horizontal cursor will treat the available vertical space as if it runs from 0 at the bottom to 1 at the top.

If you drag channels with horizontal cursors on top of each other, only the horizontal cursor for the topmost channels (i.e. the channel with a valid y axis) is visible. In the special case of channels drawn with a locked y axis and a group offset of 0, all horizontal cursors for all channels are visible because the y axis is valid for all the channels.

These commands in the Cursor menu let you control and interact with the horizontal cursors. There are also horizontal cursor context commands available by right-clicking on a horizontal cursors.

# New Horizontal Cursor

This menu command duplicates the action of the new horizontal cursor button at the bottom left of data views. The command is available when a data view is the current window and there are less than ten horizontal cursors in the view. A new horizontal cursor is added at the centre of the data for the lowest numbered visible channel. The cursor is given the lowest available number and is labelled using the horizontal cursor label style for the window.

# Fetch Horizontal

This opens a pop-up menu where you select a horizontal cursor that is placed in the centre of the visible y axis for the relevant channel.

# Delete Horizontal

The delete command activates a pop-up menu from which you can select a horizontal cursor to remove, or you can delete all of them. The available cursors are listed with their number, position and channel number as an aid to identification. Deleting a cursor removes it from the window; other cursors are not affected.

# Move To Level

This command activates a pop-up menu from which you can select the horizontal cursor to move to. The cursors are listed with their number, positions and channel as an aid to identification. The Y axis of the relevant channel will be scrolled to display the nominated cursor in the centre of the axis, or as close to the centre as possible. This command does not change the y axis scaling.

# Position Horizontal

This command opens a pop-up menu in which you can choose a horizontal cursor and then opens a dialog in which you can set the position and channel for the cursor. You can also open the dialog by right clicking on a horizontal cursor and selecting Set Position from the cursor context menu.

# Display All Horizontal

This command is the equivalent of using the Fetch Horizontal command for all cursors.

# Horizontal Label Mode

Each cursor has an optional label used to identify it. You can drag the cursor labels to the left and right with the mouse to suit the data. There are five cursor label modes: None, Position, Number, Position and Number, and User-defined. You select the most appropriate for your application using the pop-up menu or by right clicking on a cursor and choosing to set the cursor label from the context menu. To avoid confusion between the cursor number and the position, the number is displayed in bold type when it appears alone and bracketed with the position.

Each cursor stores its own mode and label, and the view has a mode that is applied to new cursors. The first four items in the menu set the view mode and the mode of all cursors. You can also set a user-defined label for cursors (but not for the view) with the Set Label command.

# Renumber Horizontal

When created, cursors take the lowest available cursor number rather than being ordered by position. You can also drag cursors over each other, confusing the ordering further. This command renumbers the cursors by position, with cursor 1 at the bottom.

# Active cursors

Normally, Signal cursors are static; they stay where they are put. Using the active cursor dialog, cursors in time and memory views can be made active; they will move to the position of a data feature, if it can be found. This search is repeated whenever the view data changes, cursor 0 is iterated, or the view switches to a different frame or the frame buffer. This repositioning is carried out in order of cursor number so cursor 2 can reliably make use of the current position of cursor 1 but not vice-versa. Active cursors can be used as a simple way of quickly finding features within your data; they are also very powerful tools for extending the capabilities of analyses that generate measurements from data files.

### Cursor 0
Vertical cursor 0 is special. It always exists and cannot be deleted, but it can be hidden (and will normally be initially hidden by Signal). Unlike the other cursors, when cursor 0 is active it is designed to iterate through the data to repeatedly find features in the waveform. Whenever cursor 0 is moved, all of the other active cursors will recalculate their positions in order of cursor number. To hide cursor 0, right-click on it and select Hide in the cursor 0 context menu. Script users can cause a nominated range of cursors to search with the CursorSearch() command.

### Valid and invalid cursors
Active cursor positions are either *valid* or *invalid*; invalid cursors have an exclamation mark at the end of the label. The cursor position is invalid if the active search fails and the Position if search fails field is empty or does not contain a valid expression. Expressions that use invalid cursor positions are also invalid. The XY Trend plot and Measurements analyses reject points that come from invalid measurements. Cursor positions are made valid by any operation that moves them to a specific place such as dragging. If a search results in an invalid position, the cursor is not moved. Script users can test if a cursor has a valid position with the CursorValid() command.

### Previous cursor position
From Signal version 6.05, each time a cursor moves we save the old position. You can access this with the CnX dialog expressions and the CursorX() script command. This can be useful with iterative searches and also to position a cursor the same distance from cursor 0 (for example) as it was in the previous search.

### Cursor positions after a failed search
Cursor 0 does not move after a failed search. The remaining cursors are set to result of evaluating the Position if search fails field. If this field is empty, or if the expression fails to evaluate, the cursor is set to the mid-point of the search region(which may be inconveniently far away). A useful trick is to position a failed search cursor at the same relative position to cursor 0 as for the previous search, for example by setting the Position if search fails for cursor 1 to C0-C0X+C1. This takes advantage of the fact the at the time of the cursor 1 search, C1 is the old cursor 1 position, and C0-C0X is the distance that cursor 0 moved.

# Active mode

A wide variety of active cursor search modes are available and can be used to find many types of feature in your data including spike peaks, other nervous system signals such as EPSPs, evoked responses and periods of data without activity (silent periods). This command opens a dialog from which you can select a cursor and set up its active search mode. The contents of the dialog depends upon the Search method field and the selected cursor. An active cursor has an associated Search channel and start and end positions for the search that define the data within a frame that is searched. To make it easier to keep track of which cursor is which the cursor selection control displays the cursor label and the '...' button to the left of the cursor selector provides the cursor label mode dialog for the currently selected cursor.
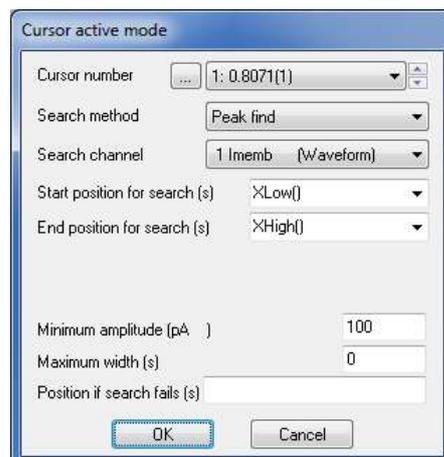
Cursor 0 does not have a Position if search fails parameter. However, it does have a Minimum step field; iterations of cursor 0 forwards or backwards will reject features that are closer to the previous feature than this limit. Cursor 0 also has a restricted range of search methods: Peak find, Trough find, Rising threshold, Falling threshold, Outside dual thresholds, Within dual thresholds, Slope peak, Slope trough, +ve slope threshold, -ve slope threshold, Turning point and Expression.

The start and end search positions can be a fixed time, but they will often be expressions that involve the positions of other active cursors, particularly when cursor 0 is being used to iterate through features. For cursor n, an expression that refers to an active cursor less than n refers to the new cursor position. An expression that refers to a cursor greater than or equal to n refers to the old position – this is to be avoided as it will cause odd effects when switching between data frames.

If the End position for search is less than Start position for search, searches go backwards through the data. If a search is backwards, read *previous* for *next* and *last* for *first* in the descriptions of the search methods. As far as is possible, we have designed searches that run backwards to work in a similar manner, and to find the same positions, as searches that run forwards. However you will find that threshold crossing searches that use the Delay after crossing parameter will behave differently when running backwards.

Several modes use the slope of a waveform. These methods all have the field Width for slope measurement, which sets the time range over which slopes are calculated. Signal uses the data points from Width/2 before the current point to Width/2 after to calculate the slope unless there are more than 2000 points, in which case 1000 points before and after are used. The contribution of each point to the slope is proportional to the distance of the point from the current position. Because the slope at any point uses data around it, the slope within Width/2 of the ends of the data frame cannot be relied upon.

The active cursor modes that are available are:

### Static

When you add a new cursor, it starts off in Static mode. In this state, the cursor stays where you put it; it is not changed by a change in the data or the position of a lower numbered cursor.

### Maximum and Minimum

This finds the position of the maximum or minimum value found within the search range.

### Extreme

This finds the position of the data point that is the maximum distance in the y direction (either above or below) away from a reference level. There is an extra field in this mode for the Reference level.
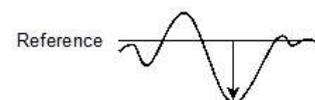
### Peak, Trough

The Minimum amplitude field sets the minimum acceptable size of the peak or trough; by how much the data must rise before a peak and fall after it (or fall before a trough and rise after it), to be accepted. In the diagram of a peak search, the first peak is not detected because the data did not rise by Minimum amplitude within the time range of the search. The Maximum width for peak field rejects peaks that are too broad (set it to 0 for no width restriction). The peak position is located by fitting a parabola through the highest point and the points on either side.
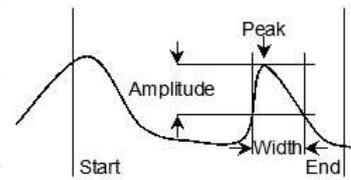
### Rising threshold, Falling threshold, Threshold

The data must cross Threshold from a level that is at least Noise rejection/hysteresis away from it and stay crossed for a time of at least Delay after crossing. For a Rising threshold mode, the data must increase through the threshold, for a Falling threshold mode it must fall through the threshold. In Threshold mode the crossing can be in either direction. The picture shows a rising threshold. The crossing point is found by linear interpolation of the data points on each side of the crossing.
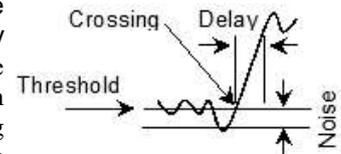
### Outside dual thresholds, Within dual thresholds

These two modes are similar to the rising and falling threshold modes except that they search for the data being outside or within two threshold levels. In addition to the First threshold, Delay after crossing and Noise rejection/hysteresis fields there is a Second threshold which sets the other threshold level. The data must cross a threshold from a level that is at least Noise rejection away from it.

### Steepest rising, Steepest falling, Extreme slope (+/-)

This finds the position of the maximum, minimum or the largest absolute value of the waveform slope. The Width for slope measurement field sets the length of data used to calculate the slope.
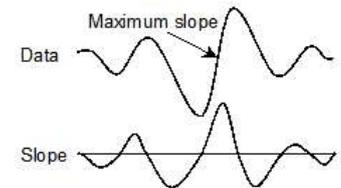
### Slope threshold, +ve slope threshold, -ve slope threshold

This finds the position at which the slope crosses a particular threshold level. The Width for slope measurement field sets the length of data used to calculate the slope. The Threshold units are y axis units per second.
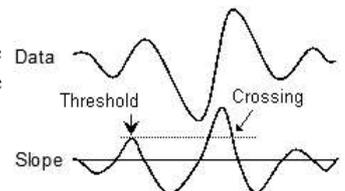
### Slope peak, Slope trough

These modes calculate the slope of the data in the search range, and then find the first peak or trough in the result that meets the Amplitude limit. The Width for slope measurement field sets the length of data used to evaluate the slope at each data point. The Amplitude field sets how much the slope must rise before a peak and fall after it (or fall before a trough and rise after it), to be accepted. The Amplitude units are y axis units per second.
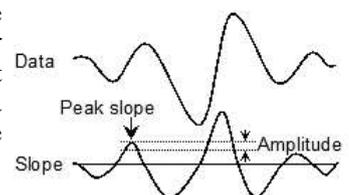
### Turning point

This mode finds the first point in the search range where the slope changes sign. Put another way, it finds a localised peak or trough. The Width for slope measurement field sets the data range to calculate the slope. The picture shows this method used to find the top of a sharp rise where Maximum mode would get the wrong place. To use this you would probably set a cursor on the peak slope and start the search from that point
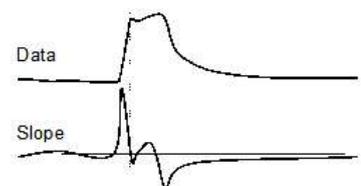
**Slope%**

This method finds the start and end of a fast up or down stroke in a waveform. The Width for slope measurement field sets the time width used to calculate the slope. The Slope% field sets the percentage of the slope measured at the start of the search that we want to find. To use this mode, set a cursor at the peak slope, then use it as the start point and search forwards or backwards for the required percentage. 15% usually works well.
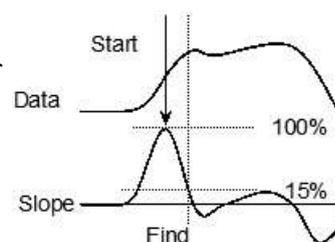
**Repolarisation %**

This mode finds the point at which the waveform returns to a set percentage of the distance to a baseline. The search start position defines the 0% repolarisation level. The 100% position and 100% measurement width fields set the position of the 100% level (this can lie outside the search range) and the width over which it is measured. Repolarisation percentage (drawn at 80% in the picture) sets a threshold relative to the measured 0% and 100% levels. The position is the first point in the search range to cross it.

**Data points**

Moves on by a specified number of data points. This is most useful for stepping through marker data.

**Expression**

The cursor position is obtained by evaluating the Expression field. This field will normally hold an expression based on cursor positions, for example "Cursor(1)+2.5".

# Search Right, Search left

If cursor 0 is active, these two commands cause the cursor to search for the next or previous position that satisfies the active mode. If the cursor active mode is Expression, the cursor goes the same way for both commands.

# Active horizontal cursors

Normally, Signal horizontal cursors are static; they stay where they are put. Using the active horizontal cursor dialog horizontal cursors in time and memory views can be made active; they will move to a level found by a measurement (normally taken from the channel on which the cursor is located). This search is repeated whenever the view data changes, cursor 0 is iterated, a relevant vertical cursor is moved, or the view switches to a different frame. This repositioning is normally carried out after all the vertical cursors whose position is used have been repositioned. Active horizontal cursors can be used as a simple way of marking levels within your data; they are also useful for extending the capabilities of analyses that generate measurements from data files.

**Vertical cursors**

Active horizontal cursors are normally treated as being subsidiary to vertical cursors and are normally automatically repositioned after all vertical cursors upon which they depend (for definition of the measurement time range, normally) have themselves been repositioned. It may be useful to override this positioning - for example so that all vertical cursors can make use of the horizontal cursor position - so mechanisms are provided to allow you to force a horizontal cursor to be repositioned before or after the vertical cursors.

**Valid and invalid horizontal cursors**

Active horizontal cursor positions are either *valid* or *invalid*; invalid cursors have an horizontal exclamation mark at the end of the label. The cursor position is invalid if the active search fails or is based upon an invalid expression. Expressions and measurements that use invalid cursor positions are also invalid, the measurement analyses all reject points that come from invalid measurements. Horizontal cursor positions are made valid by a successful active mode search or by any operation that moves them to a specific place such as dragging.

# Active horizontal mode

A variety of active horizontal cursor modes are available and can be used to show various levels in your data including the mean plus or minus various error values. This command opens a dialog from which you can select a horizontal cursor and set its active measurement mode. The contents of the dialog depends upon the Active mode field.
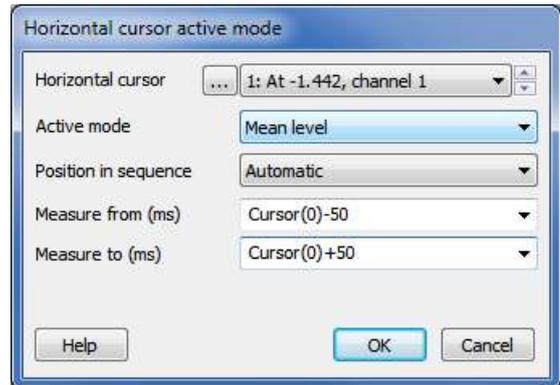
### Horizontal cursor

This control selects a horizontal cursor. The control displays the cursor number, label and the channel on which the cursor is placed as "n: At label, channel c" where n is the cursor number, label is the cursor label and c is the channel number. The '...' button to the left of the cursor selector opens the horizontal cursor label mode dialog for the current cursor to make it easy to change the label.

### Active mode

This field sets the active mode of the cursor, which in turn determines the other dialog fields that are displayed. The modes available are:

The active horizontal cursor modes that are available are:

| | |
|---|---|
| Static | When you add a new horizontal cursor, it starts off as Static, that is, not an active cursor. In this state, the cursor stays where you put it; it is not changed by a change in the data or the position of other cursors. |
| Value at point | This measures the channel value at a specified single position. If this is a vertical cursor position, the horizontal cursor will track the intersection of the vertical cursor with the data channel. |
| Mean level | This finds the mean channel data value within the specified time range. |
| Maximum | This finds the maximum channel data value within the specified time range. |
| Minimum | This finds the minimum channel data value within the specified time range. |
| Extreme | This finds the absolute maximum channel data value (the largest value ignoring the sign) within the specified time range. |
| Mean (SD*factor) | + This calculates the mean and standard deviation of the data around the mean in the time range. This can be used to detect significant changes from an otherwise flat (but noisy) baseline. |
| Expression | The horizontal cursor position is set by an expression string such as "HCursor(2)+1". If you use an expression such as At(c1) and do not supply a channel, the optional channel number is the channel of the horizontal cursor. |

Active horizontal cursors generally use a data channel for the measurement that sets their position. The channel on which the horizontal cursor is placed is used for all such measurements; if you move the horizontal cursor to a new channel then it is used thereafter for the measurements.

### Position in sequence

This field is present for all active modes. It controls when Signal repositions the horizontal cursor with respect to any active vertical cursors. The order is important, as active cursor measurements may depend on other active cursor positions. You can choose from:

| | |
|---|---|
| Automatic | After all the vertical cursors upon which it depends. Active vertical cursors are positioned in ascending numerical order, so if the highest vertical cursor used in the calculating the position of this horizontal cursor in n, vertical cursor n+1 can use the new position of this cursor to calculate its position. |
| First of all | Before the active vertical cursors move. This allows you to find a level with respect to the old vertical cursor positions. |
| Last of all | After all vertical cursors have been positioned. This means that any active vertical cursors that use horizontal cursor values will all use the previous positions, not the new ones. |

When following these rules, if multiple horizontal cursors would update at the same time, they update in rising numerical order.

Note that from Signal 6.05, each time a horizontal cursor moves, it saves the place it moved from, and this position is available with the script command `HCursorX()` and with the dialog expression commands `HCursorX()` and `HnX`.

### Measure from, Measure to, Position

These can be any View-based time expression, and will usually involve the positions of vertical cursors, particularly when cursor 0 is being used to iterate through features. If Measure to is less than Measure from then they are reversed so that they still specify a sensible X axis range.

### Scaling factor

This field is used with commands that calculate a level as a measure of the signal mean plus some multiple of a measure of the signal width. This will usually be a number in the range -5 to +5, but we do not place any restriction on the range of allowed values.

# Display Y values

This command opens a new window containing the values at the position of any cursors in the current data view. Columns for cursors that are absent, or for which there is no data, are blank. The cursor Y values display can also be opened using the `CursorOpen()` script function.

The values displayed depend upon the channel type and display mode. There is an entry in the table showing the time for each cursor, plus entries for each channel displayed. The displayed values are as follows:

| Cursors | Cursor 1 | Cursor 2 | Cursor 3 | Cursor 4 |
|---|---|---|---|---|
| s | 0.00844749 | 0.0144894 | 0.0205479 | 0.0276712 |
| 5 Keyboard | 0.018 | 0.018 | | |
| 4 ADC 3 | 0.0317383 | 0.0927734 | 0.0366211 | 0.012207 |
| 3 ADC 2 | 0.737305 | 0.0146484 | 0.078125 | 0.141602 |
| 2 ADC 1 | 1.17188 | 0.0830078 | 0.00976563 | 0 |
| 1 ADC 0 | 0.915527 | 0.0146484 | 0.0268555 | 0.0146484 |

*Cursors for example.cfs*

| Waveform | The y axis value of the nearest data point that is within one sample period of the cursor, or nothing if there is no data point close enough. Waveform measurements are not affected by the drawing mode. |
|---|---|
| Marker as Rate | The height of the rate bin that the cursor crosses. If the cursor lies on a bin boundary, the cursor is considered to lie in the bin to the right. |
| Marker | The time of the next marker at or to the right of the cursor. |

The X zero check box enables relative cursor time measurements. If checked, the cursor marked with the radio button is taken as the reference time, and the remaining cursor times are given relative to it. The reference cursor displays an absolute time, not 0.

The Y zero check box enables relative cursor value measurements. The radio buttons to the right of the check box select the reference cursor. The remaining channels display the difference between the values at the cursor and the values at the reference. The values for the reference cursor are not changed.

### Selecting and copying data

You can select areas of this window by clicking on them. Hold down the `Shift` key for extended selections. You can select entire rows and columns by clicking in the cursor and channel title fields. Use the `Ctrl` key to select non-contiguous rows and columns.

To copy selected rows and columns to the clipboard, by right-click the values window and use the Copy command in the popup menu. If you use the Log command the selected text is copied and pasted directly into the log window in one operation. You can also print the selected portions of the window by right-clicking and using the Print command in the popup menu, or use the Font command to change the window font.

# Cursor Regions

This command opens the Cursor regions dialog for the current data view. The dialog displays values for data regions between cursor pairs. At the top of the dialog is a row of cells showing the separation in time of each cursor pair, and a second row showing the inverse of the separation as Hz. One pair can be designated the Zero region by checking the box and selecting the column with a radio button. The value in this column is then subtracted from the values in the other columns. The pop-up menu in the bottom-left corner

| Cursor regions for example.cfs | | | | |
|---|---|---|---|---|
| Cursors | 0 - 1 | 1 - 2 | 2 - 3 | 3 - 4 |
| Time (s) | | 0.0048858447 | 0.0069406393 | 0.0094800043 |
| 1/Time (Hz) | | 204.6729 | 144.07895 | 105.48518 |
| 5 Keyboard | | 0 | 0 | 105.48518 |
| 4 ADC 3 | | 0.035351563 | 0.094281365 | 0.051472982 |
| 3 ADC 2 | | 0.0859375 | 0.17061121 | 0.03692627 |
| 2 ADC 1 | | 0.0625 | 0.28499828 | 0.032908122 |
| 1 ADC 0 | | 0.062402344 | 0.24615119 | 0.020904541 |
| ☐ Zero region | ⦿ | ○ | ○ | ○ |
| Mean ▼ | | | | |

indicates and controls how the values are calculated. The cursor regions values display can also be opened using the `CursorOpen()` script function.

# Cursor region measurements

The region set by a pair of cursors is the data starting at the first cursor up to, but not including, the data at the second cursor. For a waveform channel (including one drawn as histogram etc), the measurements are:

| | |
|---|---|
| Curve area | Each data point makes a contribution to the area of its amplitude above a line joining the endpoints multiplied by the x axis distance between the data points. The X axis distance is measured in seconds regardless of the time display mode (s, ms or us) selected in the display preferences giving an area measured in Y axis units*seconds. The picture makes this clearer. This measurement cannot be made on log-binned data. |
| Mean | The mean value of all the waveform points in the region. If there are no samples between the cursors the field is blank. |
| Slope | The slope of the least squares best fit line to waveform points in the region. |
| Area | The area between the data points and the y axis zero. The X axis distance is measured in seconds regardless of the time display mode (s, ms or us) selected in the preferences giving an area measured in Y axis units*seconds. Area is positive for sections above zero and negative for sections below zero. Use Modulus if you want areas below the y axis to be treated as positive. |
| Sum | The sum of all the waveform points in the region. If there are no samples between the cursors the field is blank. |
| Modulus | Each waveform point makes a contribution to the area of its absolute amplitude value multiplied by the time between samples on the channel. This is equivalent to rectifying the data, then measuring the area over zero. The X axis distance is measured in seconds regardless of the time display mode (s, ms or us) selected in the preferences giving an area measured in Y axis units*seconds. If a zero region is specified, the amount subtracted from the other regions is scaled by the relative width of the regions. |
| Maximum | The value shown is the maximum value found between the cursors. |

| | |
|---|---|
| Minimum | The value shown is the minimum value found between the cursors. |
| Peak-Peak | The value shown is the difference between the maximum and minimum values found between the cursors. |
| RMS amplitude | The value shown is the RMS level of the values found between the cursors. If there are no values between the cursors the field is blank. |
| SD | The value shown is the standard deviation from the mean of the values between the cursors. If there is no data, the field is blank. |
| Extreme | The value shown is the maximum absolute value between the cursors. If the maximum was +1, and the minimum was -1.5, this mode would display 1.5. |
| Peak | The value shown is the maximum found between the cursors measured relative to the baseline formed by joining the two points where the cursors cross the data. |
| Trough | The value shown is the minimum value found between the cursors measured relative to the baseline formed by joining the two points where the cursors cross the data. |
| Point Count | The number of waveform or marker channel data points, or idealised trace channel transitions. |
| SEM | The value shown is the standard error in the mean of the values between the cursors. If there is no data, the field is blank. |
| RMS error | The value shown is the RMS error relative to the mean of the values between the cursors. If there is no data, the field is blank. |
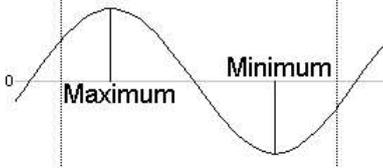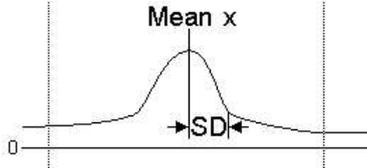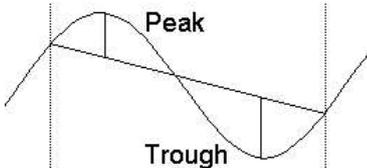
The measurements available for marker channels are Mean, Sum, Maximum, Minimum, Peak-Peak and Extreme. If you select other measurements the result is a blank field. The values calculated for the measurements are:

| | |
|---|---|
| Mean | The count of markers between the cursors divided by the time difference between the cursors. This could be thought of as the mean marker rate. |
| Sum | The total number of markers between the cursors. |
| Maximum | The maximum inter-marker interval, or the maximum histogram value for Rate display mode. |
| Minimum | The minimum inter-marker interval, or the minimum histogram value for Rate display mode. |
| Peak-Peak | The difference between the Maximum and Minimum values. |
| Extreme | The largest absolute value of Maximum and Minimum, this will always be the same as Maximum for marker channels. |

# Selecting and copying data

You can select areas of this window by clicking them. Hold down the `Shift` key for extended selections. You can select entire rows and columns by clicking in the cursor and channel title fields. Use the `Ctrl` key to select non-contiguous rows and columns.

To copy selected rows and columns to the clipboard, right-click in the values window and use the Copy command in the popup menu. If you use the Log command the selected text is copied and pasted directly into the log window in one operation. You can also print the selected portions of the window by right-clicking and using the Print command in the popup menu, or use the Font command to change the window font.

The above popup menu commands are also available via the following keyboard shortcuts:

`Ctrl+C`     Copy

| | |
|---|---|
| Ctrl+P | Print |
| Ctrl+F | Font |
| Ctrl+L | Log |

# Cursor context menus

In addition to the main menu commands it is also possible to access some commands by right-clicking on a cursor. This produces a popup menu which has a sub-menu specific to the cursor which has been clicked on. For vertical cursors the sub-menu has the following items:

| | |
|---|---|
| Active mode… | Puts up the active cursor mode dialog for the cursor. |
| Set position … | Puts up the position cursor dialog for the cursor. |
| Set Label … | Puts up the cursor label mode dialog for the cursor. |
| Copy Position=xxx uu | Copies the position (xxx with units uu) of the cursor to the clipboard. |
| Lock to cursor | Provides a list showing available (lower-numbered) vertical cursors and sets up this cursor's active mode so it is moved to maintain the current separation with the selected vertical cursor whenever the selected vertical cursor is moved. |
| Delete | Deletes the cursor (this is Hide for cursor 0). |

For horizontal cursors the sub-menu items are as follows:

| | |
|---|---|
| Active mode… | Puts up the active horizontal cursor mode dialog for the cursor. |
| Set position … | Puts up the position cursor dialog for the cursor. |
| Set Label … | Puts up the cursor label mode dialog for the cursor. |
| Copy Position=xxx uu | Copies the position (xxx with units uu) of the cursor to the clipboard. |
| Copy Position-HCursor(n)=Y | Subtract the position of horizontal cursor n from the cursor position and copy the result (Y with units uu) to the clipboard. Only available if there are multiple horizontal cursors on this channel. |
| Lock to cursor | Provides a list showing available vertical cursors and sets up this cursor's active mode so it shows the channel data level at the point where the selected vertical cursor crosses the channel data. |
| Delete | Deletes the selected cursor. |

# Sampling menu

The sampling menu controls the sampling configuration, and can start, stop and pause data capture.

## Sampling configuration

This command opens the Sampling Configuration dialog, which sets the data capture parameters used when you select the File menu New command (see under *File menu* for details). You can load and save the sampling configuration with the File menu Save Configuration and Load Configuration commands. You can also access this command from the Signal toolbar.

## Sample Bar

The Sample Bar is a dockable toolbar with up to 40 user-defined buttons, each button is linked to a Signal configuration file. When you click a button, the associated configuration file is loaded and a new data file is opened, ready for sampling and optionally, the sampling is automatically started. There is also a user-defined comment associated with each button which appears as a tool-tip when the mouse pointer lingers over a button. You can show and hide the Sample Bar and manage the Sample Bar contents using the Sample menu. You can also show and hide the Sample Bar by clicking the right mouse button on the Signal toolbar, on a blank bit of the Signal toolbar area or on the Signal background.

If you right-click on a button in the Sample Bar, you will get a small popup menu that allows you to change the Immediate start and Write to disk options for that button, to load the relevant sampling configuration and open the sampling configuration dialog so that it can be viewed or edited, to remove the button, or to open the Sample Bar List dialog.

## Sample Bar List

The Sample menu Sample Bar List… command opens the Sample Bar List dialog in which you control the Sample Bar contents.

The Add button opens a file dialog in which you can choose Signal configuration files (`*.sgcx` or `*.sgc`) to add to the bar. The first 8 characters of the file name form the initial sample bar button label and the comment is initially blank. Delete removes the currently selected item, Clear All deletes all items. You can select an item in the list and edit the label and comment, and re-order buttons in the bar by dragging items in the list. An Alt+key shortcut can be assigned to a button by putting the ampersand character (&) into the button label before the required character.

The Immediate start check box, if set, will cause sampling to begin immediately when the corresponding button in the sample bar is pressed. Similarly, the Write to disk check box can be used to force the initial state of write to disk check box in the sampling control panel to on or off.

The Sample Bar state is saved in the registry when Signal closes and is loaded when Signal opens. Each Windows logon account has a different registry configuration. If your system has three user accounts, each will have its own Sample Bar settings.
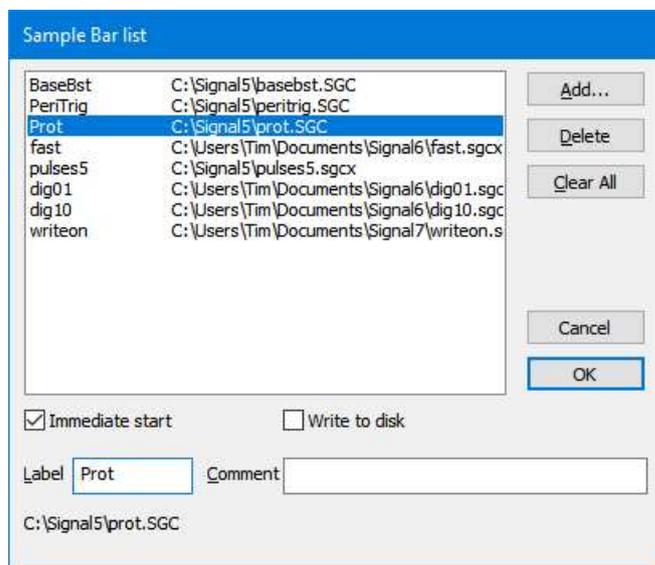
# Signal conditioner

Signal supports a number of programmable signal conditioners. These devices amplify and filter waveform signals and can provide other specialist functions. If a suitable conditioner is installed in your system, this command is available during sampling to open the conditioner dialog so that you can view and change the amplifier settings online.

# Show Sampling controls

This command, or its toolbar equivalent, hides and shows the sampling control panel; the menu item is checked when the control panel is visible. The main controls within the control panel are duplicated at the end of this menu as the Start sampling, Continue sampling, Triggered sweeps, Write to disk at sweep end, Pause at sweep end, Abort sampling and Restart sampling commands, (see *Sampling data* for full details of the control panel commands). In summary, the commands are:

### Start sampling
This command starts sampling. It is the same as the sampling control panel Start button.

### Stop sampling
When sampling has started, the Start sampling command changes to Stop sampling. This is equivalent to the sampling control panel Finish button. There is no warning before this command takes effect.

### Continue sampling
This command enables sampling of the next sweep when sampling is paused after collecting a sweep. It is equivalent to the sampling control panel Continue button.

### Triggered sweeps
This command toggles the state of the Sweep trigger check box in the sampling control panel. The menu item displays a check box when this option is selected.

### Write to disk at sweep end
This command toggles the state of the Write to disk at sweep end check box in the sampling control panel. The menu item displays a check box when this option is selected.

### Pause at sweep end
This command toggles the state of the Pause at sweep end check box in the sampling control panel. The menu item displays a check box when this option is selected.

### Abort sampling
This command aborts sampling and discards any sampled data. It is equivalent to the Abort button in the sampling control panel.

### Restart sampling
This command discards all data, returns sampling to the state it was in before sampling started and restarts sampling. It is the same as the sampling control panel Restart button.

# Show Pulse controls

This command, and its toolbar equivalent, hides and shows the pulse definition dialog that is available during sampling if pulse output is in use. This dialog can be used to change the output pulses while data acquisition is in progress, changes made will be saved in the current sampling configuration. The menu item is checked when the control panel is visible.(see *Pulse outputs during sampling* for details of the panel).

The sampling control panel, pulse controls and states control bar can all be shown and hidden by using the popup menu generated by clicking the right mouse button on an unused part of the Signal window (the blank parts of the toolbar area are suitable and always visible) during sampling.
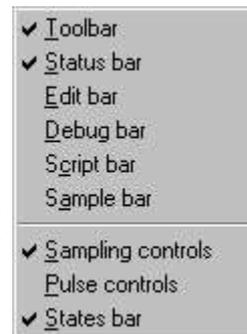
# Show States control bar

This command hides or shows the multiple states control bar that is available during sampling if multiple states are in use.

# Show Clamping controls

This command hides or shows the clamping controls that are available during sampling if clamping is in use.

# Dynamic Clamp models

This command can be used to display the main dynamic clamping models dialog that is available during sampling if dynamic clamping is in use.

# Show Sequencer controls

This command hides or shows the sequencer control panel that is available during sampling if the output sequencer is in use.

# Auxiliary states devices

If the current sampling configuration makes any use of any auxiliary states devices, this menu item will display a list of these devices; select a device to bring up the corresponding device settings dialog. You can also access auxiliary states device settings via buttons in the auxiliary states control bar, if that is displayed.

# Show auxiliary states bar

This command hides or shows the auxiliary states control bar that is available during sampling if any auxiliary states devices are in use.

# Sample now

This command is only available on the toolbar. It is equivalent to selecting New in the File menu then choosing Data Document. That is to say: it prepares Signal to start sampling with the current sampling configuration.

# Keyboard sampling control

Windows software is usually orientated towards control by means of the mouse and menus, but it is often convenient to use the keyboard instead. For quick control of sampling, using the keyboard can often be much faster. With this in mind, Signal includes keyboard shortcuts designed to handle the most common sampling control commands, see the information on controlling sampling for more information about what these commands do :

| Key | Operation |
| --- | --- |
| Ctrl+Alt+S | Start, Stop, or Finish sampling as appropriate. |
| Ctrl+Alt+C | Continue sampling (when paused) or More sampling (after it has stopped but not finished). |
| Ctrl+Shift+T | Toggle the sweep trigger. |
| Ctrl+Shift+P | Toggle pausing at the end of a sweep. |
| Ctrl+Alt+R | Restart the sampling that is in progress, discarding any collected data. |
| Ctrl+Alt+A | Abort sampling that is in progress, discarding any collected data. |
| Ctrl+Shift+W | Toggle writing sampled data to disk at the end of a sweep. |

Note that the Ctrl+Alt+S shortcut does a number of different things according to the current situation so do use it with a certain amount of care!

There are more keyboard shortcuts for data view and text view manipulation, and for data analysis.

# Script menu

The script menu gives you access to the scripting system. From it you can compile a script, run a loaded script, evaluate a script command for immediate execution and record your actions as a script.

## Compile Script

This command is enabled when the current view holds a script. It is equivalent to the Compile button in the script window. Signal checks the syntax of the script, and if it is correct, it generates a compiled version of the script, ready to run.

## Run Script

This command pops-up a list of all the scripts that have been loaded so that you can select a script to run. Signal compiles the selected script and if there are no errors, runs the script. If you run a script twice in succession, Signal only compiles it for the first run, saving the compilation time. If a script stops with a run time error, the script window is brought to the front and the offending line is highlighted.

You can also select the Load and run… option from which you can select a script to run. The script is hidden and run immediately (unless a syntax error is found in it).
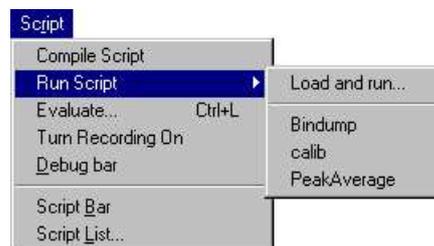
## Evaluate

This command and the `Ctrl+L` keyboard shortcut open the Evaluate dialog where you can type a line of script commands for immediate execution. The window remembers the last ten lines of script entered, which are shown in the drop-down list. You can cycle round the saved lines using the buttons. The Execute button executes the line entered, Eval(...) adjusts the line internally to include an `Eval()` on the last statement so that you can see the result. You can execute any script that can be typed in one line, which can include variable declarations.
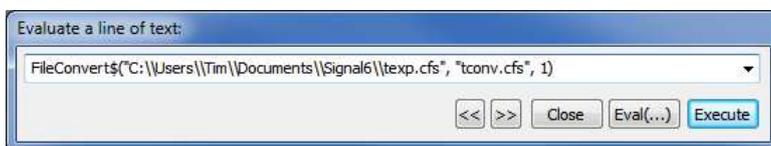
## Turn Recording On/Off

You can record your actions into a script that will produce equivalent actions. Use this command to turn recording on and off. When you turn recording on, Signal begins to save script commands corresponding to your actions. While script recording is in progress, the rightmost indicator in the Signal status bar will display the text REC as a reminder. When you turn recording off, a new script window opens that holds the saved script commands. If you then compile and run this script, the actions that you performed while recording was on will be repeated.

You can use this mechanism to record a sequence of actions that you wish to rerun at some later date, to find out what script commands correspond to a given menu command or user action or to record a sequence of actions that can be copied into another script or edited to produce a complete scripted 'application'.

## Debug Bar

You can show and hide the debug bar from this menu when the current view is a script. You can also show and hide the debug bar by clicking the right mouse button on any Signal toolbar or on the Signal background.

# Script Bar

The Script Bar is a dockable toolbar with up to 40 user-defined buttons, each button is linked to a Signal script file. When you click a button, the associated script is loaded and run. There is also a user-defined comment associated with each button which appears as a tool-tip when the mouse pointer lingers over a button. You can show and hide the Script Bar and manage the Script Bar contents from the Script menu. You can also show and hide the Script Bar by clicking the right mouse button on any Signal toolbar or on the Signal background.

If you right-click on a button in the Script Bar, you will get a small popup menu that allows you to to load the relevant script so that it can be viewed or edited, to remove the button, or to open the Script List dialog.

# Script Bar List

This command opens the Script List dialog from where you can control the contents of the Script Bar.

The Add button opens a file dialog in which you can choose one or more Signal script files (*.sgs) to add to the bar. If the first line of a script starts with a single quote followed by a dollar sign, the rest of the line is interpreted as a label and a comment, otherwise the first 8 characters of the file name form the label and the comment is blank. The label is separated from the comment by a vertical bar. The label can be up to 8 characters long and the comment up to 80 characters. A typical first line might be:
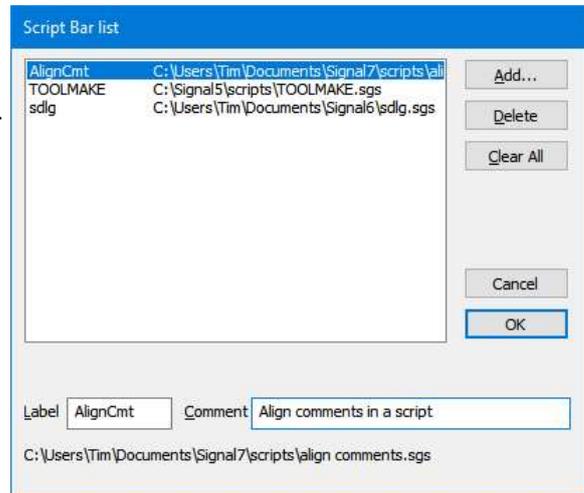
```
'$ToolMake|Write a toolbar driven script skeleton
```

You can select an item in the list and edit the label and comment, this does not change the contents of the script file. An Alt+key shortcut can be assigned to a button by putting the ampersand character (&) into the button label before the required character.

You can re-order buttons in the bar by dragging items in the list. The Delete button removes the selected item. Clear All removes all items from the list.

The list of files in the Script Bar is saved in the registry when Signal closes and is loaded when Signal opens. Each different logon to Windows has a different configuration in the registry, so if your system has three different users each has their own Script Bar settings. Alternatively, you can have different experimental configurations by logging on as a different user name.

# Window menu

The window menu provides facilities for managing the windows that Signal displays.

## Duplicate window

This command creates a duplicate file or memory view window with all the attributes (list of displayed channels, display modes, colours, cursors and size) of the original window. Once you have created the new window, it is independent of the original. Duplicating a window allows you to have different views of the same data showing different frames or with different scales and different channels visible.

You can close all windows associated with a data document using the File menu Close All command (see *File menu*). This will remember the position and state of all windows associated with the document.

## Hide

This command makes a window invisible. This is often used with script windows and sometimes is used to hide data windows during sampling when only the memory views with analysis results are required.

## Show

This command lists all hidden windows. Select a hidden window to make it visible.

## Window Title

The data document maintains both a file name and a window title. If the file has been saved, the window title is usually set to the name of the file (without the file path). When you open a new window, the title is set to the type of the window followed by a sequence number, for example Data2 or Grid23. The window title is mainly cosmetic, but it is used to suggest a file name when you save an unsaved document.

You can change the window title with the script `WindowTitle$()` command, and also from the Window Title dialog. The dialog is available from the Window menu and also from the context menu that appears when you right-click on the window title.



### Title
This field shows the original title when the dialog opens and you can edit it to a new title.

### OK
This button is enabled if you make a change to the text. Click the button to apply the change and close the dialog. Note that changes made to a title are not undoable (but see Reset).

### Cancel
Close the dialog without making any change to the title.

### Reset
This sets the title back to the default window title. If the associated document has been saved, the default title is the file name. Otherwise, the default title is the type of the window (Data, Text, Grid...) plus a sequence number.

# Tile Horizontally

You can arrange all the visible Signal windows so that they are arranged in a horizontally tiled pattern by using this command. Horizontal tiling arranges the windows so that they tend to be short and wide, the exact arrangement depends upon the number of windows.

# Tile Vertically

You can arrange all the visible Signal windows so that they are arranged in a vertically tiled pattern by using this command. Vertical tiling arranges the windows so that they tend to be tall and thin, again the exact arrangement depends upon the number of windows.

# Cascade

All windows are set to a standard size and are overlaid with their title bars visible.

# Arrange Icons

You can use this command to tidy up the windows that you have iconised in Signal.

# Close All

This command closes all windows in the Signal application. You are asked if you want to save the contents of any text windows that have changed. The positions of data document windows are all saved.

# Windows

This dialog lists all the document-related windows that are open and lets you apply common window operations to one or more of the windows. You can sort the list based on the window title, type, view number (as seen by the script language) and window state by clicking the title bar at the top of the list.

The operations available depend on the type and numbers of the selected windows. For example, the Activate and Maximize buttons are only enabled when a single view of a suitable type is selected.
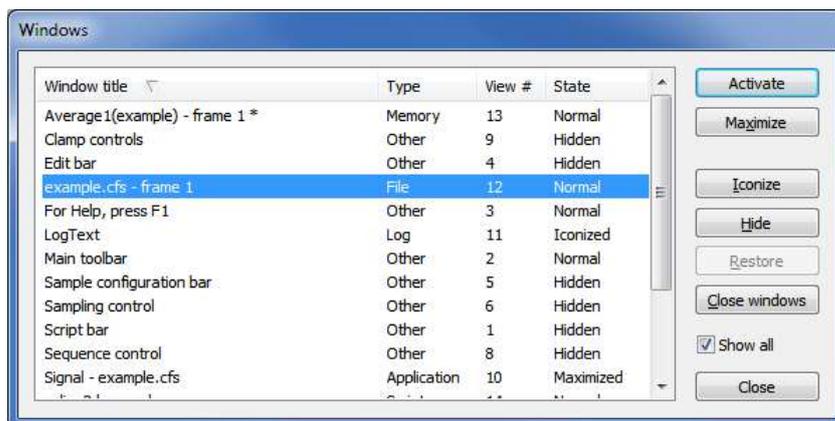


### Show all

Normally, the only windows displayed are for windows that own documents (script, output sequencer, and text views, the log view, file views, memory views and XY views). However, if you check the Show all box, the window lists all windows that have a view handle that the script language can manipulate.

# Help menu

The help menu is used to provide access to the online help system (this information) and to get program information.

## Help Index

This command provides the main help index.

## Using help

This command provides online help information on how to use the online help system.

Signal supports context sensitive help and also duplicates the contents of this manual in the help file. You can activate context sensitive help with the F1 key, or by pressing the Help button, from most dialogs to get a description of the dialog and its fields. You can use the Help menu Index command to get a dialog holding the help contents, an index to help keywords and a word search system to find topics that are not covered by the contents and index.

From a script view or the script evaluate dialog you can obtain help by placing the cursor on any keyword in the script and pressing F1. To get help on a script function, type the function name followed by a left hand bracket, for example FileOpen(, then make sure that the cursor lies to the left of the bracket and in the function name and press F1. Pressing the help button (the button with a question-mark) at the top right of the script window provides overall script language help.

The help is implemented using the standard Windows help system, with contents, indexes, hypertext links, keyword searches, help history, bookmarks and annotations. If you are unsure about using Windows help, use the Help menu Using help command to get detailed instructions.

## Tip of the day

This command provides a dialog with a small piece of information in a "Did you know?" form. Further details can then be requested. This dialog can also be set to appear when Signal is first run or, if you hate this sort of thing, prevented from running.

## View web site

If you have an Internet browser installed in your system, this command will launch it and attempt to connect to the CED web site (www.ced.co.uk). The site contains down-loadable scripts, updates to Signal and information about CED products.

## Getting started

This command runs a demonstration script that gives a quick tour of Signal basics. The option is disabled if the script is not present or if you are sampling data.

# About Signal

The About Signal command is found in the Help menu. It opens an information dialog that contains the serial number of your licensed copy of Signal, plus your name and organisation. The dialog also display information about the 1401 device driver and your 1401. If you contact us for assistance we will need the information in this dialog. The Copy button writes the Signal version and serial number, your user name and organisation (that was entered when you installed Signal), the device driver and 1401 information, your working set size and information about various key directories to the clipboard. You can then paste this into an email or other text document.

## Signal version information

The top line of text identifies the version of Signal that you are running, the target platform (x86 for a 32-bit build or x64 for a 64-bit build), Unicode or Ascii and the date when we built the code. This is followed by our copyright statement. The third line is present if Signal can contact the CED website. If there is a more recent version available, there will be a message of the form: Version 7.nn is available for download. If your version is up to date, we display 'Signal is up to date'. If we could not contact the site, or if the information we read makes no sense, the line is blank. If there is an update available you can click the Web Site button to go to our general update page to collect it.

## 1401 device driver

If there is a 1401 device driver installed, the driver revision is displayed. If the driver is older than Signal expects, you will be warned. Signal displays the type of 1401 and the monitor version if a 1401 is connected and powered up. If you have installed Signal correctly, you should not have any problems as Signal comes with the latest 1401 drivers available at the time we generated the release. If you really need to update the drivers independently of Signal you use the 1401 Windows installer link on our web site.

## 1401 firmware revision

If the 1401 monitor or 1401 firmware is not the most recent at the time this version of Signal was released, an asterisk follows the version. If it is so old that it compromises data sampling, two asterisks follow the version.

The Power1401s and Micro1401 mk II and -3 have firmware in flash memory. Flash updates and instructions for applying them are available as downloads from the CED web site; you can update the flash firmware without opening the 1401 case. There are downloads for all current 1401 types on our web site.

To fix this, you should follow the link for your 1401, go to the download and follow the detailed instructions to update your 1401. The update is done by the Try1401 program that is installed at the same time as Signal, so you already have all the tools you need to complete the task.

### 1401: Monitor or firmware too old or interface problem

This message appears if we detect that the 1401 monitor ROM is too old to run Signal safely or if there is a problem with 1401 communications such that the 1401 is detected but communications are garbled.

## Working set size

Information about the Signal application Working Set Size is shown at the bottom of the About box. The two numbers describe the minimum and maximum physical memory that the operating system allows Signal to use. If you suffer from error -544 when you sample data, these numbers are important, there is more information about error -544 in the Common Questions section.

## User data

This is the path to the folder for user-specific data; the `SignalX` folder inside the users documents folder (commonly called `My Documents`). It is recommended that you use this location for your data and files if possible.

### Shared public data

This is the path to the folder for shared user data; the `SignalXShared` folder inside the documents folder for all users.

### User application data

This is the path to the folder where Signal stores information relevant to the user; normally something like: C:\Users\User\AppData\Local\CED\SignalX\

### Application data

This is the path to the folder where Signal can store private information, normally C:\ProgramData\CED\Signal7\. It is not recommended that you make use of this directory as it may be protected, if it is then text to this effect will be appended to the path.

# Other sources of help

If you are having trouble using Signal, please do the following before contacting the CED Software Help Desk:

1.  Read about the topic in the manual. Use the Index to search for keywords related to the topic.

2.  Try the help system for more information. Use the Search facility to find related topics.

3.  Try using the CED user forums. This section of the CED web site is a discussion board system where users of CED software (and CED personnel) can post questions, expand on existing queries or answer queries. You can search the forums to see if somebody has already come up with an answer to your problem, or post a new question. You have to join the forum in order to post questions (joining is a painless process) but can search the forums as a non-member guest.

4.  If none of the above helps email (softhelp@ced.co.uk) or call the CED Software Help Desk (telephone numbers and addresses are to be found at the front of the Signal manual and in the Contacting CED help page). Please include a description of the problem, the Signal serial number and program version number and a description of the circumstances leading to the problem. It would also help us to know the type of computer you use, how much memory it has and which version of Windows you are running.

# Script language

The script language integrated into Signal can be used to customise Signal, provide specialised analyses or automate various tasks.

## Script introduction

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

1. Find the peak after the stimulus pulse.
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. Next frame. 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 Signal 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] Run button to check and run your first script. If you have made any typing errors, Signal 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? Signal recognises names followed by round brackets as a request to perform an operation (called a *function* in computer-speak). Signal has around 400 built-in functions, and you can add more with the script language. You pass the function extra information inside the round brackets. The additional information passed to a function is called the function *arguments*.

Signal 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 Signal. 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*.

Signal 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 under *Script language syntax*.

Signal 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 (an integer number) that identifies it. A view is a window in Signal that the script language can manipulate. Views fall into one of the following categories:

- The Signal application window

- Time, result, XY or text-based views

- Dialog based views and toolbars

Whenever you use a built-in function that creates a new view, the function returns a view handle. View handles run from 1 to 32767; each time you open a new view you get a different handle until all 32767 have been used, at which point we start from the lowest unused handle again. The only exception to this is when you set a sampling configuration to automatically sample a sequence of data files; in this case, all files in the sequence share the same view number (as each file is closed before the next opens). The view handle is used most commonly with the `View()` and `FrontView()` functions to specify the current view and the view that should be on top of all windows.

All views have a parent and an owner. These are often the same thing. For example, a File view is a child of the application (so the application window is the parent) and it is owned by the application. However, a cursor regions window is owned by a File or Memory view, but is a child of the Windows Desktop (so it can be moved anywhere on the Desktop, independently of the position of the Signal application). You can find the view handle of the owner of a view with the `ViewLink()` command.

There is also a concept of the input focus. The view with the input focus gets the first shot at deciding what to do with keyboard characters and mouse messages. Use `FocusHandle()` to get the view with the input focus.

### The current view
There is always a current view. Even if you close all windows the Log view, used for text output by the `PrintLog()` command, remains. The current view is an important concept as many of the built-in script commands apply to the current view. You get the view handle of the current view with the `View()` command with no arguments; use the `View(handle%)` command to set the current view. Do not confuse the current view (which is the view that script commands will use) with the Active window (the one with the highlighted title bar, usually the last window where the user clicked the mouse). The current view and the Active view will often be the same, for example after the `FrontView()` script command has set it, but can be different, for example after the `View()` command has been used.

Whenever a script creates a new view, it becomes the current view. Views are usually created invisibly so that they can be configured before appearing (which prevents unsightly screen flashes). You can use `WindowVisible(1)` to display a new window.

When debugging a script, you can see the current view handle by opening the Global variables window.

### The front view
The `FrontView()` script command with no arguments returns the File, Memory, XY or text-based view that is the Active view of the application. The Active view has a highlighted title bar and if the application has the input focus, either it or one of its children or a view it owns has the input focus. You can also use the `FrontView(handle%)` command to bring any script-controllable view to the front (or as near to the front as is possible) and make it the current view.

### The Signal application window
This view always exists and the handle can be obtained with the `App(0)` command. You can use this view handle to position, show and hide the application window. You cannot close this window with `FileClose()`, use the `FileQuit()` command to do this. The application window is a top-level window and is a child of the Desktop.

### File, memory, XY and text-based views
These are all views with an associated document that can be saved as a file on disk. They are special in the sense that one of them is always the Active view of the application. Use `FrontView()` to return the Active view.

Text-based views are: simple text windows, the Log view, output sequencer views and script views. The running script view is hidden from most script commands, however you can obtain its view handle with `App(3)` so you

can show and hide it. Closing the Log view hides it as it always exists. You can get the Log view handle with `LogHandle()`.

You can see a list of File, Memory, XY and text-based view handles by using the Window menu Windows... command.

### Dialog-based views and toolbars

These cover all the script-controllable views that do not fall into one of the other categories: Cursor windows (see `CursorOpen()`), the sampling control panel, Sequencer control panel and the Sample Status bar (see `SampleHandle()`). The `App()` command gives you access to: the System toolbar, the status bar, the Edit toolbar, the Script bar, the Sample bar, the sample control panel, the Sequencer control panel, the states bar and the clamping control bar.

# Writing scripts by example

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

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

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

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

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

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

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

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

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

# Using recorded actions

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

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

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

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

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

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

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

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

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

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

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

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

# Derived views

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

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

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

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

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

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

# Notation conventions

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

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

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

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

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

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

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

Three dots (`...`) stand for a list of further, similar items. It is also used when a function can accept an array with any number of dimensions:

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

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

# Sources of script information

This continues with information about the script window and debugging, followed by a reference for the script language syntax and then documentation for the built-in script functions, first with functions grouped by topic, and then a full alphabetical list.

When you are in the script editor, there are several features designed to help you write script code. These include pop-up help tips, right-clicks to take you to the help for a specific function, pop-up help when adding arguments to functions, automatic formatting and auto-complete of typed names.

There are example and utility script provided with Signal. These are copied to the scripts folder within the user data folder created by the Signal installer.

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

There is also a manual that has been used for our Signal training courses, held at CED and around the world. This Training Day 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. You can get a PDF of this manual from our web site on the Downloads page (follow the links Software manuals, Signal V6, Training).

# Script window and debugging

# Script editor

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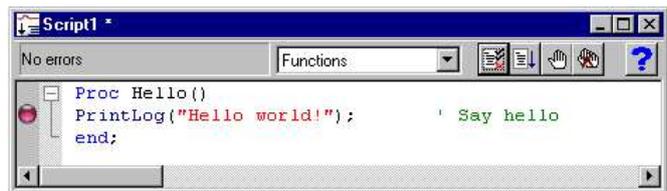


If a script is read from a read only medium or is write protected you cannot make changes to it in the Signal script editor.

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

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

**Function**



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

 **Compile**

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

 **Run**

If the script has not been compiled it is compiled first. If no errors are found, Signal runs the compiled version, starting from the beginning. Signal skips over `proc ... end;` and `func ... end;` statements, so the initial

code can come before, between or after any user-defined procedures and functions. This button is disabled once the script has started to run.

### Set break point

This button sets a break point on the line containing the text caret, or clears a break if one is already set. A break point stops a running script when it reaches the start of the line containing the break point. You can also set and clear break points by clicking the mouse pointer in the margin or by right clicking the line and using the Toggle break command. A break point is indicated by a red marker in the gutter (between the optional line number and folding margins). You can set and clear break points by clicking in the gutter, or using the context menu (right-click) in a script view.

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

Once a script has been compiled, all breakpoints in it and in any included files are remembered. You can close and open the files and the break points will still be visible. However, break points are reassessed when you click on the run button to start a script. This means that if you want to have a break in an included file you must have the file open and the break point set when you hit run. Once the script has started to run you can close any or all of the script files and break points will still work and automatically open the source file when they are hit.

### Clear all break points

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

### Help

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

### Go to user-defined Proc or Func

You can navigate to the start of a user-define `Proc` or `Func` by right-clicking on the name and selecting Go to name from the context menu. This works, even if the named item is defined in an included file (as long as the included file can be located).

## Syntax colouring

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

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

## Editing features for scripts

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

# Debug overview

Despite all our 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 (and on one documented occasion did) cause hardware problems.

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

- Step one statement at a time
- Step into or over procedures and functions
- Step out of a procedure or function
- Step to a particular line
- Enter the debugger on a script error to view variable values
- View local and global variables
- Watch selected local and global variables
- Edit variable values
- See how you reached a particular function or procedure
- Set and clear break points

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

# Preparing to debug

Unlike most languages, the Signal script language does not need any special preparation for debugging except that you must set a break point or include the `Debug();` command at the point in your script at which you want to enter the debugger.

Alternatively, 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()`, `DlgShow()` 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 from entering the debugger 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 an infinite loop.

You can also choose to enter the debugger on a script error by checking the Enter debug on script error box in the Script page of the Preferences dialog (accessed from the Edit menu). Depending upon the error, this may let you check the values of variables to help you find how to fix the problem.

The debug toolbar contains buttons for functions that are useful for debugging a script. The debug toolbar is normally disabled, when your script enters the debugger the toolbar opens if it was not already visible, the toolbar is enabled, and any debug  windows that were open when the last debug session ended are restored. The picture shows the toolbar as a floating window, but you can dock it to any side of the Signal window by dragging it over the window edge and releasing. The buttons available on the debug toolbar are:

Stop running the script (`Ctrl+Alt+Shift+F5`). There is no check that you really meant to do this; we assume that if you know enough to open the debugger, you know what you are doing! You can use the `Debug()` command to disable the debugger. This also cancels any chained script set with `ScriptRun()`.

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

If the current statement contains a call to a user-defined `Proc` or a `Func`, step into it, otherwise just step (`Ctrl+F5`). This does not work with the `Toolbar()` command which is not user-defined, but which can cause user-defined routines to be called. To step into a user-defined `Func` that is linked to a `Toolbar()` command, set a break point in the `Func`.

Step over (execute) this statement and move on to the next statement (`Shift+F5`). If you have more than one statement on a line you will have to click this button once for each statement, not once per line.

Step out of a procedure or function (`Alt+F5`). This does not work if you are in a function run from the `Toolbar()` command as there is nowhere to return to. In this case, the button behaves as if you had pressed the run button.

Execute the script up to the start of the line with the text caret (`Ctrl+Alt+F2`). This is slightly quicker than setting a break point, running to it, then clearing it (which is how this is implemented).

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

Show the local variables for the current user-defined `func` or `proc` (F7). If there is no current routine, the window is empty. 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 (`Ctrl+F7`). You can edit a global variable by double clicking on it. The very first entry in this window lists the current view by handle, type and window name.

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

Open the Watch window. This displays globals and locals that were selected in the global or local variables windows. Local variables are displayed with values when the user-defined `Proc or Func` in which they are defined is active.

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; if the Status bar is visible, a longer description can be seen there.

# Enter debug on error

There is an option in the Script page of the Preferences dialog that allows you to enter the debugger if an error occurs. After an error you can inspect the values of local and global variables and the contents of the call stack, but you are not allowed to continue running the script.

# Inspecting variables

If the watch, 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, a line is highlighted if the variable value was changed by the last step. If you double click on one a new window opens in which you can edit the value of the variable.

If you double click on an array, a new window opens that lists the values of the elements of the array. You choose the element by editing the index fields, one for each dimension.

## Menu commands

The variable windows contain a menu with the following commands:

| | | |
|---|---|---|
| File | Close | Closes the debug window |
| Edit | Copy | Copy selected text to the clipboard |
| | Log | Copy selected text to the Log view |
| | Select All | Select all windows text |
| | Find... | Open the Find text window |
| | Find Next | Repeat the last find operation |
| | Find Previous | Repeat last find operation searching backwards |
| | Toggle Bookmark | Set/clear bookmark in Global variable window |
| | Next Bookmark | Jump to the next bookmark in the Global variable window |
| | Previous Bookmark | Jump to previous bookmark in the Global variable window |
| | Clear All Bookmarks | Remove all bookmarks from the Global variable window |
| View | Font | Open the Font dialog to change the window settings |
| Watch | Add to the Watch window | Globals and Locals only: add selected items to the end of the Watch window |
| | Delete from the Watch window | Watch window only: remove selected item from the window |
| | Delete all watched variables | Watch window only: remove all items from the window |
| | Delete 'Not found' variables | Watch window only: remove all items that are not in the current script |
| | Sort variables into alphabetic order | Watch window only: sort items into alphabetic order |

Most commands have keyboard shortcuts listed in the menu. The Watch menu items are also available on a right-click context menu, where appropriate.

**Watch window**

You can add variables to the watch window by right-clicking on them in the locals or globals window and choose the option to copy the selected variables to the watch window. In the watch window, right-click to see available options to control the watched variables. The watch window remembers the watched variables between debugging sessions. If a variable does not exist in the current script, it is still remembered, but is marked as not existing.

# 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. Recursive functions can be very useful, but must be used with care.

A tricky fault to diagnose is a script that has mutually recursive functions. If there is no way out, the script gets deeper and deeper into the call stack until it runs out of memory. The call stack window can help to detect such problems. The example to the right demonstrates the use of a recursive function to generate a factorial (but see `LnGamma()` and `BinomialC()` for large factorials). From Signal version 6.03 onwards, excessive call stack use is trapped and will stop a runaway recursive script (unless it runs out of memory before hitting the stack limit).
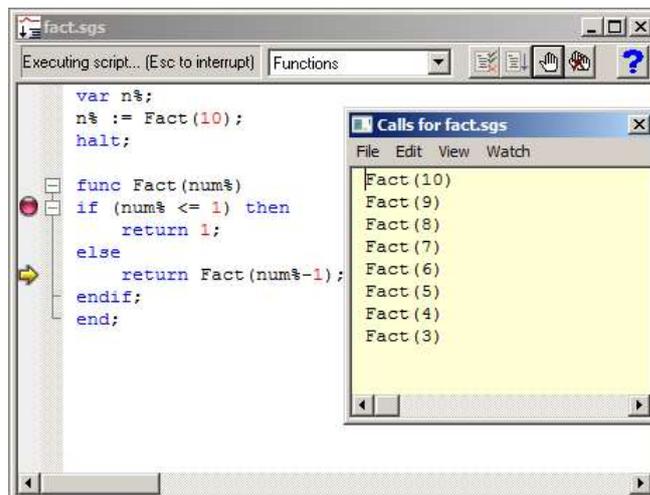
# Script language syntax

A Signal **script** is composed of lines of text. Each line can be up to 240 characters long (this is an arbitrary limit - most lines will be much shorter). Scripts are usually written in the script editor, but can be written with any editor and imported. The script uses 8-bit UTF-8 characters.

A script consists of program **statements**. Statements that generate looping or branching constructs include other statements within them. A statement is terminated by a semicolon or by a keyword that is part of an enclosing statement. Statements are not terminated by an end of line character, so a statement can be spread over multiple lines. Two semicolons in a row generate an empty statement, which generates no code.

Within a statement, **white space** of any length is treated as a single space character, and white space between script tokens is ignored unless is it is required to separate two tokens. White space consists of the space and tab characters and end of line characters.

A **token** is an indivisible entity such as a script keyword, the name of a constant, variable, procedure or function or a numeric or string constant. White space can be added anywhere without changing the meaning of a script except in the middle of a token.

# Keywords and names

All keywords, user-defined functions and variable names in the script language start with one of the letters a to z or _ (underscore) followed by the characters a to z, 0 to 9 and underscore. 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. We reserve starting a symbol name with underscore for CED use. There is nothing to stop you using it, but we will use a leading underscore for future constant names, so your use might collide with ours. Underscore was added as an acceptable character at Signal versions 6.03 and 5.12.

Variables and user-defined functions use the characters % and $ at the end of the name to indicate integer and string type. The % or $ is considered part of the name.

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   breakbxor case
const conti diag  do    docas else
      nue               e
end   endca endiffor   func  halt
      se
if    mod   next  not   or    proc
repea resiz returstep  then  to
t     e     n
trans until var   view  wend  while
xor
```

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

# Data types

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

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

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

# Real data type

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

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

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

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

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

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

### Technical stuff

Real numbers in Signal are stored in the double format defined in IEEE 754-1985, which is a standard format used in most computers. Each number uses 64 bits, of which 11 hold an exponent, there is one sign bit, and 52 bits hold the magnitude of the data, making 64 bits. The format manages to squeeze one extra bit out of the magnitude by stating that all normalised numbers start with a 1 bit that need not be supplied. You can find more information and further links here.

Unlike integer numeric expressions, which have a constant precision unless you overflow their value range, the precision of a calculation involving real numbers may be problematic. Multiplication and division do not cause any degradation in precision other than a rounding effect in the least significant bit of the result. However, addition and subtraction can cause significant loss of precision. For example, the expression: `1e15 + (1.0/3.0) - 1e15`

does not have the value 0.33333333... as you might hope, but 0.375. The expression `1e16 + (1.0/3.0) -`
`1e16` has the value 0. If you are forming the difference of numbers of similar sizes, the precision of the result is
limited by the precision of the larger number. You can sometimes reduce the loss of numerical accuracy by
rearranging the expressions to be evaluated.

When comparing real numbers that are the results of calculations, it is a good idea to avoid testing for exact
equality. For example, the following will never end:

```
var a := 0.0;
repeat
    a += 1/999.0;
until a = 1.0;
```

The problem is that 1/999.0 cannot be represented exactly in the real number format, the closest that can be stored
is 0.00100100100100100100; this is slightly less than the exact result. It is easy to see that after adding this up 999
times, the result is 0.99999999999999999900, which is not 1. Changing the last line to:

```
until a >= 1.0;
```

guarantees that the loop will terminate.

If you want to store integral numbers in a real data type, you can store integral values in the range -
9007199254740992 to 9007199254740992 (this is 2 to the power 53). Beware that the functions `Ceil()`,
`Floor()`, `Trunc()` and `Round()` have no effect on numbers greater in magnitude than 9007199254740992 as
they have no fractional part.

# Integer data type

The integer type is identified by a `%` at the end of the variable name and stores 64-bit signed integer (whole)
numbers in the range -9223372036854775808 to 9223372036854775807. 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; it is an error to assign numbers beyond the integer range. Real numbers
have any fractional part removed before assignment (-1.99 is -1 as an integer). If you assign an integer to a real
number you can lose accuracy as real numbers have typically 53 bits of precision. Put another way, if you assign an
integer larger than `9007199254740992` (`0x20000000000000`, $9 \times 10^{15}$) to a real number you will lose
precision. When differencing very large integers, use integer arithmetic for maximum accuracy.

The following are examples of integers:

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

**Integer limits and older versions of Signal**

| Feature | 64-bit | 32-bit |
|---|---|---|
| Decimal range | -9223372036854775808 to 9223372036854775807 | -2147483648 to 2147483647 |
| Hexadecimal range | 0x8000000000000000 to 0x7fffffffffffffff | 0x80000000 to 0x7fffffff |
| As a power of 10 | $-9.223 \times 10^{18}$ to $9.223 \times 10^{18}$ | $-2.147 \times 10^{9}$ to $2.147 \times 10^{9}$ |
| Size in bytes | 8 | 4 |

Before Signal version 7.00 the script language used 32-bit integers, if you wish your script to run in version 6 you
must limit your use of integers to the 32-bit range.

# 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
\r  The `Carriage Return` character (ASCII code 13)

### Unicode

If you are using a Unicode build of Signal (look in the About Signal dialog if you are not sure), there are additional escape sequences to allow you to embed Unicode characters into the string literal:

\uxxx  xxxx stands for up to 4 hexadecimal digits that specify a Unicode character in the range 0x0000 to 0xffff. However, characters in the range 0xd800 to 0xdfff are not allowed (these are reserved for UTF-16 lead and trail codes). You can use this to input characters that are not easily available from your keyboard. For example, the Greek letter $\pi$ is \u03c0. In reference material, Unicode characters are often written as U+03CO, and the hexadecimal digits after the U+ are the ones you need. You can omit leading 0s as long as the following character is not hexadecimal; you could write $\pi$ as \u3c0.

\Uxxx  Although most common characters in most languages can be represented by a 4 digit hexadecimal code, the
xxx    full Unicode code range if from 0x000000 to 0x10ffff. To represent these characters you can use \U followed by up to 6 hexadecimal characters. You can use fewer if the following character is not hexadecimal. For example the Unicode character for a music notation treble clef is code U+1D11E, so we could write this as \U01d11e or \U1d11e.

In addition to using an escape sequence, you can type in all the characters that your keyboard supports and you can also use Input Method Editors.

There is no guarantee that all Unicode codes exist in the fonts on your computer. Characters that are missing from the font are usually rendered in a fall-back font (so may not match the style of the selected font), and failing that, as an empty square box. For example the treble clef character mentioned above is not present on my computer in a fall-back font.

### Unicode surrogate characters

Signal strings in Unicode mode use the same underlying encoding as Windows does to store characters. Most characters fit in 16-bits of data, and the text strings are organised as arrays of 16-bit numbers. Characters with code points greater than 0xffff are represented by two consecutive characters known as surrogates. The string manipulation routines that use indices, such as `Left$()`, `Mid$()`, `Right$()`, `DelStr$()` and `InStr()`, use an index to these 16-bit elements. The routines that extract strings will never return a string that starts or ends half-way through a surrogate pair. They achieve this by moving on by one 16-bit element to the next. It would be possible for us to hide this from all users by making all indices into strings indices to characters, not to the underlying 16-bit codes. However, there would be a performance penalty for doing this and as surrogate pairs are usually very rarely encountered, it does not feel worth doing. We may review this in the future.

### See also:
Real data type, Integer data type, Conversion between data types

# 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. You can use the `Round()`, `Trunc()`, `Floor()` and `Ceil()` functions to convert floating point values to integral floating point values. The `Asc()`, `Chr$()`, `Str$()`, `Print$()` , `ReadStr()`, and `Val()` functions convert between strings and numbers.

# Variable declarations

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

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

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

The dimensions of global arrays must be constant expressions (but see `resize`). The dimensions of local arrays can be set by variables or calculated expressions. Simple variables (not arrays) can be initialised when they are declared. Uninitialised numeric variables are 0; uninitialised strings are empty.

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

### Compatibility with previous versions of Signal

Before Signal version 5, the initialising expression could not include variables or function calls. If you want to write a script that is compatible with version 4, the previous example must be written as:

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

# Constant declarations

Constants are created by the `const` keyword. A constant can be of any of the three basic data types, must be initialised in the declaration with a constant expression and (before Signal version 6.03) cannot be an array. A constant expression is composed of previously defined numeric constants, numbers and the operators add, subtract, multiply and divide (+-*/) or is a string constant. The syntax and use of constants is the same as for variables, except that you cannot assign to them or pass them to a function or procedure as a reference parameter.

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

### Constant arrays

From Signal 6.03 onwards you can declare a constant array. This is only useful if the array is initialised:

```
const day$[] := {"Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"};
const prime%[10] := {2, 3, 5, 7, 11, 13, 17, 19, 23, 29};
```

The syntax for initialising a const array is the same as for a var. If you do not initialise the values, a const array has zero values or permanently empty strings. You cannot resize a const array. A const array may only be passed to a func or proc or built-in function that declares the argument as const.

### Built-in constants

From Signal 6.03 onwards the following named constants are pre-defined. They were also added to Version 5.12 onwards. You will get a compilation error if you attempt to redefine them to anything else:

| Name | Type | Use |
| --- | --- | --- |
| _Version | Integer | The program version number times 100 as a constant. So for version 6.03 this has the value 603. This is the same value as returned by `App(-1)`, but it is known at compile time. This means that if a new built-in function MagicFunc() was added at version 6.05 you could write: |

```
if _Version >= 605 then MagicFunc(23) else WorkAround(23) endif;
```

Where WorkAround() stands for code to emulate what the new function does. This works in version 6.03 and 6.04 (where the new function does not exist) because the compiler first builds an internal representation of the program in which `MagicFunc()` is assumed to be a user-defined function that it has not yet found. Then the compiler notices that the `then` condition will never be run, so it converts the entire line to `WorkAround(23)`, and then it checks for undefined functions. This is no longer a problem as `MagicFunc()` is no longer used.

| | | |
|---|---|---|
| _VerMinor | Integer | For version 6.03 this is 0, for 6.03a it would be 1, for 6.03b it would be 2, and so on. |
| _pi | Real | The value of the mathematical constant $\pi$ (3.141592653589...) as accurately as can be stored in a real number. |
| _e | Real | The value of the mathematical constant e (2.718281828459...) as accurately as can be stored in a real number. |

# Arrays of data

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

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

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

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

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

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

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

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

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

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

as this will generate a "Name multiply defined or redefined" error. In a Proc or Func, you can declare an array inside a loop, and change the size of the dimensions each time around the loop. However, version 4.06 provided the resize statement, and we urge you to declare arrays outside loops and use resize to do any required size changes.

```
Proc BadStyle()
var i%;
for i% := 1 to 100 do
var arr[i%]; 'this has always been allowed...
...
next;
end;

Proc BetterStyle()
var arr[0], i%; 'declare the array once
for i% := 1 to 100 do
resize arr[i%]; 'resize it
...
next;
end;
```

We may make resizing an array using var illegal in the future. Note that before Signal version 4.06, resizing with var preserved the original data when the last dimension was changed, but changes to any other dimension would not preserve the data.

### Vector subsets

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

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

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



### Matrix subsets

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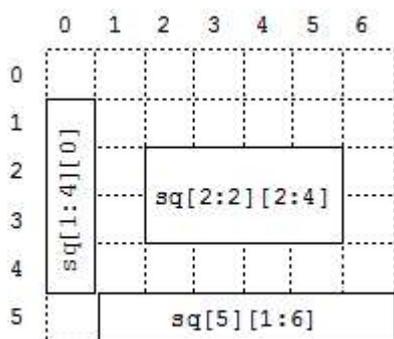


| | |
|---|---|
| `sq[a][b:c]` | a vector of length c |
| `sq[a][]` | a vector of length 7 |
| `sq[a:b][c]` | a vector of length b |
| `sq[][c]` | a vector of length 6 |
| `sq[a:b][c:d]` | a matrix of size `[b][d]` |
| `sq` or `sq[][]` | a matrix of size `[6][7]` |

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

### N-Dimensional array subsets

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

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

### Transpose of an array

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

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

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

### Diagonal of a matrix or array

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

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

### Memory usage of arrays

Each element of a real array uses 8 bytes of memory. Each element of an integer array uses 4 bytes of memory and each element of a string array uses 16 bytes, plus the memory to hold the text in the string (one byte per character) plus a terminating 0. Each array is held in contiguous memory (except that each string element holds a pointer to the string text that is not in this contiguous memory). An array can require a lot of memory. For example, a real array with three dimensions of 1000, 100 and 10 has 1 million elements, each using 8 bytes. If you use many, large arrays, it is possible to run out of system memory. When this will happen is difficult to predict as it depend on many system settings, on how much physical memory your system has and how much virtual memory space the system will allocate to you. Once you exceed the limits of available physical RAM, random array access can become very slow as data is swapped between memory and disk. Adding more physical memory to your system can help if this is a problem. Signal scripts will stop with an error if you request more memory than the system can allocate to you.

### Changes to arrays at version 5.09

Global arrays can be defined with a variable size. The following would not have compiled in previous versions:

var n% := 10, x[n%];

You can now pass arrays of zero size to user-defined functions; previously this caused a fatal error. You can also pass zero-sized arrays to some built-in functions, for example ArrConst().

## Initialise an array

At the point where an array is declared it can also be initialised. The following examples show how you initialise integer arrays, but the same syntax holds for real arrays and for string arrays (but replacing the numbers with literal strings). All the data used to initialise the arrays must be known at compile time. The syntax for `var` and `const` arrays is the same, so they are used interchangeably in the examples. Any data in the array that is not explicitly initialised is set to 0 (or an empty string).

If you need to set the values in an array (to other than 0 or empty strings) and ArrConst() and its friends will not do the job, it is usually more convenient (less typing) and faster at run time to initialise them than to write out assignment statements.

### Vectors

These are the simplest to initialise. The syntax for a vector is:

```
var name[optional size] := {list of items separated by commas};
```

If the size of the vector is specified, it must be a constant expression (known at compile time) and the list of items must be no longer that the size. If the size is not specified, the vector length is set by the initialisation list size.

```
var data%[] := {0, 1, 2, 3, 4};
const c%[10] := {,,,,1,1};
```

In the first example, no array size is specified, so the size is defined by the initialisation data. In the second example, the size is specified, but values are provided only for the fourth and fifth elements. Any unspecified elements are set to 0 (or to an empty string in the case of an array of strings), so the initialiser is equivalent to:

```
{0,0,0,0,1,1,0,0,0,0}
```

If the size of an array dimension is declared, it is an error to give more initialisation data than can fit:

```
var bad%[4] := {1, 2, 3, 4, 5};
const JustAsBad%[4] := {,,,,};
```

### Matrix

These are a little more complicated: The syntax for a matrix is:

var name[rows][cols] := {{list of row 0 data}, {list of row 1 data}, ... ,{list of cols-1 data}};

Where ... means continue in a like manner. Both rows and cols are optional, but must be constant expressions if they are supplied. If either is missing, the size is set by the largest row or column of the initialisation data. Within the list of row data, you may omit values by having two consecutive commas, and the missing value will be set to 0. You can also set an entire row to 0 (or empty strings) by omitting it:

var name[rows][cols] := { , {list of row 1 data}, ... ,{list of cols-1 data}};

In this case, row 0 is initialised to zeros. The next example sets up two arrays to hold the same data.

```
var d2d%[2][3] := {{1, 2}, ,{5,6}};
var same%[][] := {{1,2}, {0,0}, {5,6}};
```

### 3 or more dimensions

The syntax continues in a logical sequence. So to initialise a 3D array:

var d3d%[2][3][4] := {{{1, 2},   {3, 4},   {5, 6}},

        {{7, 8},   {9, 10},  {11, 12}},

        {{13, 14}, {15, 16}, {17, 18}},

        {{19, 20}, {21, 22}, {23, 24}}};

You can think of this as a 4 x 3 matrix of vectors of length 2. To handle a 4D one:

```
const d4d%[1][2][3][4] := {{{{ 1},{ 2}}, {{ 3},{ 4}}, {{ 5},{ 6}}},
                           {{{ 7},{ 8}}, {{ 9},{10}}, {{11},{12}}},
                           {{{13},{14}}, {{15},{16}}, {{17},{18}}},
                           {{{19},{20}}, {{21},{22}}, {{23},{24}}}};
```

You can think of this as a 4 x 3 matrix of 2 x 1 matrices. We leave the 5D array as an exercise. You can zero fill (or fill with empty strings) any region by omitting the contents of any pair of matching curly braces, you can even omit the curly braces (except the outermost pair). If we wanted to zero the elements with values 1-6 and 13-18 in the d3d% array we could have written:

```
var d3d%[2][3][4] := { ,
                  {{7, 8},   {9, 10},  {11, 12}},
                  {},
                  {{19, 20}, {21, 22}, {23, 24}}};
```

In the first case we have omitted everything, and in the second case we have omitted the contents of the curly braces.

# Resize array

You cannot change the number of dimensions of an array, but you can change the size of the dimensions. This is done with the resize statement (added at Signal version 4.06), which has a syntax that is very similar to var:

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

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

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

A pair of empty square brackets means that you want to preserve the current size of the corresponding dimension. The resize statement preserves data in the array (unless you make one or more dimensions smaller, when data is omitted). When you make dimensions larger, new numeric array elements are set to 0; new string elements are set to an empty string (""). If the new array dimensions are the same as the existing ones, nothing is done and no time is wasted.

In most cases, you will only want to change one dimension to cope with adding more items to an array. It is more efficient to increase the last dimension as in this case it is often possible to extend (or reduce) the memory allocated to the array without physically moving it in memory. If you change any other dimension than the last, the resize statement allocates a new array of the required size, copies data into it, replaces the original array with the new one and releases the memory used by the original array.

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

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

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

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

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

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

**Efficiency**

If you are adding items to an array, it is very inefficient to increase the array size for each item added. Apart from being very slow, this will cause a pattern of memory allocation that is about the worst possible for the performance of the system. The standard solution in this case is to start with a reasonable size, one that will be big enough for most situations, then when you need more, to allocate a sensible extra portion of space. If you have no idea how big the target is, the best algorithm (best in terms of reducing the number of reallocations and memory fragmentation) is to double the size each time you run out. However, this is also the most wasteful of memory. Increasing by a fixed amount or a fixed proportion of the existing size may work. Do NOT increase by one each time unless the array is very small and is never going to get very big.

# Data views as arrays

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

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

**Closing referenced result view**

If you pass a result view as an array as a reference argument to a user-defined func or proc, then close the result view, you will get a fatal script error:

```
A result view was closed that was aliased by a script array or variable
```

If the result view was closed by the user (for instance during `Interact()`) rather than by a script you will get the error when the script resumes. This error is fatal as there is no way for the script to continue if it holds references to something that no longer exists. The following minimal code demonstrates both ways of generating this error.

```
var rv% := SetMemory(1, 100, 0.01, 0, 0, 0, 0, "test", "x");
Err0(view(rv%, 1).[]);          'Pass as an array
Err1(view(rv%, 1).[10]);        'Pass an element as a reference

proc Err0(data[])
FileClose();                    'Error if data[] is the memory view array
end;

proc Err1(&v)                   'v is passed by reference
FileClose();                    'Error if v is part of the memory view
end;
```

# Statement types

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

- A variable or constant declaration

- An assignment statement of the form:

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

The `+=`, `-=`, `*=` and `/=` assignments were added at version 3.02. `+=` can also be used with strings (`a$+=b$` is the same as `a$:=a$+b$`, but is more efficient).

- A flow of control statement, described here.

- 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
'This is a whole line comment that includes Japanese (日本語) text
```

# Expressions and operators

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

Numeric operators

String operators

The ternary operator

Examples of expressions

Mathematical constants

# Numeric operators

| | Operators | Names |
|---|---|---|
| Highest | `trans()`, `` ` ``, `[]`, `()` | Matrix transpose (2 forms), array subscript, round brackets |
| | `-, not, ~`<br>`!` | Unary minus, logical not, bitwise not (version 7.00 onwards)<br>You can use `!` for logical not from version 7.00 |
| | `*, /, mod`<br>`%` | Multiply, divide and modulus (remainder)<br>You can use `%` for modulus from version 7.00 |
| | `+, -` | Add and subtract |
| | `<<, >>, >>>` | Shift left, shift right, bit shift right (all new at version 7.00) |
| | `<, <=, >, >=` | Less, less or equal, greater, greater or equal |
| | `=, <>` | Equal and not equal |
| | `and, band`<br>`&& &` | Logical and bitwise and<br>You can use `&&` and `&` from version 7.00 |
| | `or, xor, bor, bxor`<br>`\|\| \| ^` | Logical `or`, exclusive `or` and bitwise versions<br>You can use `\|\|`, `\|` and `^` from version 7.00 |
| Lowest | `?:` | Ternary operator |

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

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

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

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

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

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

### Backwards compatibility
If you write scripts that must run in versions of Signal before 7.00 do not use any of the following symbols as operators:

`~ ! % << >> >>> && & || | ^`

### Unary operators
Unary operators take a singe argument and transform it to generate a result. The type of the result depends on the argument and the operator.

| Op | Example | Use |
|---|---|---|
| `` ` `` | `M[][]` | Transpose a matrix or convert a vector to a matrix. The operator comes after the argument. |

| | | |
|---|---|---|
| `tra`<br>`ns` | `trans(`<br>`M)` | This is an alternative to the ` operator. |
| `[]` | `a[2]` | Array subscript operator. |
| `()` | `a*(b+c`<br>`)` | Round brackets, used to force order of evaluation and for argument lists. |
| `-` | `-a` | Negate the following numeric item. The result is of the same numeric type (real or integer). Negating the full-scale negative integer value (0x8000000000000000) generates an overflow error. |
| `not`<br>`!` | `not a`<br>`!a` | Logical not. Has the value 1 if `a` is 0, otherwise 0. `a` can be real or integer. |
| `~` | `~1` | Bitwise not. The argument must be an integer and the result is the bitwise inversion of the argument. The result of ~1 is -2, in hexadecimal `~0x0000000000000001` is `0xfffffffffffffffe`. |

## Binary operators

Binary operators take two arguments, one before the operator and one after. They can be sub-divided into arithmetic, bitwise, comparison and logical operators.

### *Arithmetic*

The result type of an arithmetic binary operator is integer if both arguments are integers, otherwise it is real.

| Op | Example | Use |
|---|---|---|
| `*` | `a*b` | Forms the product of the argument. Multiplying two integers will cause an overflow error is the result exceeds 64-bit integer range. |
| `/` | `a/b` | Divides the first argument by the second. Division by 0 is an error. It is a common error to forget that the result of dividing an integer by an integer is an integer, so `1/3` is 0. `1/3.0` (or `1.0/3` or `1.0/3.0`) is `0.333333...` |
| `mod`<br>`%` | `a mod b`<br>`a%b` | The remainder after dividing the first argument by the second, equivalent to `a-(a/b)*b` |
| `+` | `a+b` | Form the sum of the two argument. There will be an overflow error if both arguments are integers and the result exceeds integer range. |
| `-` | `a-b` | Form the difference of the two arguments. There will be an overflow error if both arguments are integers and the result exceeds integer range. |
| `<<` | `a%<<b%` | Bitwise left shift the first integer argument by the second integer argument. The shift must be positive and bits shifted in at the bottom are 0. For example, `1<<3` is 8. In binary terms: 0001 becomes 1000. |
| `>>` | `a%>>b%` | Signed bitwise right shift of the first integer argument by the second integer argument. The shift must be positive and the bits shifted in at the top match the sign of the number. For example `3>>2` is 0 and `-3>>2` is -1. `8>>3` is 1. |
| `>>>` | `a%>>>b%` | Unsigned bitwise right shift of the first integer argument by the second integer argument. The shift must be positive and the bits shifted in at the top are always 0. If the first argument is positive, the result is the same as for `>>`. However, `-1>>>1` is 9223372036854775807. In hexadecimal `0xffffffffffffffff>>>1` is `0x7fffffffffffffff`. |

### *Bitwise*

The bitwise operators all take integer arguments and generate an integer result.

| Op | Example | Use |
|---|---|---|
| `<<` | `a%<<b%` | Bitwise left shift the first argument by the second argument. The shift (second argument) must be positive and bits shifted in at the bottom are 0. For example, `1<<3` is 8. In binary terms: 0001 becomes 1000. |
| `>>` | `a%>>b%` | Signed bitwise right shift of the first argument by the second argument. The shift must be positive and the bits shifted in at the top match the sign of the number. For example `3>>2` is 0 and `-3>>2` is -1. `8>>3` is 1. |

| | | |
|---|---|---|
| `>>>` | `a%>>>b%` | Unsigned bitwise right shift of the first argument by the second argument. The shift must be positive and the bits shifted in at the top are always 0. If the first argument is positive, the result is the same as for `>>`. However, `-1>>1` is 9223372036854775807. In hexadecimal `0xffffffffffffffff >>> 1` is `0x7fffffffffffffff`. |
| `band` `&` | `a% band b%` `a% & b%` | Bitwise AND. If bit `n` of both the arguments is 1, the output bit `n` is 1, else it is 0. For example, `5 band 1` is 1 (in binary `0101 & 0001` is `0001`). The bitwise and is often used to clear out bits from a number. For example to clear bits 0-3 from `a%`: `a% := a% & ~15;` |
| `bor` `|` | `a% bor b%` `a% | b%` | Bitwise OR. If bit `n` of either argument is 1, the output bit `n` is 1, else it is 0. For example, `4 bor 1` is 5 (in binary `0100 & 0001` is `0101`). Bitwise OR is often used to set bits after they have been cleared by a bitwise AND: `a% := a% | 7;` |
| `bxor` `^` | `a% bxor b%` `a% ^ b%` | Bitwise exclusive OR. If bit `n` of both arguments differ, the output bit `n` is 1, else it is 0. For example, `5 bor 1` is 4 (in binary `0101 & 0001` is `0100`). Bitwise exclusive OR is often used to toggle bits: `a% := a% ^ 1;` changes the state of bit 0. |

## Comparison

Comparison operators take two arguments that can be real or integer values. If both arguments are integers, the integer version of the operator is used, otherwise the real value version. The operators return the integer value 1 if the result of the comparison is true, and the integer value 0 if it is false. There are six comparison operators:

| Op | Example | Use |
|---|---|---|
| `<` | `a < b` | Less than. |
| `<=` | `a <= b` | Less than or equal. |
| `>=` | `a >= b` | Greater than or equal. |
| `>` | `a > b` | Greater than. |
| `=` | `a = b` | Equal. Be very cautious when testing real values for equality, especially if you are testing the result of an expression. |
| `<>` | `a <> b` | Not equal. |

## Logical

The logical binary operators take two numeric arguments and combine them to generate an integer result of 1 if the logical operation is true and 0 if the logical operation is false. The two arguments are treated as false if they are equal to zero and true if they are not equal to zero. There are three logical operators:

| Op | Example | Use |
|---|---|---|
| `and` `&` | `a and b` `a && b` | Logical AND. If both the arguments are non-zero, the result is 1, otherwise it is 0. |
| `or` `||` | `a or b` `a || b` | OR operation. If either argument is non-zero, the result is 1, otherwise it is 0. |
| `xor` | `a xor b` | Exclusive OR operation. This is equivalent to `(a <> 0) <> (b<>0)`. The result is 1 (true) if one of the arguments is true (not zero) and the other is false (zero). |

# String operators

In string expressions, the following operators are allowed, listed in order of precedence:

| | Operators | Names |
|---|---|---|
| Highest | `*` | Multiply. `"-" * 10` is equivalent to `"----------"`. This was added at version 7.00 |
| | `+` | Concatenate |
| | `<, <=, >, >=` | Less, less or equal, greater, greater or equal |
| | `=, <>` | Equal and not equal |
| Lowest | `?:` | Ternary operator |

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

*Multiplication*

As applied to a string, the multiplication operator expects a string on the left and an integer on the right. If both the string and the integer are known at compile time, the string is generated as a literal, but this will generate an error if the result is more than 1000 characters. Otherwise, the string is processed at run time, but it is an error if the resulting string exceeds the maximum allowed string length.

*Concatenation*

This operator expects a string on the left and the right and generates a string that is the first string followed by the second. If both strings are known at compile time, the operation is done then to generate a literal. Otherwise, the strings are processed at run time; it is an error if the resulting string exceeds the maximum allowed string length.

*Comparisons*

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. The character order for comparisons (lowest to highest) of the standard ASCII characters is:

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

If two strings match up to the length of the shorter string, the longer string is considered greater. Script strings are stored as Unicode characters in UTF_16LE format. This uses 16-bit values to store the characters with the majority of common characters fitting in one 16-bit value. However, the Unicode standard allows for over 1,000,000 characters, so some characters require two values to encode them. The encoding system is carefully arranged so that lexical sorting systems will still generate sensible results by comparing at the 16-bit level. Our comparisons also work at the 16-bit level and give sensible results for the comparison operators even for uncommon characters.

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$;`.

**Backwards compatibility**

If you write scripts that must run in versions of Signal before 7.00, you must not use the string `*` operator.

# The ternary operator

The ternary operator `?:` was added to the script language at Signal version 3 and has the following format:

```
numeric expression ? expression1 : expression2
```

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

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

# Examples of expressions

The following (meaningless) code gives examples of expressions.

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

The following expressions are valid in version 7.00 onwards, but not in previous versions of Signal:

```
sue% := jim && not sam; 'Result is 0 (jim = -2.0 and sam = 1.0)
pip% := 1 && 0 || 2>1;  'Result is 1
jim% := 1 << 4;           'Result is 16
sue% := 9 & 8;            'Result is 8 (9=1001 in binary, 8=1000)
sue% := 9 ^ 8;            'Result is 1
alf$ := "alf"*2;          'Result is "alfalf"
```

# Mathematical constants

Before Signal version 6.03 and 5.12 we didn't provide maths constants e and $\pi$ as built-in constants _e and _pi. If you need to write a script that will also work with older version of Signal, $e$ is `Exp(1.0)` and $\pi$ (pi) is `4.0*ATan(1.0)`. Alternatively, here are their values to as much accuracy as you can store in a double :

$\pi$     3.141592653589793
*e*     2.718281828459045

# Flow of control statements

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

# if...endif

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

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

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

```
if num% mod 2 then
   num%:=num%+2;                 ' the semicolons before...
else                            '...else and endif are optional.
   num%:=num%+1;
endif;

'The following are equivalent
if num% mod 2 then num%:=num%+2 else num%:=num%+1 endif;
num% += num% mod 2 ? 2 : 1;'An alternative using ?:
```

# docase...endcase

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

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

This example displays the type of a file handle:

```
'case.sgs
var i%,m$;
i% := ViewKind();       'get type of the current view
docase
   case i% = 0 then m$ := "file";
   case i% = 1 then m$ := "text";
   case i% = 2 then m$ := "output sequence";
   case i% = 3 then m$ := "script";
   case i% = 4 then m$ := "memory";
   case i% = 8 then m$ := "external text";
   case i% = 9 then m$ := "external binary";
   case i% = 10 then m$ := "Signal";
   case i% = 12 then m$ := "XY view";
   else m$ := "something else...";
endcase;
message("Current window is of type "+m$);
```

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

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

# repeat...until

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

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

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

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

You can break out of a repeat loop early using the `break` statement. You can skip to the until statement with the `continue` statement.

```
var t;
repeat
   t := NextTime(3, t);          'get a time
   if (t<0) then break endif;    'stop if time not found
   if (t<10) then continue endif; ' skip if not enough time yet
   PrintLog("At time %5.2f 10 second count = %d\n", t, Count(3, t-10, t));
until t >= MaxTime(3);
```

# 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;
```

You can break out of a while loop early using the `break` statement. You can skip back to the start of the loop with the `continue` statement.

```
var i% := 0;
while i% <= 999 do
    ...
    if StopEarly() then
       break
    else
       if SkipRestOfLoop() then continue endif;
    endif;
    ...
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;
```

This is a more complicated example which prints prime numbers:

```
const MaxN% := 1000;        'the maximum number to search
var prime%[MaxN%],i%,t%;    'the "Sieve" and loop variables
for i%:=1 to MaxN%-1 do     'ignore 0 as not included in search
   prime%[i%] := 1          'mark all numbers as possible primes
next;
for t%:=2 to MaxN%-1 do     'start search at 2 as 1 is special case
   if prime%[t%] then       'if number remains, must be prime
      for i% := 2*t% to MaxN%-1 step t% do
         prime%[i%]:=0      'remove all multiples of prime number
      next;
   endif;
next;
t% := 0;                    'use this to put output in columns
for i% := 1 to MaxN%-1 do   'search the "sieve" for next prime
   if prime%[i%] then       'have we found one?
      if t% mod 10 = 0 then PrintLog("\n") endif;
      t% := t% + 1;         'count the number
      PrintLog("%5d", i%);  'output the prime
   endif;
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. You can break out of a `for` loop early using the `break` statement. You can skip to the next statement with the `continue` statement.

# continue & break

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

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

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

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

The break and continue statements are useful because they give you a tidy way of interfering with the execution of a loop and a way to jump out of deeply nested `if` or `docase` statements inside a loop in a similar way that the `return` statement allows you to jump out of the middle of a `Proc` or `Func`.

The `break` and `continue` statements are new in Signal version 4.06; if you use them your script will not compile in older versions of Signal.

# 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 unless `ScriptRun()` has been used to set the next script to run. The `Halt` statement can occur at the outer level of a script or in a function or procedure. For example:

```
Proc DoSomething(arg)
if arg < 0 then
    Message("Bad argument to DoSomething(%g)", arg);
    Halt;
endif;
    ...
end;
```

# 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.

This example function finds the greatest common divisor of two integers:

```
var n1%, n2%;
n1% := Input("First number", 10000, 1, 2000000000);
n2% := Input("Second number", 30000, 1, 2000000000);
Message("Greatest common divisor of %d and %d is %d", n1%, n2%, gcd%(n1%, n2%));
halt;
func gcd%(m%, n%) 'm% and n% both > 0!
var d%;
while n% do
   d% := Max(m%, n%) mod Min(m%, n%);
   m% := Min(m%, n%);  'get smaller value
   n% := d%
wend;
return m%;
end;
```

### 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.

You can use a `return` statement in a function or procedure to break out of a loop or an if or `case` statement. If you are using a return to break out of a loop early and do not need to be compatible with version 3, you could consider using the `break` statement.

### Call tips

You can define pop-up call tips that appear when you hover the mouse over a user-defined `Proc` or `Func` name. These are enabled in the script file settings dialog.

# Argument lists

The argument list is a list of variable names separated by commas.

### Pass by Value or by Reference

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, so this is equivalent to making a copy. Changes made in the function to a variable passed by value do not affect the calling context. Passing by value is the standard method for everything except arrays.

Reference    Arguments passed by reference are the same variables (by a different name) as the variables passed from the calling context. For reasons of efficiency, arrays are always passed by reference. Non-array variables have & before the argument name declaration to pass by reference. 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. For convenience, we do allow you to pass a literal of matching type to a **const** reference argument.

By default, 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.

```
proc BothWays(a, &b)
a := 6;      'a is local to BothWays
b := 6;      'This changes
end;


var x,y;
BothWays(x, y);     'x is unchanged, y is set to 6
```

Arrays and sub-arrays are always passed by reference. Array arguments have empty square brackets in the function declaration, for example `one[]` for a vector and `two[][]` for a matrix. The number of dimensions of the passed array must match the declaration. The array passed in sets the size of each dimension. You can find the size of an array with the `Len()` function. You can declare an array argument as **const** and the compiler will generate an

error if you attempt to modify the array or pass it as a non-**const** argument to another function. An individual array element is a treated as a simple variable.

You can even change the size of an array that was passed as an argument as long as you meet the rules for resizing, and the new size will be seen by the caller of your function.

If you use the `trans()` or `diag()` operators to pass the transpose or diagonal of an array to a function, or pass a sub-array, the array is still passed by reference and changes made in the function to array elements will change the corresponding elements in the original data. You cannot resize such an array. If the original array was declared **const**, the result of these operators is also **const** and you can only pass such objects into functions with a **const** argument..

### const arguments

You can place the keyword **const** before any argument (including a reference argument). If you do this, it is an error to modify the variable within the **func** or **proc**. Declaring an argument **const** is usually done to make it clear to the user that then function will not modify the argument.

### Default arguments

Simple (non array) arguments that are passed by value can also specify a default value. This is done by following the variable name by `:= value`, for example:

```
func fred(a := 1, b$ := "", c% := 0)
...
return a;
end;
```

When arguments are defined with default value you can omit them entirely when they are trailing arguments, or skip over them if they are not. For example, all the following are legitimate:

```
'call as          equivalent to
fred();           'fred(1,"",0)
fred(2);          'fred(2,"",0)
fred(,"x");       'fred(1,"x",0)
fred(,,6);        'fred(1,"",6)
```

Default arguments are just a convenience. The code is generated exactly as if you had supplied the arguments. A common use is to extend an established command to cover a special case. By adding an addition argument with a default value you can leave all your existing code unchanged and then use the extra argument only when required.

### References and deprecated behavior

Before Signal version 6.02 we allowed you to pass a real variable where a reference to an integer variable was expected. This was to remain compatible with ancient versions of the compiler. If your script uses this, you can make it compile by checking the *Allow float script var to be passed as an integer reference matching 6.02* option in the Compatibility page of the Preferences dialog.

# return

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

# Examples of user-defined functions

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

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

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

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

```
func sumArr(data[])         'sum all elements of a vector
var sum:=0.0;               'initialise the total
var i%;                     'index
for i%:=0 to Len(data[])-1 do
   sum := sum + data[i%];   'of course, ArrSum() is much faster!
   next;
return sum;
end;
Func SumArr2(data[][])      'Sum of all matrix elements
var rows%,cols%,i%,sum;     'sizes, index and sum, all set to 0
rows% := Len(data[0][])     'get sizes of dimensions...
cols% := Len(data[][0]);    '...so we can see which is bigger
if rows%>cols% then         'choose more efficient method
 for i%:=0 to cols%-1 do sum += ArrSum(data[i%][]) next;
else
 for i%:=0 to rows%-1 do sum += ArrSum(data[][i%]) next;
endif;
return sum;
end;
```

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

# Scope of user-defined functions

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

### Mutually recursive functions

Because functions are globally visible, you may define two functions that call each other (mutually recursive). If you do this it is your responsibility to ensure that there is a way out of the mutual recursion. If you do not, the script will compile, but will crash at run time as it will (eventually) exhaust system memory.

# 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. This means you can also define default arguments. 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. If you need to pass such a function you must wrap it in a user-defined function:

```
func USin(x) return Sin(x) end; 'USin(x) has an unambiguous signature
```

# Channel specifiers

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

`cSpc$`  A string channel specifier 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. Virtual channels can be specified using the `vn` channel numbers shown by Signal, for example `"1, 5, v1,v3-v5"`.

`cSpc%[]`  An array channel specifier uses the first element of the array to hold the number of channels. The remaining elements are channel numbers. It is an error for the array to be shorter than the number of channels+1. This matches the data that is returned by `ChanList()`.

`cSpc%`  An integer channel specifier is either a channel number from 1 upwards, or –1 for all channels, -2 for visible channels, –3 for selected channels, -4 for waveform channels, -5 for all marker channels, -6 for all selected (or visible if none are selected) waveform channels, -7 for all visible waveform channels, -8 for all selected waveform channels, -9 for all idealised trace channels, -10 for all selected (or visible if no selection) channels of any type, -11 for all real marker channels and -12 for all marker and real marker channels.

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

# Include files

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

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

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

1. Search the folder where the file with the `#include` command lives. If this fails...
2. Search the folder of the file that included that file until we reach the top of the list of nested include files. If this fails...
3. Search any `\include` folder inside the `Signal7` folder inside your `My Documents` folder (this location was added at version 5.08). If this fails...
4. Search the `\include` folder inside the `Signal7Shared` folder inside the `Public Documents` folder (this location was added at version 5.08). If this fails...
5. Search any `\include` folder in the folder in which Signal is installed. If this fails...
6. Search the current folder.

Included files are always read from disk, even if they are already open. If you have an included file open and have modified it, but not saved it, the script compiler will detect this and stop the compilation with an error. You must save the included file to compile your script.

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

The `#include` command was added to Signal at version 4.00 and is not recognised by any version before this. A typical file using `#include` might start with:

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

### Opening included files

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

You can also open an include file by right-clicking on the name of any user-defined Proc or Func that is defined in an included file and selecting the Go to command. This will open the include file with the text caret at the start of the Proc or Func.

# Include files and debugging

You can debug a script that uses included files in exactly the same way as one that does not. If you step into a user-defined function or procedure that is in an included file, the included file will open and the stepping marker will show where you have reached. If you want to set a break point in an included file before you run the script, open the included file, set the break point and then run the script (leaving the included file open).

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

If you want Auto complete to work with included files you must check the Included files checkbox in the Edit menu Auto Complete... command.

# Program size limits

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

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

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

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

- The maximum number of global objects (variables and constants that are visible throughout the script) is 10000 (was 15000 before version 4.00). Note that there are about 500 global objects defined before any you create with a script, so you only have space for some 9500.

- The maximum number of local variables in all `proc` and `func` items is 12000 (was 7000 before version 4.00).

- The maximum number of strings that are longer than a few characters is 7000.

- The maximum number of `proc` and `func` items is 2767.

- The maximum number of characters in distinct literal strings is around 1000000. Before version 4.00 the limit was around 65000 characters.

- The maximum size of an array (product of all the array dimensions) is 100,000,000 elements.

We can expand these limits if anyone manages to hit them.

# Script functions by topic

This section lists script commands by function.

# Windows and views

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

| | |
|---|---|
| App | Get the application window view handle |
| ChanNumbers | Show or hide channel numbers |
| Colour | Get or set the palette entry associated with a screen item |
| ColourGet | Get the colour associated with a screen item |
| ColourSet | Set the colour associated with a screen item |
| Dup | Get the view handle of one of the duplicates of the current view |
| Draw | Draw invalid regions of the view (and set x axis range) |
| DrawAll | Update all invalid regions in all views |
| FileClose | Closes a window or windows |
| FileComment$ | Gets and sets the file comment for file views |
| FileConvert$ | Convert a foreign file to a Signal file and open it |
| FileName$ | Gets the file name associated with a window |
| FileNew | Opens an output file or a new text or data window |
| FileOpen | Opens an existing file (in a window) |
| FilePrint | Prints a range of data from the current view |
| FilePrintVisible | Prints the current view |
| FilePrintScreen | Prints all text-based and data views |
| FileQuit | Closes the Signal application |
| FileExportAs | Export from a data view in a variety of formats |
| FileSave | Save a view with same name |
| FileSaveAs | Save a view with specified new name |
| FocusHandle | Get the view handle of the window which has the input focus |
| FontGet | Read back information about the font |
| FontSet | Set the font for the current window |
| FrontView | Get or set the front window on screen |
| Grid | Get or set the visibility of the axis grids |
| LogHandle | Gets the view handle of the log window |
| PaletteGet | Get the RGB colour of a palette entry |
| PaletteSet | Set the RGB colour of a palette entry |
| SampleHandle | Gets the view handle of sampling windows and controls |
| ViewColour | Override application colours for a view |
| ViewColourGet | Get colour override settings for a view |
| ViewColourSet | Override application colours for a view |
| View | Change or override current view and get view handle |

| | |
|---|---|
| `ViewFind` | Get a view handle from a view title |
| `ViewKind` | Get the type of the current view or other view |
| `ViewLineNumbers` | Enable or disable line number display in text view |
| `ViewLink` | Get the view that owns the current view |
| `ViewList` | Form a list of handles of views that meet a specification |
| `ViewMaxLines` | Sets the maximum lines for a text view |
| `ViewStandard` | Returns a window to a standard state |
| `ViewUseColour` | Get and set monochrome/colour use for view |
| `ViewZoom` | Increases and decreases text view font size |
| `Window` | Sets the window size and position |
| `WindowDuplicate` | Duplicate a data view |
| `WindowGetPos` | Get window position |
| `WindowSize` | Changes the window size |
| `WindowTitle$` | Gets or changes the window title |
| `WindowVisible` | Sets or gets the visibility of the window (hide/show) |
| `XAxis` | Get or set the visibility of the x axis |
| `XAxisAttrib` | Get or set x axis attributes such as logarithmic |
| `XAxisMode` | Get or set the visibility of the x axis' features (eg large ticks) |
| `XAxisStyle` | Get or set the x axis major and minor tick spacing |
| `YAxis` | Get or set the visibility of the y axis |
| `YAxisAttrib` | Get or set y axis attributes such as logarithmic |
| `YAxisMode` | Get or set the visibility of the y axis' features (eg large ticks) |
| `YAxisStyle` | Get or set the y axis major and minor tick spacing |

## Data views

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

| | |
|---|---|
| `AppendFrame` | Add a new data frame to end of data |
| `DeleteFrame` | Delete a data frame if not on disk |
| `ExportChanFormat` | Set channel format for export |
| `ExportChanList` | Set list of channels for export |
| `ExportFrameList` | Set a list of frames for export |
| `ExportTextFormat` | Set format for text output of channels |
| `ExportTimeRange` | Set x axis range for export of data |
| `FileExportAs` | Export from a data view in a variety of formats |
| `FileApplyResource` | Apply a resource file to the current data view |
| `FileGlobalResource` | Set a global resource file |
| `FileSaveResource` | Create a resource file matching the current data view |
| `Frame` | Get or set the current frame |
| `FrameAbsStart` | Get or set the current frame absolute start time |
| `FrameComment$` | Get or set the comment with the current frame |
| `FrameCount` | The number of frames in the file or memory view |
| `FrameFlag` | Gets or sets a frame flag |
| `FrameList` | Fills an array with frame numbers according to a frame spec |
| `FrameMean` | Gets or sets the flag saying if the frame shows a mean or a total |
| `FrameSave` | Saves changed frame data or discards changes |
| `FrameState` | Gets or sets the frame state value |
| `FrameTag` | Gets or sets the frame tag |
| `FrameUserVar` | Gets or sets a frame user variable |
| `Maxtime` | Maximum x axis value in the current frame |
| `Mintime` | Minimum x axis value in the current frame |
| `Overdraw` | Enables or disables drawing the frame display list |
| `Overdraw3D` | Controls the 3D overdrawing the frame display list |
| `OverdrawFrames` | Sets a frame display list for the view |
| `OverdrawGetFrames` | Gets the frame numbers from the display list for the view |
| `ShowBuffer` | Get or set the frame buffer display state |
| `Sweeps` | Number of items processed into memory view |
| `TimeRatio` | Get the scaling factor from X units to seconds |
| `TimeUnits$` | Get the current time units |
| `XLow` | The start of the displayed area in x axis units |

| | |
|---|---|
| XHigh | The end of the displayed area in x axis units |
| XRange | Set the x axis range for next draw |
| XScroller | Show or hide the x axis scroll bar and controls |
| XTitle$ | X axis title. Can be set, in a memory or sampling document view |
| XUnits$ | X axis units. Can be set, in a memory or sampling document view |

# XY views

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

| | |
|---|---|
| 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 |
| XYInChan | Count XY points inside polygon defined by a channel |
| XYInCircle | Count the number of XY points within a circle |
| XYInRect | Count the number of XY points inside a rectangle |
| XYJoin | Get or set the point joining method |
| XYKey | Control the display of the XY view channel key |
| XYOffset | Set a drawing offset for an XY view channel |
| XYRange | Get rectangle containing one or more channels |
| XYSetChan | Create or modify an XY channel |
| XYSetData | Change the data in an XY channel data point |
| XYSize | Get or set maximum size of an XY channel |
| XYSort | Change the sort (and draw) order of a channel |

# Grid views

Grid views hold a rectangular array of cells.

| | |
|---|---|
| Draw | Set the first column to display |
| EditClear | Clear selected cells |
| EditCopy | Copy selected cells to the clipboard |
| EditCut | Cut selected cells to the clipboard |
| EditPaste | Replace selection with clipboard contents |
| EditSelectAll | Select entire grid |
| FileName$ | Get associated file name |
| FileNew | Create a new grid view |
| FileOpen | Open an existing grid view |
| FilePrint | Print current file |
| FileSave | Save as current name |
| FileSaveAs | Save with a different name |
| FontGet | Get the current font characteristics |
| FontSet | Set the current font characteristics |
| GrdAlign | Set the alignment of grid columns |
| GrdColWidth | Set the width of grid columns |
| GrdGet | Read back the contents of grid cells |
| GrdSet | Set the contents of grid cells |
| GrdShow | Show and hide column and row headers |
| GrdSize | Get and set the rows and columns in the grid |
| Modified | Get and set the modified and read-only state of views |
| MoveBy | Move the selection relative to the current position |
| MoveTo | Move the selection to an absolute position |
| ViewStandard | Reset the formatting to the standard state |
| XRange | Set the first column to display |
| YRange | Set the top row to display |

# Infomation windows

Information windows generate high-visibility displays for various purposes.

ColourSet            Set the application colours including for information windows
FontGet              Get the current font used in an information window
FontSet              Set the current font used in an information window
InfoCloseAll         Set the alignment of grid columns
InfoImage            Set or read back the information window background image settings
InfoNew              Create a new information window
InfoReset            Reset the timer in an information window
InfoSettings         Set or read back the current settings for an information window
InfoStart            Start the timer in one or more information windows
InfoStop             Stop the timer in one or more information windows
InfoText$            Set and get the text used in an information window
ViewColourSet        Set the colours used by an information window or all attached information windows

# Channels

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

BinError             Error value for a given bin
BinSize              X axis interval for waveforms or resolution for markers
BinToX               Bin or item number at x axis position
BinZero              X axis position of the first bin on the channel
Chan                 Converts a channel number in a string into a number
Chan$                Converts a channel number or list into a string
ChanColour           Override channel colours
ChanColourGet        Get channel colour override settings
ChanColourSet        Set channel colour overrides
ChanCount            Count channels of a given type
ChanDecorate         Set real marker channel drawing decoration
ChanDelete           Delete a channel from an XY view or an idealised trace
ChanDiff             Differentiate data in specified channels
ChanImage            Set a bitmap file to be used as the channel background
ChanIndex            Select the real marker value to be drawn & used
ChanKey              Control XY view keys
ChanIntgl            Integrate data in specified channels
ChanItems            Count items in a channel within an x axis range
ChanKind             Get the type of a channel
ChanList             Get a list of channels meeting a specification
ChanMean             Mean of waveform level within an x axis range
ChanMeasure          Take a variety of measurements on channel data
ChanNegate           Negate (invert) data in specified channels
ChanOffset           Offset data in specified channels by a constant value
ChanOrder            Modify and retrieve the channel order and y axis grouping
ChanPenWidth         Set the pen width for a data view channel
ChanPixel            Get the pixel size in channel units
ChanPoints           Number of data items in the channel in the frame
ChanRectify          Rectify data in specified channels
ChanScale            Scale data in specified channels by a constant value
ChanSearch           Scan a channel of data for a particular feature
ChanSelect           Select and report on selected state of channels
ChanShift            Shift data in specified channels left or right
ChanShow             Make a channel visible or invisible
ChanSmooth           Smooth (3 or 5 point) data in specified channels
ChanSubDC            Remove DC offset from data in specified channels
ChanRange            Get start and count of items in an  x axis range

| | |
|---|---|
| `ChanTitle$` | Get or set the channel title string |
| `ChanUnits$` | Get or set the channel units |
| `ChanValue` | Get channel data at a particular time or x axis position |
| `ChanVisible` | Get the visibility state of a channel |
| `ChanWeight` | Change the relative vertical space of a channel |
| `ChanZero` | Clear data in specified channels |
| `DrawMode` | Get or set display mode for a channel |
| `LastTime` | Find the previous item in a channel (and return values) |
| `MarkCode` | Get a marker code(s) |
| `MarkEdit` | Edit a marker code(s) |
| `MarkShow` | Select the marker code to be used for drawing |
| `MarkTime` | Change position of a marker |
| `MaxTime` | Time of last item on the channel |
| `MemChan` | Create a memory channel |
| `MemDeleteItem` | Delete one or more items from a memory channel |
| `MemDeleteTime` | Delete one or more items based on time in a memory channel |
| `MemGetItem` | Get information on item in memory channel |
| `MemImport` | Import items into a memory channel |
| `MemSetItem` | Edit or add an item in a memory channel |
| `MinMax` | Find minimum and maximum values (and positions) |
| `MinTime` | Time of first item on the channel |
| `NextTime` | Find the next item in a channel (and return values) |
| `Optimise` | Set reasonable y range for channels with axes |
| `VirtualChan` | Create virtual channel using expression |
| `XToBin` | Convert x axis value to bin number |
| `YLow` | Get lower limit of y axis for a channel |
| `YHigh` | Get upper limit of y axis for a channel |
| `YRange` | Set y axis range for a channel |

# Buffer

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

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

# Vertical cursors

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

| | |
|---|---|
| `Cursor` | Set or get the position of a cursor |
| `CursorActive` | Set and get active cursor parameters |

| | |
|---|---|
| `CursorDelete` | Delete a designated cursor |
| `CursorExists` | Test if a cursor exists |
| `CursorLabel` | Set or get the cursor label style |
| `CursorLabelPos` | Set or get the cursor label position |
| `CursorNew` | Add a new cursor (at a given position) |
| `CursorOpen` | Open the cursor value and cursor region dialogs, test if open |
| `CursorRenumber` | Renumber the cursors in ascending position order |
| `CursorSearch` | Cause active cursors to execute a search |
| `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 belong to a channel, but can be dragged to different channels within a view by the user. Horizontal cursors have a value and are drawn at the y axis position corresponding to the value. If the value is beyond the range of the y axis, the cursor is invisible. If you delete a channel with horizontal cursors, the cursors are deleted.

| | |
|---|---|
| `HCursor` | Set or get the position of a horizontal cursor |
| `HCursorActive` | Set or get horizontal active cursor parameters |
| `HCursorChan` | Gets the channel that a horizontal cursor belongs to |
| `HCursorDelete` | Delete a designated horizontal cursor |
| `HCursorExists` | Test if a horizontal cursor exists |
| `HCursorLabel` | Gets or sets the horizontal cursor style |
| `HCursorLabelPos` | Gets or sets the horizontal cursor position |
| `HCursorNew` | Add a new horizontal cursor on a channel (at a given position) |
| `HCursorRenumber` | Renumbers the cursors from bottom to the top of the view |
| `HCursorValid` | Test if an active horizontal cursor search succeeded |
| `HCursorVisible` | Get or set horizontal cursor visibility |

## Sampling configuration commands

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

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

| | |
|---|---|
| `SampleBurst` | Set or get the burst mode flag |
| `SampleClear` | Set the Sampling configuration to a known state |
| `SampleDigMark` | Add or remove the digital marker channel |
| `SampleKeyMark` | Add or remove the keyboard marker channel |
| `SampleMode` | Set or get the sweep mode for sampling |
| `SamplePause` | Set or get pause at sweep end flag |
| `SamplePoints` | Set or get the number of data points per ADC port |
| `SamplePorts` | Set or get which ADC ports to sample from |
| `SampleRate` | Set or get the ADC sample rate per channel in Hz |
| `SampleTrigger` | Set or get the triggered sweeps flag |
| `SampleVaryPoints` | Set or get the variable sweep points flag |
| `SampleWrite` | Control writing data to sampling file |
| `SampleXConv` | Control the use of external ADC convert pulses |
| `SampleXCPointsPerUnit` | Control the sample points per unit when using external ADC convert pulses |
| `SampleXCUnits$` | Control the X axis units when using external ADC convert pulses |
| `SampleZeroOffset` | Set or get the x-axis zero offset. |

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

| | |
|---|---|
| `SamplePortFull` | Set ADC port value for full input |
| `SamplePortName$` | Set ADC port title |
| `SamplePortOptions$` | Set ADC port online processing options |
| `SamplePortUnits$` | Set ADC port units |

| | |
|---|---|
| SamplePortZero | Set ADC port value for zero on the input |
| SampleTel | Set up of telegraph gains for an amplifier |

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

| | |
|---|---|
| SampleClamp | Set up or read back clamp settings and clamping sets |

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

| | |
|---|---|
| SampleAbsLevel | Set or get the absolute pulse levels flag |
| SampleAbsTime | Set or get the absolute pulse times flag |
| SampleDacFull | Set or get a DAC full-scale value |
| SampleDacMask | Set or get the DAC output enable mask |
| SampleDacUnits$ | Set or get a DAC units string |
| SampleDacZero | Set or get a DAC zero value |
| SampleDigOMask | Set or get the digital outputs enable mask |
| SampleMaxWaveRates | Set or get the maximum waveform rates flag |
| SampleOutClock | Set or get the outputs clock period |
| SampleOutMode | Set or get the outputs mode |

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

| | |
|---|---|
| SampleAuxStateNew | Add a new auxiliary states device to the list of those in use |
| SampleAuxStateParam | Set or get parameters for an auxiliary states device |
| SampleAuxStateParX | Set or get parameters for a specified auxiliary states device |
| SampleAuxStateRemove | Remove an auxiliary states device from the list of those in use |
| SampleAuxStateValue | Set or get an auxiliary states device's settings |
| SampleAuxStateValX | Set or get a specified auxiliary states device's settings |
| SampleDigIMask | Set or get digital inputs enable mask |
| SampleStates | Set or get states enable and number of extra states |
| SampleStatesIdle | Set or get states ordering cycles before idling |
| SampleStatesMode | Set or get multiple states mode |
| SampleStatesOrder | Set or get multiple states ordering mode |
| SampleStatesOptions | Set or get the multiple states options flag. |
| SampleStatesRepeats | Set or get multiple states repeats count |
| SampleStateDac | Set or get DAC data for individual state |
| SampleStateDig | Set or get digital bits for individual state |
| SampleStateRepeats | Set or get repeats for individual state |

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

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

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

| | |
|---|---|
| SampleArtefactGet | Get the artefact rejection settings |
| SampleArtefactSet | Set the artefact rejection settings |
| SampleAutoFile | Set or get the automatic file save flag |
| SampleAutoName$ | Set or get the template for automatic file naming |
| SampleLimitFrames | Set or clear the limit on the number of frames in the new file |
| SampleLimitSize | Set or clear the size limit of the new file |
| SampleLimitTime | Set or clear the limit on the overall sampling time |
| SampleAutoOpt() | Set or clear the automatic Y axis optimisation flag |

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

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

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

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

These commands correspond to the outputs sequencer.

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

# Runtime sampling commands

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

| | |
|---|---|
| `SampleAbort` | Exit from sampling and throw data away |
| `SampleAccept` | Accept or reject the current sweep |
| `SampleHandle` | Gets the view handle of sampling windows and controls |
| `SampleKey` | Adds to the keyboard marker channel, controls output sequencer |
| `SamplePeriHyst` | Alter hysteresis for peri-trigger + or - analog types |
| `SamplePeriLevel` | Alter threshold level for peri-trigger analog types |
| `SamplePeriLowLev` | Alter lower threshold level for peri-trigger =analog type |
| `SampleProtocol` | Set protocol to be used for state sequencing |
| `SampleReset` | Clear all data from the new file and get ready to start again |
| `SampleStart` | Start sampling after creating a new file view |

| SampleState | Set state for next frame to be sampled a new file view |
| SampleStatesPause | Set or get the current sample state sequencer pausing |
| SampleStatesReset | Reset states sequencing and pulse variations |
| SampleStatesRun | Set state sequencing run mode or manual |
| SampleStatesStep | Get the current states sequencing step counter. |
| SampleStatus | Get the current sampling state |
| SampleStop | Stop sampling and keep the data |
| SampleSweep | Start another sampling sweep |

# Analysis

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

| MeasureToChan | Create a memory channel holding marker data and associated measurement process |
| MeasureToXY | Create an XY view and associated measurement process |
| SetAmplitude | Set up an amplitude histogram view derived from a file view |
| SetAutoAv | Set up an auto-average multi-frame view derived from a file view |
| SetAverage | Set up a waveform average view derived from a file view |
| SetLeak | Set up a leak-subtracted view produced from a file view |
| SetOpCl | Set up an idealised trace using threshold crossings |
| SetOpClScan | Set up an idealised trace using the SCAN method |
| SetOpClAmp | Set up an amplitude histogram formed from an idealised trace |
| SetOpClBurst | Set up a burst time histogram formed from an idealised trace |
| SetOpClHist | Set up a dwell time histogram from an idealised trace |
| SetPower | Set up a power spectrum view derived from a file view |

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

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

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

| Process | Carry out the analysis process on the current frame from file |
| ProcessAll | Process all memory views attached to a file view |
| ProcessFrames | Carry out the analysis process on multiple frames from file |
| ProcessOnline | Carry out the analysis process during sampling |
| OpClFitRange | Fits an idealised patch clamp trace to observed data |
| OpClNoise | Makes baseline measurements to single channel data to allow SCAN analysis to be done |
| Sweeps | Number of items processed into memory view |

# Signal conditioner control

These functions control serial-line controlled signal conditioners.

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

# Editing operations

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

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

# 1401 access functions

These functions communicate with a 1401 when Signal is not sampling.

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

# CFS variables

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

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

# String functions

These functions manipulate strings and convert between strings and other variable types.

| | |
|---|---|
| Asc | ASCII code of first character of a string |
| Chr$ | Converts a code to a one character string |
| DelStr$ | Returns a string minus a sub-string |
| InStr | Searches for a string in another string |
| LCase$ | Returns lower case version of a string |
| Left$ | Returns the leftmost characters of a string |
| Len | Returns the length of a string or array |

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

# Array and matrix arithmetic

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

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

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

| | |
|---|---|
| ArrAdd | Adds an array or constant to an array |
| ArrConst | Copies an array, or sets an array to a constant value |
| ArrCWT | Carry out the Continuous Wavelet Transform on an array. |
| 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() |
| ArrMapImage | Convert a matrix (a 2-D array) to a bitmap image |
| ArrMul | Multiples an array by another array or constant |
| ArrRange | Find the range of values present in an array |
| ArrSort | Sort an array and optionally order others in the same way |
| ArrSpline | Interpolate one array to another using cubic splines |
| ArrSub | Subtract constant from array, or difference of two arrays |
| ArrSubR | Subtract array from constant, or reversed difference of arrays |
| ArrSum | Sum, mean and standard deviation of an array |
| Len | Returns the length of a string or array |
| MATDet | Calculate the determinant of a matrix |
| MATInv | Invert a matrix |
| MATMul | Multiply matrices and vectors. |
| MATSolve | Solve a set of linear equations |
| MATTrans | Transpose a matrix (also see the trans() operator) |
| PCA | Principal component analysis (singular value decomposition) |

# Mathematical functions

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

| | |
|---|---|
| Abs | Absolute value of a number or array |
| ATan | Arc tangent of number or array |
| Ceil | Ceiling of a number or array (next highest integral value) |
| Cos | Cosine of a number or array |

| | |
|---|---|
| Cosh | Hyperbolic cosine or a number or array |
| Exp | Exponential function of a number or array |
| Floor | Floor of a number or array (next lowest integral value) |
| Frac | Remove integral part of a number or array |
| GammaP | Incomplete Gamma function, used to calculate probabilities |
| LinPred | Use linear prediction to predict values or estimate spectra |
| Ln | Natural logarithm of a number or array |
| LnGamma | Natural logarithm of the Gamma function (use for big factorials) |
| Log | Logarithm to base 10 of a number or array |
| Max | Finds maximum of array or variables |
| Min | Finds minimum of array or variables |
| Pow | Raise a number or an array to a power |
| Rand | Generate pseudo-random numbers with uniform density |
| RandExp | Generate random numbers with exponential density |
| RandNorm | Generate random numbers with normal density |
| Round | Round a real number to the nearest integral value |
| Sin | Sine of a number or array |
| Sinh | Hyperbolic sine of a number or array |
| Sqrt | Square root of a number or an array |
| Tan | Tangent of a number or array |
| Tanh | Hyperbolic tangent of a number or array |
| Trunc | Remove fractional part of number or array |
| ZeroFind | Find a zero (root) of a user-defined continuous function |

## MatLab interaction

The following functions that can be used to make use of MatLab are built into Signal.

| | |
|---|---|
| MatLabOpen | Open a copy of MatLab for use |
| MatLabClose | Close a previously opened copy of MatLab |
| MatLabPut | Copy script data into a MatLab workspace |
| MatLabGet | Copy MatLab workspace data into script variable |
| MatLabEval | Cause MatLab to evaluate a command line or function |
| MatLabShow | Show and hide an open copy of MatLab |

## 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 |
| IIRApply | Apply an IIR filter bank filter to a waveform channel. |
| IIRBp | Create and/or apply an IIR bandpass filter |
| IIRBs | Create and/or apply an IIR bandstop filter |
| IIRComment$ | Get or set the comment for an IIR filter in the filter bank |
| IIRCreate | Create a new IIF filter definition in the filter bank |
| IIRHp | Create and/or apply an IIR highpass filter |
| IIRLp | Create and/or apply an IIR lowpass filter |
| IIRName$ | Get or set the name of an IIR filter in the filter bank |
| IIRNotch | Create and/or apply an IIR notch filter |

| `IIRReson` | Create and/or apply an IIR resonator filter |
|---|---|

# Fitting functions

The following fitting functions are built into Signal.

| `ChanFit` | A higher level fitting routine to emulate the fit dialog |
|---|---|
| `ChanFitCoef` | Get or set a fit coefficient |
| `ChanFitShow` | Show or hide a fit to a particular channel |
| `ChanFitValue` | Get the value of a fitted function |
| `FitData` | Fit a selected function to arrays of x,y data points |
| `FitExp` | Fit to multiple exponentials |
| `FitGauss` | Fit to multiple gaussians to data |
| `FitLine` | Fit a straight line to waveform channel data |
| `FitLinear` | Fit to linear combination of user-defined functions |
| `FitNLUser` | Fit to non-linear user-defined function |
| `OpClFitRange` | Fits an idealised trace using the filter characteristics |
| `FitPoly` | Fit to a polynomial |
| `FitSigmoid` | Fit to a sigmoid |
| `FitSin` | Fit to multiple sinusoids |

# Image related

These are the script commands that can be used with images or bitmaps:

| `ArrMapImage` | Map a matrix into a bitmap using a colour scale. |
|---|---|
| `ChanImage` | Set an image to display behind data. |
| `ColourGet` | Collect information about a colour scale. |
| `ColourSet` | Create a colour scale. |
| `EditCopy` | Create a bitmap on the clipboard from a matrix or the current view. |
| `EditImageSave` | Save a clipboard image to a file. |
| `EditPaste` | Copy a clipboard image to an array. |
| `MousePointer` | Create or load mouse pointer |
| `Spline2D` | Create images by splining 2D and 3D data. |
| XY Views | These can be used for line and point drawing. |

# User interaction commands

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

| `Input` | Prompt user for a number in a defined range |
|---|---|
| `Input$` | Prompt user for a string with a list of acceptable characters |
| `Interact` | Allow user to interact with data |
| `Message` | Display a message in a box, wait for OK |
| `MousePointer` | Control the mouse pointer that is in use |
| `Print` | Formatted text output to a file or window |
| `PrintLog` | Formatted text output to the Log window |
| `Print$` | Formatted text output to a string |
| `Query` | Ask a user a question, wait for response |
| `Sound` | Play a tone or a .wav file |
| `Speak` | Convert text to speech on systems that support this |
| `Yield` | Give idle time to the system; delay for a time |
| `YieldSystem` | Surrender current time slice and sleep Signal |

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):

| `DlgAllow` | Set allowed actions and change and idle functions |
|---|---|
| `DlgButton` | Add buttons to the dialog and modify existing buttons |
| `DlgChan` | Define a dialog entry as prompt and channel selection |
| `DlgCheck` | Define a dialog item as a check box |

| | |
|---|---|
| `DlgCreate` | Start a dialog definition |
| `DlgEnable` | Enable and disable dialog items in change or idle functions |
| `DlgFont` | Set the font to be used in all script-created dialogs. |
| `DlgGroup` | Position a group box in the dialog |
| `DlgInteger` | Define a dialog entry as prompt and integer number input |
| `DlgLabel` | Define a dialog entry as prompt only |
| `DlgList` | Define a dialog entry as prompt and selection from a list |
| `DlgMouse` | Set the mouse position, link user functions to mouse actions |
| `DlgReal` | Define a dialog entry as prompt and real number input |
| `DlgShow` | Display the dialog, get values of fields |
| `DlgSlider` | Define a dialog entry as a slider with optional prompt |
| `DlgString` | Define a dialog entry as prompt and string input |
| `DlgText` | Define a fixed text string for the dialog |
| `DlgValue` | Gives access to dialog fields in change and idle functions |
| `DlgVisible` | Show and hide dialog items in change or idle functions |
| `DlgXValue` | Define a dialog entry to collect an x axis value |

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

| | |
|---|---|
| `App` | Get the view handle of the toolbars |
| `Toolbar` | Let the user interact with the toolbar |
| `ToolbarClear` | Remove all defined buttons from the toolbar |
| `ToolbarEnable` | Get or set the enable state of a toolbar button |
| `ToolbarMouse` | Link user functions to mouse actions |
| `ToolbarSet` | Add a button (and associate a function with it) |
| `ToolbarText` | Display a message using the toolbar |
| `ToolbarVisible` | Get or set the visibility of the toolbar |
| `SampleBar` | Controls the sample bar buttons |
| `ScriptBar` | Controls the script bar buttons |

# File system

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

| | |
|---|---|
| `FileConvert$` | Convert a foreign file to a Signal CFS file and open it |
| `FileCopy` | Copies a file from one place to another |
| `FileDate$` | Retrieve date of creation of a Signal data file |
| `FileDelete` | Delete one or more files |
| `FileList` | Get a list of files or directories |
| `FilePath$` | Get the current directory or other special directories |
| `FilePathSet` | Change the current directory or directory for new data files |
| `FileSize` | Retrieve the size of a data file associated with a view |
| `FileTime$` | Retrieve the time of creation of a Signal data file |
| `FileTimeDate` | Retrieve time and date of creation of Signal data file as numbers |
| `FileTimeDateSet` | Set the time of creation (start of sampling) of a data file |

# Text files

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

| | |
|---|---|
| `FileNew` | Open a new text file in a window |
| `FileOpen` | Open a text file in a window or for reading and writing |
| `FileSaveAs` | Save a view in a variety of formats, including text |
| `Print` | Write formatted output to a file or log window |
| `Read` | Extract data from a text file |
| `ReadSetup` | Specify what characters will delimit read text |
| `TabSettings` | Get or set the tab and indent settings for a text view |
| `ViewMaxLines` | Get or set the maximum nimber of text lines allowed |

# Binary files

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

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

# Serial line control

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

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

# Debugging operations

These functions can be used when debugging a script.

| | |
|---|---|
| App(-4) | Get the number of system handles held by Signal |
| Debug | Set a permanent break point and disable/enable debugging |
| DebugHeap() | Get information about Signal memory usage |
| DebugList | List internal Signal script objects |
| DebugOpts | Gets and optionally sets system level debugging options |
| Eval | Convert the argument to text and display |

# Environment

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

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

## Multiple monitor support

These functions give the script writer control over positioning the Signal application and Signal windows in a multiple monitor environment.

DlgCreate          Position a script-controllable dialog relative to monitors
DlgMouse          Set mouse position relative to dialog (useful with multiple monitors)
System          Get monitor count, positions and identify the primary monitor
Window          Position a script-controllable window relative to monitors
WindowGetPos          Get position of a window, optionally including screen information
WindowVisible          Maximise the application over the entire desktop

## Alphabetical script function index

## A

## 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 function returns the application window handle, and some special Signal view handles. Other Signal view handles are returned by functions creating or by finding them. Other sampling-related special window handles can be obtained by using `SampleHandle()`.

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

type%   This specifies the window handle to return, or a negative value can be used to retrieve special values:

-1       100 times the program revision.
-2       The highest view handle currently used by Signal - the highest value of `type%` that returns a value.
-3       The program serial number.
-4       The number of system handles held by the application (for debugging) or 0 if this is not supported. Added at 4.02.
-5       The number of GDI (graphic) objects used by the application (for debugging) or 0 if this is not supported. Added at 4.02.
-6       Indicates if this is a 64-bit build of Signal (1) or a 32-bit build (0). Added at version 6.00.
-7       The number of User graphic objects used by the application (for debugging) or 0 if this is not supported. Added at version 6.04.

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

| | | |
|---|---|---|
| 0 Application | 4 Edit toolbar | 8 Sequencer control panel |
| 1 Main toolbar | 5 Script bar | 9 States control bar |
| 2 Status bar | 6 Sample bar | 10 Clamping toolbar |
| 3 Running script | 7 Sample control panel | |

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

For example:

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

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

# AppendFrame()

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

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

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

Returns  Zero or a negative error code.

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

# Array commands

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

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

The array commands are:

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

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

`source`  A real or integer array, or a view.

`value`   A real or integer value

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

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

```
var fred[100], tom[100], jim%[200];
var val:=0.5;
ArrAdd(fred[],jim%[]);      'Add elements 0-99 of jim% to fred
ArrSub(fred[],tom[]);       'Subtract each tom from each fred
ArrSubR(fred[]),tom[]);     'The negative of the above result
ArrSub(jim%[],val);         'Subract val from all elements of jim%
ArrAdd(jim%[2:8],10);       'Add value 10 to elements 2-9 of jim%
```

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

```
var chans[2][100];          'Array of 2 rows and 100 columns
var data[3][30];            'Array of 3 rows and 30 columns
var jim%[200]
ArrDot(chans[0][],chans[1][]); 'form the dot product of two rows
```

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

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

and this would do exactly the same

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

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

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

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

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

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

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

# ArrAdd()

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

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

dest    The destination array (1 to 5 dimensions).

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

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

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

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

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

**See also:**
```
ArrXXX(), ArrConst(), ArrDiv(), ArrDivR(), ArrDot(), ArrMul(), ArrSub(), ArrSubR(),
ArrSum(), Len(), BuffAdd(), BuffAddTo()
```

# ArrConst()

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

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

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

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

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

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

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

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

**See also:**
```
ArrXXX(), ArrAdd(), ArrDiv(), ArrDivR(), ArrDot(), ArrMul(), ArrSub(), ArrSubR(),
ArrSum(), BuffCopy(), BuffCopyTo(), BuffExchange(), Len()
```

# ArrConv()

This function performs the discrete convolution of a signal f with a pulse g. Convolutions arise when you want to know the effect of passing a signal through a process. If this process is a filter, the g pulse is the impulse response of the filter. There is a nice description of the discrete convolution here. Convolutions can be calculated using the Fast Fourier Transform (FFT) or by summing array products.

```
Func ArrConv(conv[], const f[], const g[]{, flags% {,nOff%}});
```

conv    A vector to hold the result. If this vector follows the rules for being resizeable with the resize command, it can be any length and the command will resize it to the correct length for the result. Otherwise, it must be the correct size for the result, which is `Len(f)` or `Len(f)+Len(g)`, depending on `flags%`. This can be the same vector as `f`.

f       A vector that we think of as the signal. This is not changed by the operation.

g       A vector we think of as the pulse. This is not changed by the operation.

flags%  This controls how we perform the convolution and the size of the result. If omitted it takes the value 0, which performs the calculation using FFTs and returns a result which is `Len(f)+Len(g)` long. It is the sum of:

   1    This causes the calculation to be done by summing all the array products instead of using FFTs. This can be faster than using FFTs when g is short compared to f.

   2    The result array is the same length as `f` instead of `Len(f)+Len(g)`. You may also wish to set `nOff%` to determine which points of the convolution to copy to the result.

nOff%   An optional argument used when 2 is added to `flags%`. If present it must be in the range 0 to `Len(g)-1`. If omitted it takes the value `Len(g)/2`. It is used to determine how many points of the convolution result of length `Len(f)+Len(g)` to skip before copying the result to `conv[]`.

Returns  If all went well, the return value is the length of the result array, `conv[]`. Otherwise it returns a negative error code.

**Details**

The discrete convolution can be expressed generally as:

```
result[n] = Summ(f[m]*g[n-m]) = Summ(f[n-m]*g[m])
```

where m ranges from minus to plus infinity and elements of `f` and `g` that do not exist have the value 0. From this definition, you can see that `f` and `g` are interchangeable. This is also the case in our implementation, however the code that does the calculation without using the FFT is optimised on the assumption that `Len(g) <= Len(f)`.

There are two ways commonly used to calculate the convolution:

1. By computing the sums as described above. This is a relatively slow operation, of order (n*m) operations (written as O(nm) where n and m are the sizes of the arrays).

2. By taking advantage of the Convolution theorem which takes O(N Log N) operations where N is the next power of two that is greater than or equal in size to Len(f)+Len(g). This takes the forward FFT (Fast Fourier Transform) of the two source arrays (after zero-padding them to N points), takes the product of the transforms, then takes the inverse FFT of the result and adjusts it for the zero padding. The zero padding requirement is because the FFT we use is not implemented for sizes other than powers of 2.

### ArrFilt command

The `ArrFilt()` script command is optimised to apply an FIR digital filter to a vector, and behaves in a very similar way to using this command with `flags%` set to 3 and the `g` vector set to the `coef[]` array backwards. There are also differences caused by how we treat the first and last points to allow for end effects. `ArrFilt()` is optimised for the case of `Len(f)` much greater than `Len(g)`.

# ArrCWT()

This command computes the Continuous Wavelet Transform (CWT) of a sequence of real value for several commonly-used Wavelet functions. The CWT transforms an input into the amplitudes and positions of a Wavelet at a variety of scales. The output of this command can be used as the input to the `ArrMapImage()` command to create a bitmap that displays the result of the transform. You can add the bitmap to an XY view to display the result using the `ChanImage()` command. See this example.

The input to the command is a vector of `n%` real values `src[n%]`. The output is a matrix of real or complex values `dest[n%][s%]` where `s%` is the number of scales. The scales are arranged to increase in an exponential manner; they are not linearly increasing. Each output `dest[][s%]` holds results for one scale.

The command has several variants: a setup variant to set the type of transform, a variant to return information and variants to apply the transform.

### Setup transform

This variant is used first to set the type of output requires, the wavelet family to use and to determine the range of scales that are to be generated. The default values for omitted arguments, or for all arguments if you never call this setup routine, the default are chosen so that the output is likely to be useful in all cases.

```
Func ArrCWT(wav%{, dScl{, minScl{, m%|w0}}}});
```

wav%     This determines the wavelet family to use for the transform. You have a choice of:

| Value | Family | Output | Description |
|---|---|---|---|
| 0 | Morlet | Complex | A sinusoid windowed by a Gaussian. The `w0` argument sets the angular frequency (the $\omega_0$ parameter), the default is 6. This is the default wavelet family if you do not call this setup function. |
| 1 | Paul | Complex | The `m%` argument sets the order, the default is 4. |
| 2 | DOG | Real | Derivative of a Gaussian. The `m%` argument sets the order of the derivative. The default is 2, which is the Mexican Hat wavelet. |

dScl     This determines the ratio of successive scales. The scale of the $j^{th}$ scale is given by $s_j = s_0 \, 2^{j*dScl}$. where $s_j$ is the wavelet size in points. The $s_0$ value is set by the next argument (`minSc%`). Put another way, $1/dScl$ is the number of scales to double the scale length. The default value of `dScl` is 0.25 (4 scales per octave).

minScl     This determines the minimum scale value for the result and is width of the smallest wavelet, in points. The default value is 2, which is probably the minimum value that can reasonably be used. Values are limited to a minimum of 1.

w0      This sets the angular frequency of the Morlet wavelet. We limit the values to the range 2 to 20. The default value (6.0) is reasonable.

m%      This sets the order of the Paul and DOG wavelets. We limit the values to the range 1 to 10. The default values are 4 (Paul) and 2 (DOG).

Returns  0 for success or -1 if the combination of wavelet family and output format do not make sense or one of the arguments is out of range.

## Read back information

```
Func ArrCWT(get%{, maxScl|scale|freq|index{, sInt}});
```

get%    A negative value that selects information to be returned:

| Value | Returned information |
|---|---|
| -1 | The wavelet family as set by the `wav%` argument. |
| -2 | The ratio of successive scales, as set by `dScl`. |
| -3 | The minimum scale as set by `minScl`. |
| -4 | The `wPar` argument value for the current settings. |
| -5 | You must provide the `maxScl` argument. This returns the number of scales required to span the range `minScl` up to and including `maxScl`. This is provided as a convenience and is calculated as: `Ceil(Ln(maxScl/minScl)/(Ln(2) * dScl))`. The maximum sensible scale is the number of points in the input waveform, although setting a scale of this size is unlikely to be useful. The maximum scale size in most cases is likely to be some fraction of this. The result is -1 if the scale value is |
| -6 | Peak frequency that corresponds to a given scale. You must supply the `scale` and optionally `sInt`. The power spectrum of each wavelet spans a range of frequencies; the calculated value is the peak of the power spectrum. In the case of a Morlet wavelet, this is the frequency of the sinusoid. The result is 0 if `scale` or `sInt` are zero or negative. |
| -7 | Scale that corresponds to a given peak frequency. You must supply the `freq` and optionally `sInt`. See the -6 option for the meaning of frequency. The result is 0 if `freq` or `sInt` are zero or negative. |
| -8 | Cone of influence value for a given scale. You must provide the `scale`. Due to edge effects, the first and last points of the results at each scale are unreliable. If `sInt` is omitted the returned value is the number of unreliable points at the start (and end) of the results for this scale value, otherwise it is the duration of the unreliable region in x axis units. |
| -9 | The index of the last dimension of the output matrix (`dest[][index]` or `dest[][][index]`) that corresponds to the input `scale` value. The result is -1 if the scale is a stupid value (zero or negative). It will also be negative if `scale` is less than the `minScl` value. |
| -10 | The scale value that corresponds to the last output matrix dimension (`dest[][index]` or `dest[][][index]`). The result is 0 if the index is negative or stupidly large. |

maxScl  Used when `get%` is -5 to set a scale that is greater than `minScl`. This is the width of the largest wavelet, in points, that you wish to have values calculated for.

scale   The wavelet scale to use for the -6, -8 and -9 `get%` values.

freq    A frequency in Hz to use for the -7 `get%` value.

index   The index into the last dimension of the output array to convert to a scale when `get%` is -10.

sInt    The interval between the source item values (and defaults to 1 if omitted). If the source data comes from a waveform channel in a data view, this will be in seconds and should be set to `BinSize(chan%)`. This is normally supplied for `get%` values -6 and -7 when converting between a scale and a frequency. If you omit it, frequencies become per point, not per second, and the cone of influence value is returned in points, not x axis units.

Returns  The value requested.

**Apply the transform**

This variant applies the Continuous Wavelet Transform to a source vector to generate a matrix of results. The time to process this data is approximately proportional to `nData * Log₂(nData) * nScale`, where `nData` is the number of source data points, and `nScale` is the number of output scales to calculate. To give you an idea of the time, on my Core i-7 4770K 3.5 GHz CPU, to process 60000 points to 110 scales took 518 ms. A Core i-5 laptop I also use takes maybe 50% longer.

```
Func ArrCWT(src[], dest[][]{, out%});
Func ArrCWT(src[], dest[2][][]{, out%});
```

src    The data to be transformed. The length of this array, henceforth `n%`, sets the size of the first dimension of the `dest` matrix. This data is typically taken from a data channel using the View(fh%, ch%)[] script mechanism. The length of the array must be at least 8 points and is currently limited to a maximum of 262144 points (which is 2 to the power 18).

dest    The destination matrix to hold the result of the transform. If the output of the transform is purely real, this array has two dimensions, `dest[n%][s%]`, where `n%` is the length of the `src` array and `s%` is the number of scales. If the transform output is complex, or is set to the amplitude and phase of the result, `dest` has three dimensions, and the first dimensions is of size at least 2.

out%    If present, this sets the desired output format (which must be compatible with `dest`). You have a choice of:

| Value | Result | Description |
|---|---|---|
| 0 | [n%][s%] | The absolute value of a complex transform (Morlet or Paul) or the real result of a Real transform (DOG). This is the default value if `dest` is a matrix (has 2 dimensions). |
| 1 | [n%][s%] | The power of the result. |
| 2 | [2][n%][s%] | The result of a complex transform as pairs of Real and Imaginary components at each output point. [0][i%][j%] is the Real component, [1][i%][j%] is the imaginary component. This is the default value if `dest` has three dimensions. |
| 3 | [2][n%][s%] | The result of a complex transform as amplitude and phase. [0][i%][j%] is the amplitude, [1][i%][j%] is the phase. |

Return    0 for success or -1 if `out%` is not compatible with `dest`.

**See also:**
ArrMapImage(), View(), ChanImage()

# The Continuous Wavelet Transform

The Continuous Wavelet Transform (CWT) of a sequence of N data points $x_0, x_1 ... x_N$ is:

$$W_n(s) = \sum_{n'=0}^{N-1} x_{n'} \overline{\psi} \left[ \frac{(n'-n)\partial t}{s} \right]$$

where $n$ is the index of the resulting entry in the CWT vector at scale $s$, $n'$ is the translation along the x axis, $\partial t$ is the sampling interval along the x axis, and $\psi$ is an admissible wavelet function and a bar means complex conjugate. Applying the CWT in this form is computationally expensive so we use the Convolution theorem to decrease the order of operations from $O(n^2)$ to $O(n\log(n))$ per scale.

The term admissible means that the wavelet function must have a mean value of zero and be localised in both the x axis space and in frequency space. A smoothly changing, non-infinite waveform that tapers to zero at each end and has a mean of zero will satisfy these conditions.

The scale of a wavelet can be thought of as its x axis width. The larger the scale, the more x axis space it occupies. To make wavelets at different scales comparable, they are normalised so that the power of the wavelet at each scale is unity.

You can find a detailed explanation by following this link. This article, "A Practical Guide to Wavelet Analysis" by C. Torrence and G. P. Compo, 1998, is well worth reading and contains a detailed description of the methodology.

# Wavelet families

We support three wavelet families: Morlet, Paul and DOG (Derivative of Gaussian). You can find a detailed explanation of these and the methodology used in this paper. In the equations, below, the $\eta$ is a dimensionless parameter (most often interpreted as time) and is the x axis value in the images.

## Morlet

This is a complex wavelet given by the equation:

$$\pi^{-1/4} e^{i\omega_0\eta} e^{-\eta^2/2}$$

The $\omega_0$ parameter is the angular frequency (the `w0` argument to the set up call). The image below shows the Morlet wavelet with the default value of $\omega_0$, which is 6. The wavelet is composed of a sinusoid multiplied by a Gaussian. If you are interested in the frequency content of a signal, for example if you are looking for the equivalent of a sonogram, then this is the wavelet family to use.
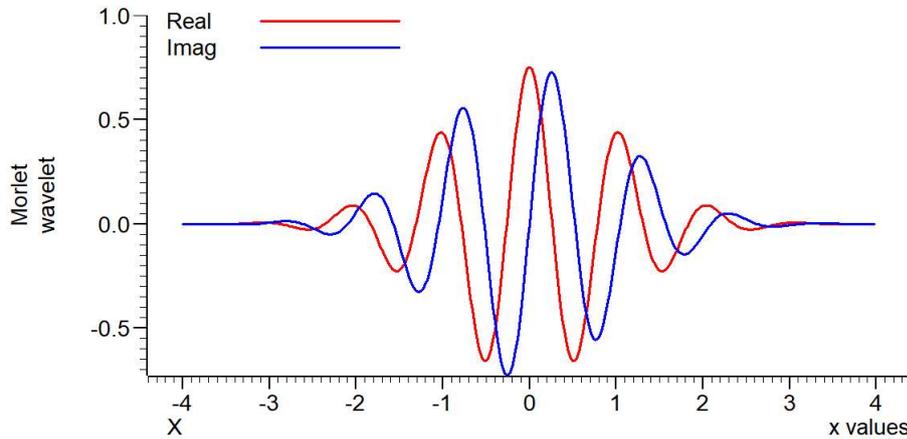


## Paul

This is a complex wavelet given by the equation:

$$\frac{2^m i^m m!}{\sqrt{\pi(2m)!}} (1 - i\eta)^{-(m+1)}$$

The *m* parameter is an integer and is set by the `m%` argument of the setup call. The image shows the Paul wavelet for the default value of *m* (4).
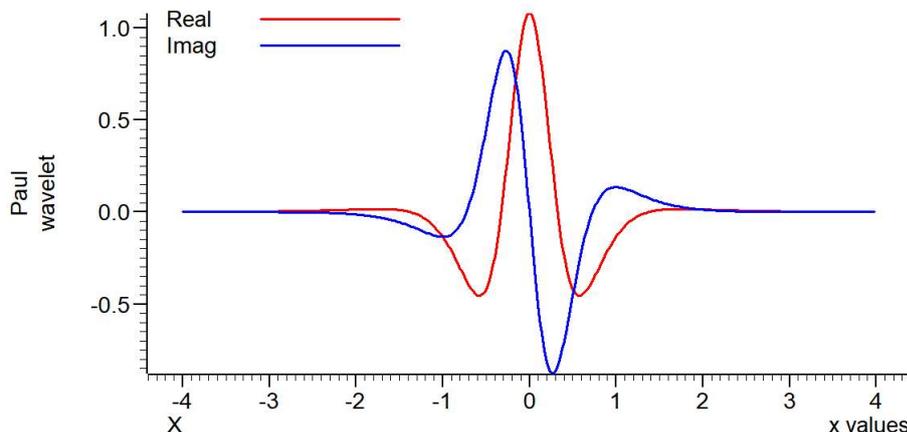
### Derivative of Gaussian (DOG)

This is a wholly real wavelet given by the equation:

$$\frac{(-1)^{m+1}}{\Gamma\left(m+\frac{1}{2}\right)} \frac{d^m}{d\eta^m}\left(e^{-\eta^2/2}\right)$$

The *m* parameter is an integer and is set by the `m%` argument of the setup call. The image shows the DOG wavelet for the default value of m (2), which generates the so-called Mexican Hat wavelet.
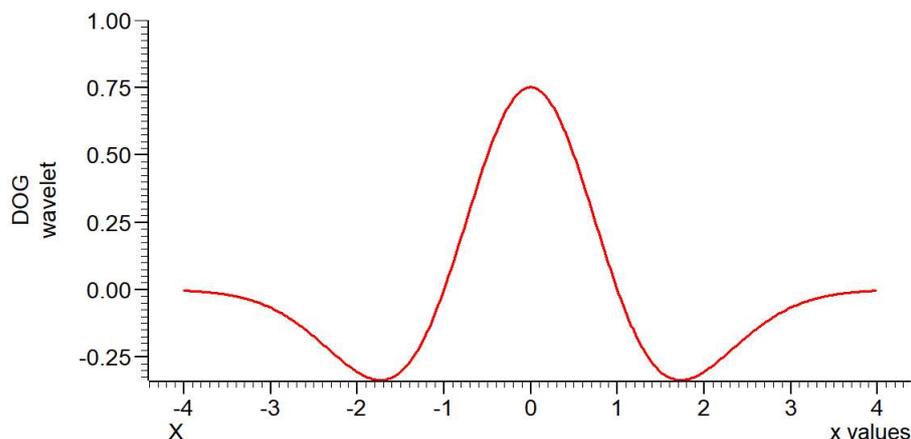


# Example

This example script demonstrates how to use the `ArrCWT()` command to generate the following image from the `demo.smr` data file supplied with the Spike2 distribution and converted to `demo.cfs`.
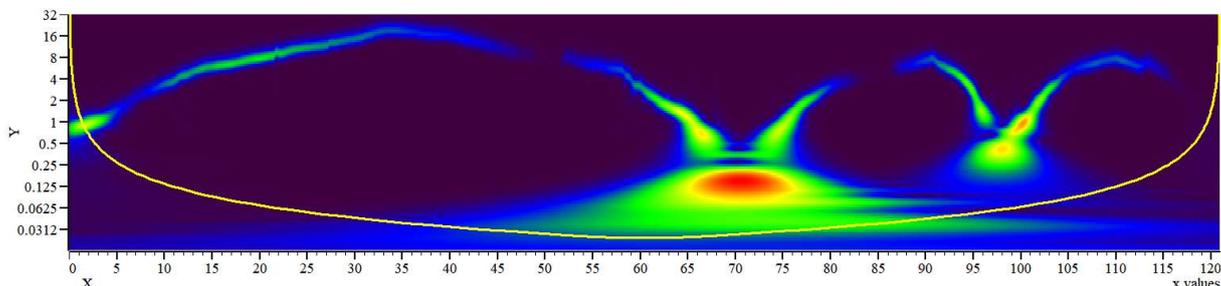


This example displays the Continuous Wavelet Transform of the waveform channel from the file using the Morlet wavelet. The Y axis is showing frequency in Hz and the X axis the time in seconds. The yellow line shows the *Cone of Influence*, which is discussed below. We have broken the script up into several parts. Paste the parts (in order) into an empty script view to generate the full script. The image shows significant energy at very low frequencies. This is because the data has a large DC offset. If you remove any DC component from the waveform, then run the analysis again, the very low frequency energy is removed.

### Get the waveform data and setup values

The first part of the script opens the data file and fills an array with the waveform data to transform:

```
'Open data file and extract data array
var dataPath$:=FilePath$(-4)+"\\demo.cfs"; 'C:\Users\...\Documents\Signal7\demo.cfs
var vh% := FileOpen(dataPath$, 0, 1); if (vh% < 0) | ViewKind() then Message("No view"); halt en

const ch% := 1;                        'Data channel in the view
var wavFamily% := 0;                   'Choice of wavelet family, 0=Morlet
var minScl := 3;                       'Sets minimum scale to start the analysis at
var dScl := 0.1;                       'Sets 10 scales per frequency doubling

var sInt:= BinSize(ch%);               'Data channel sampling interval
var time := MaxTime();                 'How far we need the analysis to go
var nX% := time / sInt;                'Maximum size of array assuming contiguous data
var maxScl := nX%/2;                   'Maximum scale for the analysis
var a[nX%];                            'Array to analyse
ArrConst(a[], view(0, ch%).[]);        'Fill array from chosen channel - could be used directly
```

This first couple of lines open the `demo.cfs` data file and stop the script with a message if it is not found.

The next lines set the channel to analyse, set the wavelet family to use, the minimum wavelet scale to process and the number of scales to generate per octave (doubling of frequency). The minimum scale is expressed in terms of data points, that is a wavelet with 3 data points. You might choose to start the analysis at a higher scale to save calculation time (it costs the same time to analyse each line, so analysis time is proportional to the number of scales analysed).

The last group of lines extract information from the file that we use for analysis parameters, sets the maximum scale to generate (in this case being half the data length, in points, but it could be quite a bit less), and allocates a data array large enough to hold the waveform data in the file, and then copies the data from the file into the array.

### Perform the Continuous Wavelet Transform
This part of the script sets up the transform, allocates memory for the result, and runs the analysis:

```
'Do CWT analysis
ArrCWT(wavFamily%, dScl, minScl, 6);   'Setup Morlet with w0 of 6
var nScl% := ArrCWT(-5, maxScl);       'Number of scales to span minScl-maxScl
var m[nX%][nScl%];                      'Matrix to store the transform
ArrCWT(a, m, 0);                       'Transform the array
```

The first line prepares the `ArrCWT()` command by setting the wavelet family to use, the scale growth between successive output lines, the scale of the first output line and the parameter to define which wavelet of the selected family to use. This example is set to use the Morlet family of wavelets, which is particularly well-suited when you want to extract frequency information from the raw data as the wavelet is the product of a sinusoid and a Gaussian.

The second line gets the number of lines of output that are required to span the range of scales from `minScl` to `maxScl`. In this case we have stopped the analysis at a scale of half the number of input data points. You should note that the larger the scale, the more the result is smeared in time, so if you were not interested in the results at low frequencies (large scales), you might choose to stop at a smaller scale.

The third line generates the matrix to hold the result. For each point in the source data array we get `nScl%` results, one for each scale.

The last line generates the results as the absolute value of the result (which is complex for a Morlet wavelet).

### Draw the matrix as a bitmap and place it in an XY view
The results are a little difficult to interpret as numbers; it is much easier to see what is going on in a picture. The next section of the script generates a bitmap from the results and then displays it as the background image of an XY view.

```
'Draw the matrix using ArrMapImage()
ArrMapImage(m, "<CB>");                'Make a bitmap in the clipboard
FileNew(12, 1);                        'Create new XY view
XRange(0, nX%*sInt);                   'Set display x range to match source data
var fMax := ArrCWT(-6, minScl, sInt);  'Convert min scale to max freq
var fMin := ArrCWT(-6, maxScl, sInt);  'and max scale to min freq
YRange(-1, fMin, fMax);                'Set display range (all channels)
YAxisAttrib(-1, 1);                    'Set the Y axis as logarithmic
YAxisStyle(-1, 0, Log(2));             'Set Y axis major ticks at powers of 2
ChanImage(1, "<CB>");                  'Set the clipboard image in the view
var ret% := ChanImage(1, 2, 1.0, 0, fMin, nX%*sInt, fMax);  'Place the image
```

The first line converts the matrix holding the results into a bitmap and copies it to the clipboard. We have chosen to use the default parameters, so this has found the largest and smallest values in the result matrix and mapped these onto the Rainbow colour scale. By setting different arguments you can set other colour scales and define you own range of values to map onto the colour scale.

The second line creates an XY view and makes it visible. The view has one, empty data channel that we use in the final code section (below) to draw the Cone of Influence. A more responsible script would save the view handle of the XY view and check that there was no error when creating it; we omit this here to keep the code short.

The third line sets the X axis range to display, which is set to match the time range of the original data.

Lines 4-8 of this section generate a logarithmic Y axis, labelled in powers of two. To do this the code asks the `ArrCWT` command for the frequencies that match the minimum and maximum scales. We then set the Y axis to display this range. However, if we stop at this point, the display will show a linear axis. However, we know that the bitmap shows scales that increase by a constant factor for each line. We can express this as a logarithmic axis. The `YAxisAttrib()` command switches to a logarithmic axis, which will be in powers of 10, by default. For this data file this generates major ticks at 0.1, 1 and 10 Hz. To illustrate how to choose other tick marks, the `YAxisStyle()` command sets the major ticks at powers of two, and the result for this file shows ticks at 32, 16, 8, 4, 2 and so on.

The last two lines set a channel image as the image on the clipboard generated by `ArrMapImage()` and the final line ties this image to specific co-ordinates in the view, rather than being a general background.

### Draw the *Cone of Influence*

The final section draws the so-called *Cone of Influence*. There are edge effects introduced into the CWT due to the mathematical short-cut of using the Fourier Transform to do the calculation. The problem is that the Fourier transform assumes that the raw data is cyclic, but this is not the case. Further, the Fourier Transform we use works on a power of 2 data points, so we have to extend the raw data to the next power of two points by zero-padding. This will introduce discontinuities into the data which affects the results. Of course, if your data naturally reaches 0 at the start and end, these effects are greatly mitigated. The *Cone of Influence* marks the region of the result where the results are unreliable due to this effect. The final part of the code draws a line (in yellow) to illustrate this. Areas of the image below this line should be considered unreliable.

```
'Draw Cone of Influence to XY view channel 1
var i%, coi, freq;
var dLimit := (nX%-1) * sInt;        'x axis at right hand side of image
for i% := 0 to nScl%-1 do            'Loop over scales for left side of the COI
    coi := CalcCOI(i%, freq);        'Calculate COI and associated frequency
    if (2*coi >= dLimit) then break endif;
    XYAddData(1, coi, freq);         'Add this COI point
next;

while (i% > 0) do                    'Draw the right side of the COI
    i% := i% -1;
    coi := CalcCOI(i%, freq);        'Calculate COI and associated frequency
    XYAddData(1, dLimit-coi, freq);  'Add this COI point
wend;

XYDrawMode(1,2,0);                   'Set data points to size 0
XYDrawMode(1,4,3);                   'Set lines to size 3
XYJoin(1, 1);                        'Join consecutive points
ChanColourSet(1,1,1,1,0);           'Make lines yellow

func CalcCOI(i%, &freq)              'Calc COI value and associated frequency
var scale := minScl*pow(2, dScl*i%); 'Generate scale
freq := ArrCWT(-6, scale, sInt);     'Generate frequency for this scale
return ArrCWT(-8, scale, sInt);      'Generate COI value
end;
```

The `CalcCOI(i%, &freq)` user-defined function works out the centre frequency and returns the COI value for a particular line of the output. The rest of the code draw the COI values forwards until it reaches a value that would be more than half-way through the data (to draw the left-hand side), then works backwards to draw the right-hand side.

# ArrDiff()

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

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

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

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

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

**See also:**
`ArrXXX(), ArrFilt(), ArrIntgl(), ChanDiff(), Len()`

# ArrDiv()

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

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

`dest` An array of real or integer values.

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

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

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

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

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

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

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

# ArrDivR()

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

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

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

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

value  A value used as the numerator of the division.

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

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

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

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

**See also:**
ArrXXX(), ArrAdd(), ArrConst(), ArrDiv(), ArrDot(), ArrMul(), ArrSub(), ArrSubR()
ArrSum(), BuffDiv(), BuffDivBy(), Len()

# ArrDot()

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

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

arr1  An array of reals or integers.

arr2  An array of reals or integers.

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

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

# ArrFFT()

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

**Func ArrFFT(dest[], mode%{, wnd%});**

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

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

wnd%  Used only in mode 0 to set 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. This argument does not exist before version 6.03.

Returns  The function returns 0 or a negative error code.

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

is composed of sine and cosine waves of frequencies: `0, 1/(n*t), 2/(n*t), 3/(n*t)` up to `(n/2)/(n*t)` or `1/(2*t)`.

### Display of phase in data views

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

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

### Mode 0: Window the data

This mode is used to apply a window of the selected type to the data array. See the power spectrum analysis for an explanation of windows. The selected data is multiplied by a window of maximum amplitude 1.0, minimum amplitude 0.0. The standard Hanning window causes a power loss in the result of a factor of 3/8. You can select a different window or supply your own window to taper the data using the array arithmetic commands. The Hanning (raised cosine) window used by default is a reasonable general purpose window.

### Mode 1: Forward FFT

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

| point | frequency | contents | point | frequency | contents |
|-------|-----------|----------|-------|-----------|----------|
| 0 | DC(0) | 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) as this sine wave has amplitude 0 at all sampled points.

### Mode 2: Inverse FFT

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

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

### Mode 3: dB and phase

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

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

There is no phase information for DC or for the point at 4/(n*t) because the phase for both of these is zero. If you want phase in degrees, multiply by 57.3968 (180 degrees/**pi**)]]. The log amplitude is calculated by taking the result of a forward FFT (same as mode 1 above) and forming:

$dB = 10.0$ 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* = $\texttt{Sqrt}(\cos^2+\sin^2)$ $\qquad$ 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 in terms of RMS power. The power is calculated by taking the result of a forward FFT (same as mode 1 above), and forming:

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

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

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

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

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

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

## ArrFilt()

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

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

$\texttt{dest[]}$ A real vector holding the data to filter. It is replaced by the filtered data.

$\texttt{coef[]}$ A real vector of filter coefficients. This is usually an odd number of data points long so that the result is not phase shifted. If you use an even number, the result is delayed by 1/2 a sample. Prior to Signal version 5.06, the result was advanced by 1/2 a sample. An even number of coefficients is used with a full differentiator.

Returns The function returns 0 if there was no error, or a negative error code.
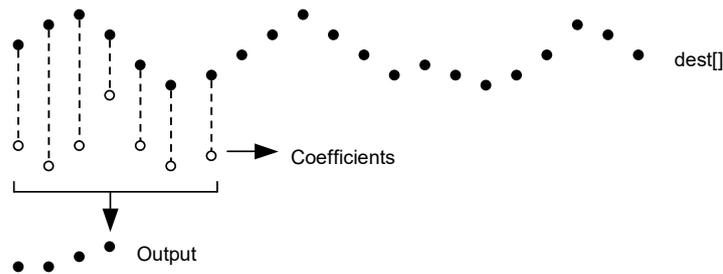
This diagram shows how the FIR filter works. Open circles represent filter coefficients, solid circles are the input and output waveforms. Each output point is generated by multiplying the waveform by the coefficients and summing the result. The coefficients are then moved one step to the right and the process repeats.

From this description, you can see that the filter coefficients (from right to left) are the *impulse response* of the filter. The impulse response is the output of a filter when the input signal is all zero except for one sample of unit amplitude. In the example above with seven coefficients, there is no time shift in the output. If the filter has an even number of coefficients, there is an output time shift of half a sample.

The filter operation is applied to every vector element. There is a problem at the start and end of the vector where some coefficients have no corresponding data element.
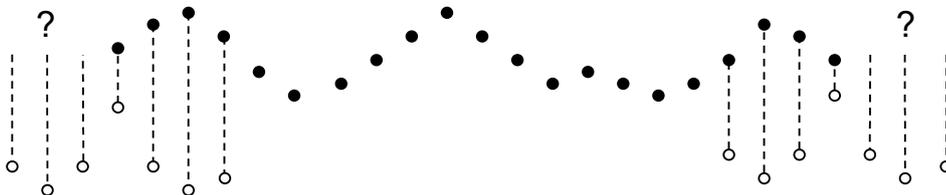


The simple solution is to take these missing points as copies of the first and last points. This is usually better that taking these points as 0. You should remember that the first and last $(nc+1)/2$ points are unreliable, where $nc$ is the number of coefficients.

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

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

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

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

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

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

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

**See also:**
ArrXXX(), ArrAdd(), ArrConst(), ArrDiv(), ArrDivR(), ArrDot(), ArrFFT(), ArrMul()
ArrSub(), ArrSubR(), ArrSum(), ChanSmooth(), Len()

# ArrHist()

This function generates a histogram from a data array. This function is designed for repeated use with the same destination histogram and different source data; it is up to the caller to zero the histogram array before the first use. You can use Min(), Max(), ArrSum() and ArrStats() to find suitable values for the start and binSz paramaters.

```
Func ArrHist(hist%[], data[], start, binSz{, &left%{, &right%}});
```

hist%  An integer array to hold the generated histogram. The elements of this array are incremented by 1 for each item in the data array that falls in one of the histogram bins. The array is not set to zero before incrementing the bins. The bin to increment for data element `i%` is given by: `(data[i%]-start)/binSz`.

data  An array of data values to be added into the histogram.

start  The value at the start of the first bin,

binSz  The width of each bin. This can be negative but must not be 0.

left%  Optional. This integer variable is incremented by the number of data items with a negative bin number (falling to the left of the histogram).

right%  Optional. This integer variable is incremented by the number of data items with a bin number greater than or equal to `Len(hist%[])` (falling to the right of the histogram).

Returns  The number of data items that fell inside the histogram.

**See also:**
`Min(), Max(), ArrSum(), ArrStats(), ArrConst()`

# ArrIntgl()

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

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

dest  An array or real or integer data.

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

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

**See also:**
`ArrXXX(), ArrDiff(), ChanIntgl(), Len()`

# ArrMapImage()

This command maps a matrix into a bitmap of the same size as the matrix and writes the bitmap to the clipboard or to an image file on disk. This command is commonly used to convert the output of the `ArrCWT()` command into an image. It can also be used to convert the result of any process that generates a 2D map into an image. The bitmap will be created as 32-bits per pixel unless this causes a memory problem in which case an 8 bit per pixel map with a maximum of 256 colours is generated. This function was added in Signal version 7.00.

You can create bitmaps of any width with this command (within the memory constraints of the system). However, Windows does not expect bitmaps wider or taller than 65535 pixels and many internal functions will fail if the bitmap is larger than this (some fail with widths more than 32767 pixels). As it is not unusual for the output of the `ArrCWT()` command to exceed this width, we provide mechanisms to restrict the width and height of the output. As an option, we allow you to generate larger maps, but these may fail with other programs. See the ArrCWT() example for code for adding an image to an XY view.

**Func ArrMapImage(arr[][], path${, flags%{, map%{, vMin{, vMax}}}});**

arr  A matrix holding data to be mapped into a bitmap. If this array is `arr[nx%][ny%]`, the bitmap will have `nx%` pixels in the horizontal direction and `ny%` pixels in the vertical direction. Unless the `flags%` argument is used to invert the vertical axis, `arr[][0]` sets the top row of pixels of the bitmap and `arr[][ny%-1]` sets the bottom row.

path$  This is either the file name on disk to write to or `"<CB>"` to place the resulting image on the clipboard as a bitmap. If this is a file name, the file extension determines the type of the image saved to disk if it is one of: `.bmp`, `.jpg`, `.jpeg`, `.gif`, `.png`, `.tif` or `.tiff`. If it is none of these, the file is saved in bitmap format. If the disk file already exists, it will be replaced.

flags% Additional controls to define how we perform the colour mapping. If omitted, this argument is 0. Bits 0-3 (values 0-7) are used to determine how we manage the situation when the bitmap would be wider than 65535 pixels. In this case, we limit the width to the maximum possible size and then map the source values to the pixels. When you use large bitmaps as a channel background (see the ChanImage() command), Signal takes special steps to render the image; other programs may not be so forgiving.

| Value | Meaning |
|---|---|
| 0 | Sub-sample the source array in the x direction to get the value to map to a pixel. |
| 1 | Take the mean value of the range of source values that map to each pixel. |
| 2 | Take the maximum value of the range of source values that map to each pixel. |
| 3 | Take the minimum value of the range of source values that map to each pixel. |
| 4-6 | Reserved, treated as 0. |
| 7 | Do not limit the width. This can generate an output image that may not be readable by some programs. |

You can also add additional flag values. Currently we define only one:

| Value | Meaning |
|---|---|
| 8 | Invert the bitmap so that arr[][0] generates the bottom row of pixels and not the top row. |

map% This defines the colour scale to be used. If this is not supplied, the default value is 1 (the Rainbow map). The available map numbers are:

| Map# | Name | Meaning |
|---|---|---|
| 0 | Monochrome | Runs linearly from black to white. |
| 1 | Rainbow | An approximation to the rainbow. |
| 2 | Thermal | Black to red to yellow to white to blue. |
| 3 | Geography | Black to blue to green to brown to grey to white. |
| 4 | Brightness | Runs from dark blue through mid-red through yellow to white. |
| 5 | The Blues | Black through shades of blue and on to white. |
| 6 | The Greens | Black through shades of green to white. |
| 7 | The Reds | Black through shades of red to white. |
| 8-9 | | Reserved for future use |

vMin The values vMin to vMax define the range of values to map onto the range of the colour map. vMin sets the data value in arr[][] to map onto the low end of the colour map. If you do not provide these values, the range of data values found in arr[][] is used; vMin will then be the lowest value found. You can use ArrRange() to find the data range.

vMax This sets the value to map to the high end of the colour map. If omitted, the highest value in arr[][] is used. There is no requirement for vMax to be greater than vMin. If vMax and vMin are the same, the display will be entirely set to the lowest colour in the map. Data values outside the range vMin to vMax are limited to the nearer end of the range.

Return 0 for success or a negative error code.

**See also:**
ArrRange(), ArrCWT(), ColourGet()

# ArrMul()

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

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

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

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

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

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

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

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

**See also:**
```
ArrXXX(), ArrAdd(), ArrConst(), ArrDiv(), ArrDivR(), ArrDot(), ArrSub(), ArrSubR()
ArrSum(), BuffMul(), BuffMulBy(), Len()
```

# ArrRange()

This command will scan an array of any number of dimensions and return the range of the data. This command was added at Signal version 7.00. There are two variants, one for Real values and one for integer:

**Real data**
```
Proc ArrRange(arr[]..., &vMin, &vMax);
```

arr     A real array with any number of dimensions to be searched for the minimum and maximum value.

vMin    A real variable that is returned holding the minimum value found in the array.

vMax    A real variable that is returned holding the maximum value found in the array.

**Integer data**
```
Proc ArrRange(arr%[]..., &vMin%, &vMax%);
```

arr%    An integer array with any number of dimensions to be searched for the minimum and maximum value.

vMin%   An integer variable that is returned holding the minimum value found in the array.

vMax%   An integer variable that is returned holding the maximum value found in the array.

**See also:**
```
Max(), Min(), MinMax()
```

# ArrSort()

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

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

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

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

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

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

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

### See also:
ArrXXX(), ArrAdd(), ArrConst(), ArrDiff(), ArrDiv(), ArrDivR(), ArrDot(), ArrMul(), ArrSubR()
ArrSum(), BuffSub(), BuffSubFrom(), Len()

# ArrSpline()

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

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

dest  A real or integer vector that holds the interpolated result. If the source array is a matrix, this is a real array.

source  A vector of y values. The associated x values are assumed to be the same as the index of each data point (0, 1, 2...). It is integer array if dest is integer or real if dest is real.

srcXY  A matrix of (x,y) source points. srcXY[][0] holds x values and srcXY[][1] holds y values. The x values must increase with the index, otherwise the return value is -1.

xInc  If present, this is the x value change between dest values. If omitted, we assume that the points in source or scrXY and dest span the same time range. There are no restrictions on xInc, it can even be negative (in which case the output is backwards relative to the input). If the source variant is in use, xInc is the ratio of the dest sample interval to the source sample interval.

xStart  If present, this is the x position of the first point of dest. If omitted, the first points of both arrays are at the same x position.

Returns  0 if all was OK or -1 if the scrXY variant is used and the x values do not increase monotonically.

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

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

### See also:
DrawMode()

# ArrStats()

This calculates the mean, variance, skewness and (excess) kurtosis of an array of real data. It can also be used with a matrix to calculate vectors of the mean, variance, skewness and kurtosis of each column of the input matrix. Use the ArrSum() command if you only require the mean and standard deviation.

```
Func ArrStats(data[], &mean, &v, &skew, &kur);
```

data    A vector of source data.

mean    A real variable returned holding the mean of the data.

v       A real variable returned holding the variance of the data.

skew    A real variable returned holding the skewness of the data.

kur     A real variable returned holding the kurtosis of the data.

Returns 0 if all was well or 1 if the variance of the data was 0.0 (in which case the skewness and kurtosis are also set to 0.0).

The second form of the command is provided as a convenience when the data is in a matrix with the data in the columns. When used in the second form, the vector must be at least the size of the first dimension of the matrix.

```
Func ArrStats(data[][], mean[], v[], skew[], kur[]);
```

data    A matrix with the data in the columns.

mean    A real vector returned holding the means of each column of data.

v       A vector returned holding the variances of each column of data.

skew    A vector returned holding the skewness of each column of data.

kur     A vector returned holding the kurtosis of each column of data.

Returns The number of columns for which the variance was 0 (in which case the skewness and kurtosis are also 0).

The first form of the command is equivalent to the following script (but runs many times faster):

```
Func ArrStats(arr[], &mean, &dVar, &skew, &kur)
var i%, n%;
mean := 0; dVar := 0; skew := 0; kur := 0;
ArrSum(arr, mean); 'Calculate the mean
n% := Len(arr);
var d, p;
for i% := 0 to n%-1 do
d := arr[i%]-mean; p := d*d;
dVar += p; p *= d;
skew += p;
kur += p*d;
next;
if (dVar > 0.0) then 'If not a degenerate case...
dVar /= (n%-1); 'calculate the variance
skew /= n%*dVar*sqrt(dVar);
kur /= n%*dVar*dVar;
kur -= 3.0;
return 0;
else
return 1;
endif;
end;
```

The second form of the command is equivalent to:

```
var i%, return% := 0;
for i% := 0 to Len(data[0][])-1 do
return% += ArrStats(data[][i%], mean[i%], v[i%], skew[i%], kur[i%]);
next;
```

**See also:**
ArrSum()

# ArrSub()

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

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

dest    A real or integer array that holds the result.

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

value   A real or integer value.

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

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

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

**See also:**
```
ArrXXX(), ArrAdd(), ArrConst(), ArrDiff(), ArrDiv(), ArrDivR(), ArrDot(), ArrMul(),
ArrSubR()
ArrSum(), BuffSub(), BuffSubFrom(), Len()
```

# ArrSubR()

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

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

dest    A real or integer array.

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

value   A real or integer value.

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

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

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

**See also:**
```
ArrXXX(), ArrSub(), ArrSum(), BuffSub(), BuffSubFrom(), Len()
```

# ArrSum()

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

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

arr A real or integer vector or matrix to process.

mean If present it is returned holding the mean of the values in the array. The mean is the sum of the values divided by the number of array elements. If arr is an m by n matrix, mean must be a vector of at least n elements and is returned holding the mean of each column of arr.

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

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

**See also:**
ArrXXX(), 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|const 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 also be an array.

c If this is present, the function uses $s/c$ for the calculation. If c is an array its dimensions must match those of s.

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

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

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

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

**See also:**
Cos(), Cosh(), Exp(), Frac(), Ln(), Log(), Max(), Min(), Pow(), Rand()
Round(), Sin(), Sinh(), Sqrt(), Tan()

# B

## BetaI()

This function computes both the Beta function and the Incomplete Beta function. The Beta function is defined as:

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

The incomplete beta function is defined as:

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

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

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

a, b     These values must be greater than zero.

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

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

**See also:**
GammaP(), GammaQ(), Binomial Distribution, Student's T Distribution, F-Distribution

## Binomial Distribution

The Binomial distribution is used when you have repeated trials and for each trial an event has a known probability *p* of occurrence. The examples below show you how to calculate the probabilities that you will get k% or fewer events in n% trials and k% or more in n% trials.

```
'p       The probability per trial of the event
'k%      A number of events in the range 0 to n%
'n%      The number of trials (greater than 0)
'Return the probability that an event occurs k% or fewer times in n% trials.
Func BinomialLE(p, k%, n%)
if k% >= n% then return 1.0 endif;
if k% < 0 then return 0.0 endif;
return BetaI(n%-k%, 1+k%, 1-p);
end;

'p       The probability per trial of the event
'k%      A number of events in the range 0 to n%
'n%      The number of trials (greater than 0)
'Return the probability that an event occurs k% or more times in n% trials.
Func BinomialGE(p, k%, n%)
if (k% <= 0) then return 1.0 endif;
if (k% > n%) then return 0.0 endif;
return BetaI(k%, n%-k%+1, p);
end;
```

The BimonialC() function will evaluate binomial coefficients for you, avoiding arithmetic overflow as much as is possible.

**See also:**
BetaI(), BinomialC(), Student's T Distribution, F-Distribution

# Student's T Distribution
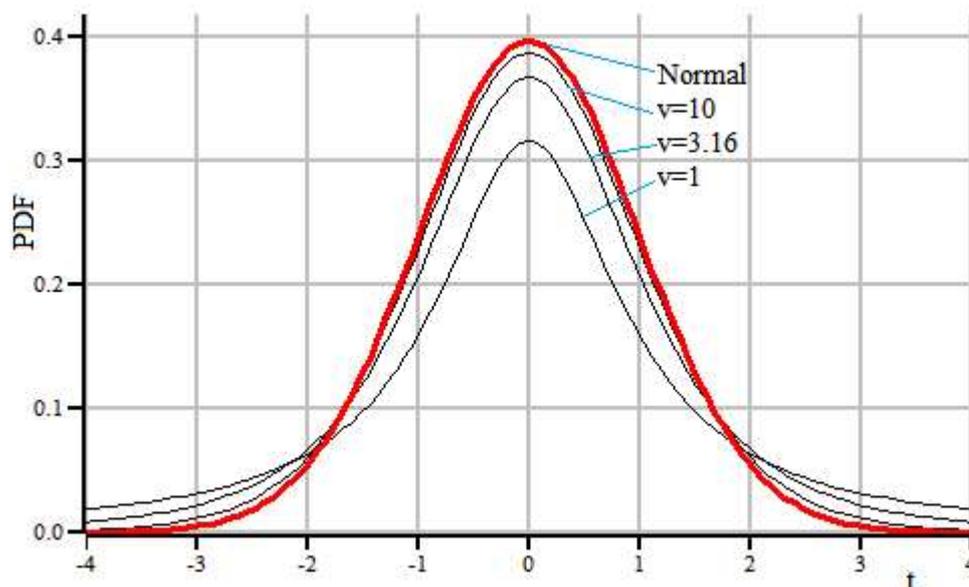


*Students's T distribution for various degrees of freedom compared to Normal distribution*

The Student's T distribution is a symmetric probability density function (PDF) used when asking questions about the mean of a sample taken from a normally distributed population when the number of samples is small. It is characterised by the number of degrees of freedom (which is usually the number of samples used to estimate the mean minus 1), and becomes indistinguishable from the Normal distribution when the degrees of freedom reaches a hundred or so. If you want to graph the function, you can do so with this script expression:

```
const pi := 3.141592653589793;
func StudentsTDist(t, v)
return pow(1+t*t/v, -(v+1)/2)*Exp(LnGamma((v+1)/2) - LnGamma(v/2))/Sqrt(v*pi);
end;
```

The t value is a statistic that relates to how different two means are and can be positive or negative. Exactly how it is calculated depends on the situation; we give some examples later in this section, but the simplest case is where you have a normally distributed population and you take single sample of `n%` items with a sample mean `Sm` and sample standard deviation `Ssd`. If the true mean of the population is `mean`, the `t` statistic is:

```
t := (Sm-mean)*Sqrt(n%)/Ssd;
```

When asking questions about how likely one would be to observe a mean value that is different from or greater than or less than some value, the more useful functions are the cumulative probability distribution functions (in the range 0 to 1 as `t` varies from minus infinity to plus infinity), and the complement of this (1 minus it). These are usefully calculated from the incomplete beta function, as follows:

```
't    The t statistic
'v    The number of degrees of freedom (>0)
Func StudentCDF(t, v)
var tt := t*t;
var z := BetaI(0.5*v, 0.5, v/(v+tt))*0.5;
return t < 0 ? z : 1-z;
end;
```

```
'This is 1.0-StudentCDF(t, v);
Func OneMinusStudentCDF(t, v)
var tt := t*t;
var z := BetaI(0.5*v, 0.5, v/(v+tt))*0.5;
return t < 0 ? 1-z : z;
end;
```

The `StudentCDF()` function returns the probability that one can reject the Null hypothesis that the means are not different (or not greater than or not less than some value). The value we usually want is 1-`StudentCDF()`, which we calculate separately to preserve the accuracy for small values. In statistical tests, the probability level we are interested in, for example 0.05 (5%) or 0.01 (1%) is often referred to as *alpha*. `StudentCDF()` can be compared with 1-alpha, which is called the confidence level (95% or 99%), `OneMinusStudentCDF()` returns the probability

that the hypothesis being tested would occur by chance and so can be rejected. That is, you would reject the hypothesis if `OneMinusStudentCDF()` > alpha.

### One-tail and two-tail tests

If you are asking the question, "are these two mean values different", you want to know how likely is a sampled mean value to occur by chance. If you want to be 99% certain that your sample differs, your t value needs to differ from 0 by a sufficient value that the cumulative probability density has reached a value such that 99% of the PDF lies at lesser values. As the PDF is symmetric about 0, this happens when 1% of the PDF is excluded, which happens when 0.5% is above some critical *t* value *Tcrit* and 0.5% is below -*Tcrit*. In this case your alpha is 0.01 (or 1%) and the value you would test for is alpha/2. This is a two-tail test as you care what happens at both ends of the probability distribution.
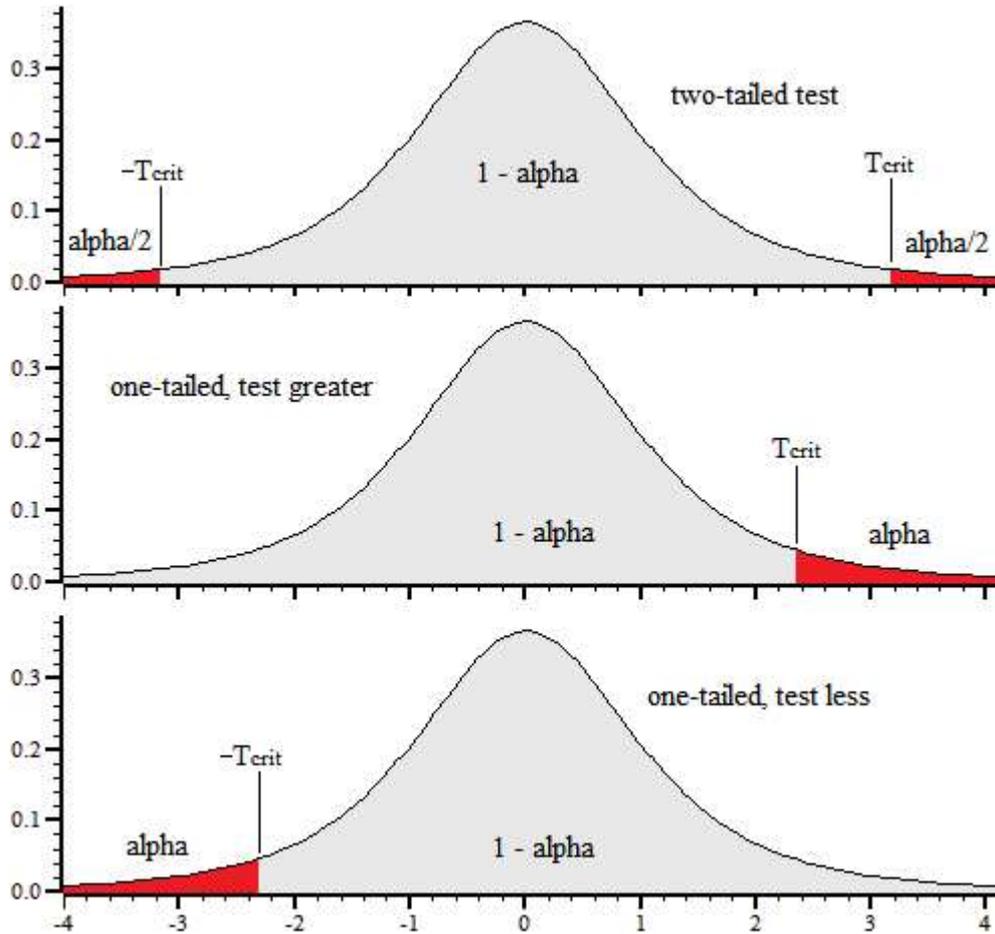


If you are asking the question, "is this mean greater than that mean", you are only looking at one end of the distribution and the value you would test against is alpha. If you want to be 99% certain, *Tcrit* is now the level at which 99% of the distribution lies below, your alpha value is 0.01 and this is the value to test for. Due to the symmetry of the distribution, to test that "this mean is less than that mean" you need a negative t value; the probability to test for is the same as for a positive t value. A one-tail test should be used with caution.

### Inverse of the distribution

The functions above calculate alpha given a t statistic and the number of degrees of freedom. If you want to specify alpha and calculate the t statistic value that corresponds to this, you need the inverse function. The following script code will do this for you:

```
'Function used by ZeroRoot() to locate the t value with the required probability
var gProb, gV;        'probability and degrees of freedom
func TZeroRoot(t)
return gProb - OneMinusStudentCDF(t, gV);
end;

'TValueFor: This calculates the t-value that you need to reject the NULL hypothesis
'given an alpha, the number of degrees of freedom and if you want a one sided test.
'alpha The desired probability level (typically 0.05)
'v     Degrees of freedom (> 0.0)
'os%   0=two sided, non-zero for 1 sided
func TValueFor(alpha, v, os%)
var t;
gProb := os% ? alpha : alpha*0.5;
gV := v;
ZeroFind(t, TZeroRoot, -20, 20);
return t;
end;
```

### Examples

The following example code follows examples given for the Boost C++ libraries, which in turn follow NIST/SEMATECH e-Handbook of Statistical Methods, http://www.itl.nist.gov/div898/handbook/ (NIST is an agency of the US Commerce Department).

### Confidence range of the mean

You have `Sn%` sample values taken from a normally distributed population with mean `Sm` and standard deviation `Ssd` and you want to know in what range of values can you be confident that you have bracketed the true population mean. This is plainly a two-tail test as we care about values in both directions. If you want to be certain at the `alpha` level, the `Tcrit` value is given by: `TValueFor(alpha/2, Sn%-1, 0)` and the range is from `Sm-Tcrit*Ssd/Sqrt(Sn%)` to `Sm+Tcrit*Ssd/Sqrt(Sn%)`. See here for details.

### Test a sample mean for difference from a known mean

This test comes up if you already know what the mean should be (for example in quality control) and you want to know if there is a change. It can also come up in a paired test where you measure a value before and after a treatment on a set of items and your data set is the difference between before and after for each item, so you are asking is there a difference from zero. The following routine does this calculation:

```
'testing sample mean for difference from a "true" mean.
'n%     Number of items in the sample
'Sm     Sample mean
'Ssd    Sample standard deviation
'mean   The "true" mean
'os%    -1 means one-sided test for below mean, +1 means one-sided test for
'       above the mean, 0 means two-sided test for different
'Return the probability that the Null hypothesis (that the sample mean is
'       not different, greater or less than the true mean).
Func TSingleSample(n%, Sm, Ssd, mean, os%)
var t := (Sm - mean)* Sqrt(n%)/Ssd;
PrintLog("\nStudent's t test for a single sample\n");
PrintLog("Number of observations    = %D\n", n%);
PrintLog("Sample mean               = %g\n", Sm);
PrintLog("Sample Standard Deviation = %g\n", Ssd);
PrintLog("Expected True mean        = %g\n", mean);
PrintLog("T Statistic               = %g\n", t);
PrintLog("Degrees of freedom        = %d\n", n%-1);
if (os% = 0) then t := abs(t) endif;
if (os% < 0) then t := -t endif;
var prob := OneMinusStudentCDF(t, n%-1);  'Single tailed value
if (os% = 0) then prob *= 2.0 endif;
PrintLog("Probability due to chance  = %g\n", prob);
return prob;
end;
```

In this test we have a single sample of `n%` items with mean `Sm` (Sample mean) and standard deviation `Ssd` (Sample standard deviation). The question is: is the mean different (two-tail) or greater or less (one-tail) that the known mean. The return value is the probability that the Null hypothesis can be rejected and is the value to compare against the alpha value for your test. You may choose to omit the `PrintLog()` lines. To use this function, for

example using data from the NIST site: 3 observations with mean of 37.8, standard deviation of 0.96437 and expected true mean of 38.9:

```
TSingleSample(3, 37.8, 0.96437, 38.9, -1); 'Test for mean < 38.9
TSingleSample(3, 37.8, 0.96437, 38.9, 0);  'Test for mean <> 38.9
TSingleSample(3, 37.8, 0.96437, 38.9, 1);  'Test for mean > 38.9
```

The output is (omitting repeated lines for second and third cases):

```
Student's t test for a single sample
Number of observations    = 3
Sample mean               = 37.8
Sample Standard Deviation = 0.96437
Expected True mean        = 38.9
T Statistic               = -1.97565
Degrees of freedom        = 2
Probability due to chance = 0.093429  (Hypothesis: mean < 38.9)
...
Probability due to chance = 0.186858  (Hypothesis: mean <> 38.9)
...
Probability due to chance = 0.906571  (Hypothesis: mean > 38.9)
```

With an alpha of 0.05 (5%), all the hypotheses are rejected. However, if the alpha level was set at 0.1 (10%), then the hypothesis that the mean is less than 38.9 is not rejected.

### Test two samples with equal variance for difference of means

In this case we have a first sample with $n1\%$ items with mean $S1m$ and standard deviation $S2sd$ and a second sample with $n2\%$ items and mean $S2m$ and standard deviation $S2sd$. Again, we want to ask if the means are different, or if the first mean is less than or greater than the second. The code to do this is (you can delete the `PrintLog()` lines if they are not wanted):

```
'ni%    The number in each set of samples
'Sim    The mean of sample i
'Sisd   The standard deviation of sample i
'os%    0 for two sided. 1 for sm1 > sm2, -1 for sm1 < sm2
'Return the probability that the Null hypothesis (that the sample means
'       are not different, Sm1 greater or less than Sm2).
Func TTwoSamplesSameVar(n1%, S1m, S1sd, n2%, S2m, S2sd, os%)
'Calculate the combined standard devaition of both samples
var S12sd := Sqrt(((n1%-1)*S1sd*S1sd + (n2%-1)*S2sd*S2sd)/(n1%+n2%-2));
var t := (S1m - S2m)/(S12sd * Sqrt(1.0/n1% + 1.0/n2%));
PrintLog("\nStudent's t test for two samples, same variance\n");
PrintLog("Number of observations    = %8d %8d\n", n1%, n2%);
PrintLog("Sample means              = %8g %8g\n", S1m, S2m);
PrintLog("Sample Standard Deviation = %8g %8g\n", S1sd, S2sd);
PrintLog("T Statistic               = %g\n", t);
PrintLog("Degrees of freedom        = %d\n", n1%+n2%-2);
if (os% = 0) then t := abs(t) endif;
if (os% < 0) then t := -t endif;
var prob := OneMinusStudentCDF(t, n1%+n2%-2);  'One-tail value
if (os% = 0) then prob *= 2.0 endif;
PrintLog("Probability due to chance = %8e\n", prob);
return prob;
end;
```

As the two samples have the same variance we can form a pooled standard deviation by combining the two standard deviations, which is what the calculation for $S12sd$ does. The number of degrees of freedom is just the sum of the degrees of freedom of the two data sets The calculation of the t statistic is then very similar to the previous single sample test, just weighting by the two sample sizes. As an example, we consider the following two data sets:

| Data set | Observations | Mean | Sd |
|---|---|---|---|
| 1 | 249 | 20.1446 | 6.4147 |
| 2 | 79 | 30.481 | 6.10771 |

The code to test each possible hypothesis is:

```
TTwoSamplesSameVar(249, 20.14458, 6.41470, 79, 30.48101, 6.10771, -1); 'mean 1 < mean 2
TTwoSamplesSameVar(249, 20.14458, 6.41470, 79, 30.48101, 6.10771,  0); 'mean 1 <> mean 2
TTwoSamplesSameVar(249, 20.14458, 6.41470, 79, 30.48101, 6.10771,  1); 'mean 1 > mean 2
```

The output (omitting duplicated lines is):

```
Student's t test for two samples, same variance
Number of observations    =     249      79
Sample means              =  20.1446   30.481
Sample Standard Deviation =   6.4147  6.10771
T Statistic               = -12.6206
Degrees of freedom        = 326
Probability due to chance = 2.636609e-030 (Hypothesis: mean1 < mean 2)
...
Probability due to chance = 5.273218e-030 (Hypothesis: mean1 <> mean 2)
...
Probability due to chance = 1.000000e+000 (Hypothesis: mean 1 > mean 2)
```

This says that the hypotheses that mean1 is less than mean2 or that mean1 is different from mean2 cannot be rejected. The hypothesis that mean1 is greater than mean2 is rejected.

### Test two samples with non-equal variance

Finally we have the code for two samples with non-equal variance. This time we cannot pool the standard deviations, and we also have a problem with the degrees of freedom. To calculate this, the Welch-Scatterthwaite approximation is used. Note that the approximation improves as the number of degrees of freedom increases. As ever, you can delete all the `PrintLog()` lines if they are not required:

```
Func TTwoSamplesDiffVar(n1%, S1m, S1sd, n2%, S2m, S2sd, os%)
var s1 := S1sd*S1sd/n1%;
var s2 := S2sd*S2sd/n2%;
'Calculate the t value
var t := (S1m - S2m)/Sqrt(s1 + s2);
'Calculate the combined degrees of freedom using the Welch-Scatterthwaite approximation.
var v := (s1+s2)*(S1+s2)/(s1*s1/(n1%-1) + s2*s2/(n2%-1));
PrintLog("\nStudent's t test for two samples, different variance\n");
PrintLog("Number of observations    = %8d %8d\n", n1%, n2%);
PrintLog("Sample means              = %8g %8g\n", S1m, S2m);
PrintLog("Sample Standard Deviation = %8g %8g\n", S1sd, S2sd);
PrintLog("T Statistic               = %g\n", t);
PrintLog("Degrees of freedom        = %g\n", v);
if (os% = 0) then t := abs(t) endif;
if (os% < 0) then t := -t endif;
var prob := OneMinusStudentCDF(t, v);   'One-tail value
if (os% = 0) then prob *= 2.0 endif;    'Convert to two-tail value
PrintLog("Probability due to chance  = %8e\n", prob);
return prob;
end;
```

If we use the same input data as for the previous example, we have:

```
TTwoSamplesDiffVar(249, 20.14458, 6.41470, 79, 30.48101, 6.10771, 0); 'mean 1 < mean 2
TTwoSamplesDiffVar(249, 20.14458, 6.41470, 79, 30.48101, 6.10771, 0); 'mean 1 <> mean 2
TTwoSamplesDiffVar(249, 20.14458, 6.41470, 79, 30.48101, 6.10771, 0); 'mean 1 > mean 2
```

The output (omitting repeated lines) is:

```
Student's t test for two samples, different variance
Number of observations    =     249       79
Sample means              =  20.1446   30.481
Sample Standard Deviation =   6.4147  6.10771
T Statistic               = -12.9463
Degrees of freedom        = 136.875
Probability due to chance = 7.854523e-026 (Hypothesis: mean1 < mean 2)
...
Probability due to chance = 1.570905e-025 (Hypothesis: mean1 <> mean 2)
...
Probability due to chance = 1.000000e+000 (Hypothesis: mean1 > mean 2)
```

In this case, although the probability that this occurred by chance is increased by several orders of magnitude compared to the equal variance situation, the conclusions are the same.

### See also:

`BetaI()`, Binomial Distribution, F-Distribution
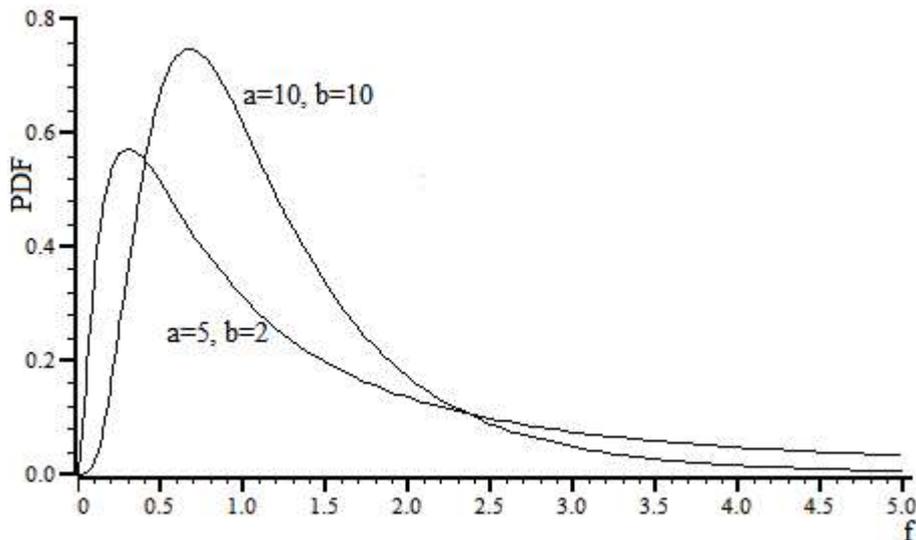
# F-Distribution



*F Distribution Probability Density Function*

The F Distribution probability density functions shows the likelihood of particular values of the F statistic occurring. The F statistic calculation depends on the situation; in a simple case it is the ratio of two variances. The F distribution can be calculated and displayed by the script code:

```
'The derivative of the BetaI function, used to calculate the Prob density of
'the F-Distribution. Maths as suggested by the Boost library.
Func BetaIDerivative(a, b, x)
if (a<=0.0) or (b<=0.0) or (x<0) or (x > 1.0) then return -1 endif;
if (x=0.0) or (x=1.0) then return 0 endif;
return  exp((b-1)*ln(1-x)+(a-1)*ln(x))/BetaI(a,b);
end;


'The Probability Density function for the F-Distribution. The use of two
'methods keeps the third argument to the derivative function away from 1.
Func FDistribution(a, b, x)
var z := b + a* x;
var y := a * b / (z * z);
if (a*x > b) then
    return y * BetaIDerivative(b/2, a/2, b/z);
else
    return y * BetaIDerivative(a/2, b/2, a*x/z);
endif;
end;


'Draw the F distribution from x=0 to range with a and b degrees of freedom
Func ShowFDist(a, b, range)
const n% := 100;          'points after 0
const np% := n%+1;        'total points to plot
var rv% := SetResult(np%, range/n%, 0, "F distibution", "f", "PDF");
DrawMode(-1,13);          'cubic spline mode
var i%, x;
for i% := 0 to n% do
    x := i%*(range/n%);
    [i%] := FDistribution(a, b, x);
next;
WindowVisible(1);
return rv%;
end;

ShowFDist(5, 2, 5);  'Draw an example
```

Unlike the Student's T distribution, the F distribution is asymmetric. For us, a more interesting and useful function is the Cumulative Distribution Function (CDF) of the F distribution. For statistical use, the complement of the CDF (1-CDF) is usually the most useful. The CDF and its complement can be calculated with:

```
'The CDF for the F-Distribution.
func FDistCDF(a, b, x)
var ax := a*x;
var z := b+ax;
return BetaI(a/2, b/2, ax/z);
end;

'We usually want the complement of the CDF
func OneMinusFDistCDF(a, b, x)
var ax := a*x;
var z := b+ax;
return BetaI(b/2, a/2, b/z);
end;
```

The value of the `FDistCDF()` function increases from 0 to 1 as x (the F statistic) varies between 0 and infinity.
The value of `OneMinusFDistCDF()` decreases from 1 to 0 over the same range.


### Inverse of the F distribution CDF complement

When asking questions such as, "What F value does 5% of the F-Distribution lie above?", we find the inverse of
the complement of the F distribution CDF at the 5% level. This can be calculated by the `InvFCDFComp()`
function:

```
var gA, gB, gAlpha; 'variables used by InvFCDFComp()

'Helper function for InvFCDFComp(), used by ZeroFind()
Func ZeroFRoot(x)
return OneMinusFDistCDF(gA, gB, x)-gAlpha;
end;

'Find the inverse of the complement of the cumulative F Distribution
'alpha The probability level
'a,b,  The degrees of freedom of the two distributions
Func InvFCDFComp(alpha, a, b)
var x;
gA := a; gB := b; gAlpha := alpha;
ZeroFind(x, ZeroFRoot, 0, 20);
return x;
end;
```


### F-Test for equality of two standard deviations

The simplest use of the F-test is to determine if the standard deviation of two populations are the same. See this
NIST handbook section for an explanation. The `FTestSameSd()` function calculates the one-tailed probability
value and also returns the critical F statistic values for both the one-tailed test (alpha) and also for a two-tailed test
(alpha/2). For a two-tailed test, both the critical levels at `alpha/2` and `1-alpha/2` apply. For the one-tailed test,
only one of the levels at `alpha` or `1-alpha` apply. In this case, the F statistic is just the ratio of the squares of the
two standard deviations and the number of degrees of freedom is the number of samples of each distribution minus
1:

```
'F-test for equal standard deviations
'n1%   Number of items in sample 1
'sd1   Standard deviation of sample 1
'n2%   Number of items in sample 2
'sd2   Standard deviation of sample 2
'alpha The probability level to calculate FCrit
Func FTestSameSd(n1%, sd1, n2%, sd2, alpha)
var f := sd1/sd2;      'The test statistic is the ratio...
f *= f;                '...of the two sd's squared
var prob := OneMinusFDistCDF(n1%-1, n2%-1, f);
PrintLog("\nF-Test for same standard deviations\n");
PrintLog("Number of samples  = %8d %8d\n", n1%, n2%);
PrintLog("Standard deviations = %8g %8g\n", sd1, sd2);
PrintLog("Test statistic (F)  = %8g\n", f);
PrintLog("Prob (one-tailed)  = %8g\n", prob);
PrintLog("Upper critical level at alpha/2 = %8g\n", InvFCDFComp(alpha/2, n1%-1, n2%-1));
PrintLog("Upper critical level at alpha  = %8g\n", InvFCDFComp(alpha, n1%-1, n2%-1));
PrintLog("Lower critical level at alpha  = %8g\n", InvFCDFComp(1-alpha, n1%-1, n2%-1));
PrintLog("Lower critical level at alpha/2 = %8g\n", InvFCDFComp(1-alpha/2, n1%-1, n2%-1));
return prob;
end;
```

With the following data:

| Sample | Items | sd |
|--------|-------|--------|
| 1 | 11 | 4.9082 |
| 2 | 9 | 2.5874 |

We use the script to test at an alpha level of 0.05:

```
FTestSameSd(11, 4.9082, 9, 2.5874, 0.05);
```

and the output is:

```
F-Test for same standard deviations
Number of samples   =        11         9
Standard deviations =    4.9082    2.5874
Test statistic (F)  =  3.59847
Prob (one-tailed)   = 0.0411439
Upper critical level at alpha/2 =  4.29513 (two-tailed test upper limit)
Upper critical level at alpha   =  3.34716 (sd1 > sd2 one-tailed limit)
Lower critical level at alpha   = 0.325557 (sd1 < sd2 one-tailed limit)
Lower critical level at alpha/2 = 0.259411 (two-tailed test lower limit)
```
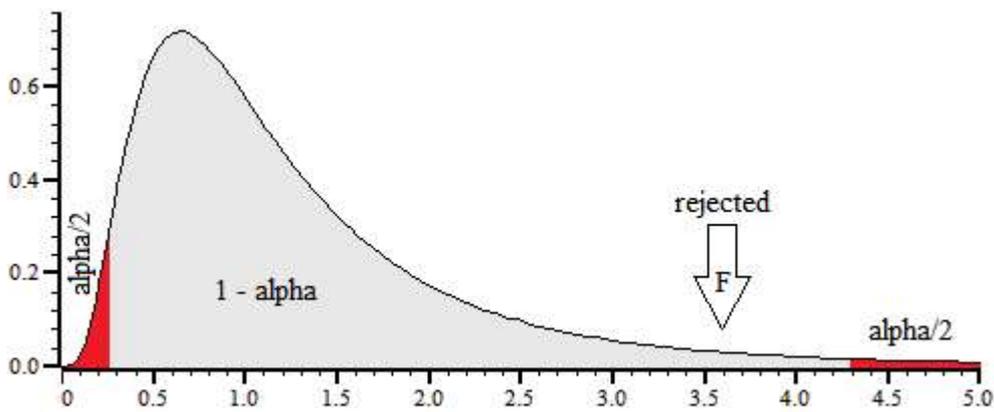


*Two-tailed F-test*

Given this result, for a two-tailed test the hypothesis that the standard deviations are different is rejected at the 5% level as the F statistic does not fall outside the range 0.259411 to 4.29513. The probability value returned (that the result occurred by chance) is for comparison with alpha for a one-tailed test; for a two-tailed test compare it with alpha/2 or double it for comparison with alpha .
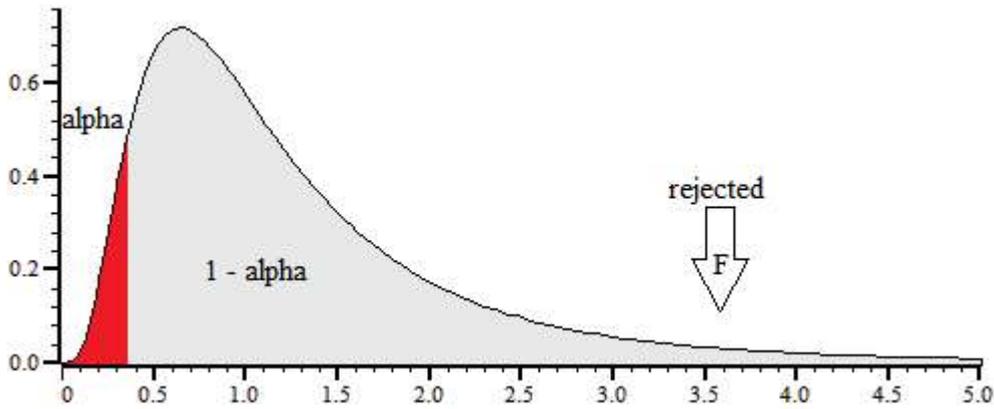


*One-tailed F test for less*

For a one-tailed test that the standard deviation is less, the result is a clear rejection (as you would expect as the value was greater).

*One-tailed F-test for greater*
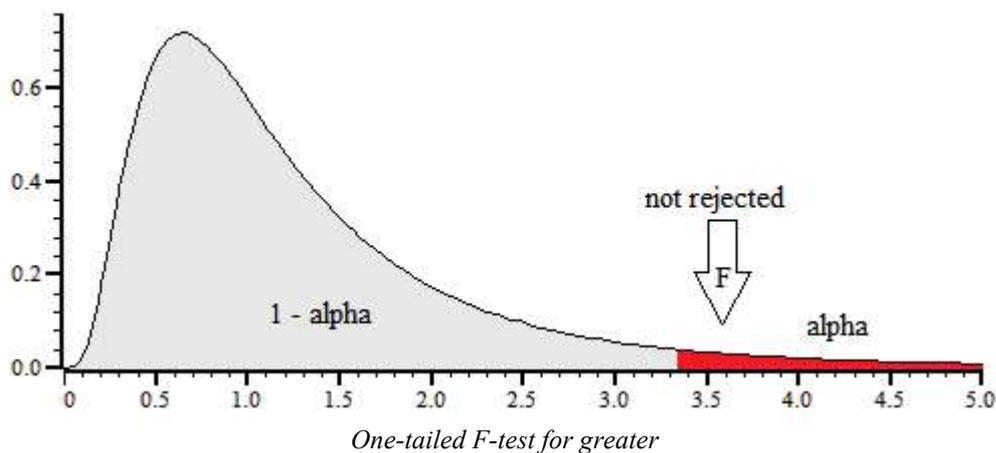
For a one-tailed test, the probability value is less than alpha, and we can see that the hypothesis that sample 1 standard deviation is greater than sample 2 is not rejected at the 5% level.

**See also:**
BetaI(), Binomial Distribution, Student's T Distribution

# BinError()

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

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

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

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

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

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

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

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

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

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

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

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

# BinomialC

This function calculates the Binomial coefficient `nCk`, which is the number of different ways to choose `k` items from n, which is `n!/(k!*(n-k)!)`. As `n` factorial grows rapidly as n increases, this can be difficult to compute for even a modest value of n.

```
Func BinomialC(n%, k%)
```

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

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

Returns  The binomial coefficient. The return is integral, but is returned as a real value as it can be very large, for example `BinomialC(34,17)` is 23336062200,which exceeds 32-bit integer range. Floating point numbers can represent integers exactly up to some 15 digits, after which accuracy cannot be maintained. If n% exceeds 1029 the result can be infinity, which means it is greater than `1.7977e+308` . If you really need results for very large n%, you can get the logarithm of the result with `LnGamma(n%)-LnGamma(k%)-LnGamma(n%-k%)`.

**See also:**
```
BetaI(), GammaP() and GammaQ(), LnGamma()
```

# BinSize()

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

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

chan%   The channel number (1 to n) for which to return information, which must exist.

nSize   If this is present it sets a new x axis resolution in a memory view. For log-binned data this value must be greater than one.

Returns  The value returned depends on the channel type:

| | |
|---|---|
| Waveform | This is the x axis interval between points on the channel. For sampled data this is the sample interval. For log-binned data this is the ratio of each bin width to the next and is always greater than one. |
| Marker | The underlying x axis resolution of the channel. |
| Real marker | The underlying x axis resolution of the channel. |
| Idealised trace | A dummy value of 1.0 is returned. |

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

# BinToX()

This function converts between bin numbers and x axis values for a channel in the current view. For waveform channels a bin is a waveform point, for marker-type channels it is a marker item and for idealised trace channels it is a trace segment.

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

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

bin     A bin number in the view, bin numbers run from zero upwards. You can give a non-integer bin number without error. If you give a bin number outside the range of the view, it will be limited to the range of the view.

Returns  The returned value is the equivalent x axis position for the relevant waveform sample, marker item or the start time of the idealised trace segment.

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

# BinZero()

This function returns the x axis position for the first bin in the frame on the given channel for a memory view, in a file view it returns the X axis position for the start of the frame. In a memory view you can set this value, from version 7.06 onwards you can also set the frame start time (for all channels) in a file view, .

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

chan%    The memory view channel (1 to n) for which to return information. If the function is being used on a file view then `chan%` should be zero.

offset   If this is provided it sets a new x axis position for the first data point for the specified channels in a memory view, in a file view it sets the overall frame start time.

Returns  The returned value is the initial x axis position of the start of the channel data for a memory view, or the initial frame start time for a file view.

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

# BRead()

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

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

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

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

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

# BReadSize()

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

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

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

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

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

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

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

# BRWEndian()

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

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

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

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

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

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

# BSeek()

This function moves and reports the current position in a file opened by `FileOpen()` with a `type%` code of 9. The next binary read or write operation to the file starts from the position returned by this function.

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

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

`rel%`  This determines to what the new position is relative:

    0   Relative to the start of the file (same as omitting the argument).

    1   Relative to the current position in the file.

    2   Relative to the end of the file.

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

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

# Buffer commands

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

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

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

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

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

# BuffAcc()

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

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

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

Returns  Zero or a negative error code.

**See also:**
```
ArrAdd(), BuffUnAcc(), BuffXXX()
```

# BuffAdd()

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

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

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

Returns  Zero or a negative error code.

**See also:**
```
ArrAdd(), BuffXXX()
```

# BuffAddTo()

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

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

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

Returns  Zero or a negative error code.

**See also:**
```
ArrAdd(), BuffXXX()
```

# BuffClear()

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

```
Func BuffClear();
```

Returns  Zero or a negative error code.

**See also:**
```
ArrConst(), BuffXXX()
```

# BuffCopy()

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

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

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

Returns  Zero or a negative error code.

**See also:**
`ArrConst(), BuffXXX()`

# BuffCopyTo()

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

`Func BuffCopyTo({frame%});`

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

Returns  Zero or a negative error code.

**See also:**
`ArrConst(), BuffXXX()`

# BuffDiv()

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

`Func BuffDiv({frame%});`

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

Returns  Zero or a negative error code.

**See also:**
`ArrDiv(), BuffXXX()`

# BuffDivBy()

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

`Func BuffDivBy({frame%});`

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

Returns  Zero or a negative error code.

**See also:**
`ArrDiv(), BuffXXX()`

# BuffExchange()

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

`Func BuffExchange({frame%});`

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

Returns  Zero or a negative error code.

**See also:**
`ArrConst(), BuffXXX()`

# BuffMul()

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

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

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

Returns  Zero or a negative error code.

**See also:**
`ArrMul(), BuffXXX()`

# BuffMulBy()

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

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

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

Returns  Zero or a negative error code.

**See also:**
`ArrMul(), BuffXXX()`

# BuffSub()

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

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

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

Returns  Zero or a negative error code.

**See also:**
`ArrSub(), BuffXXX()`

# BuffSubFrom()

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

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

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

Returns  Zero or a negative error code.

**See also:**
`ArrSub(), BuffXXX()`

# BuffUnAcc()

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

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

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

Returns Zero or a negative error code.

**See also:**
ArrSub(), BuffAcc(), BuffXXX()

# 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(const arg1 {,const arg2 {,...}});
```

arg Arguments may be of any type, including arrays. Signal fills a block of memory equal in size to the combined size of the arguments with the data held in the arguments and copies it to the file. An integer uses 4 bytes and a real uses 8 bytes. A string is written in UFT-8 format with a zero byte to mark the end. Use `BWriteSize()` to write a fixed number of bytes. The arguments are not modified, so `const` arrays can be written.

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

**See also:**
FileOpen(), BRead(), BReadSize(), BSeek(), BRWEndian(), BWriteSize()

# BWriteSize()

This function writes variables or arrays as binary into a file opened by `FileOpen()` with a `type%` code of 9. It writes 8, 16 and 32-bit integers and 32 and 64-bit reals and strings. It allows you to write formats other than the 32-bit integer and 64-bit real used internally by the script and to write variable-length strings into fixed-size fields in a binary file.

```
Func BWriteSize(size%, const arg1 {,const 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.

**Unicode**
If you write a string, it is converted to UTF-8 first. Beware that this means that the number of bytes needed to hold the string may be more than the number of characters reported by `Len()`. There is an example of getting the equivalent UTF-8 string length in the description of `Len()`.

**See also:**
FileOpen(), BRead(), BReadSize(), BSeek(), BRWEndian(), BWrite(), Example to write a bitmap

# Using BWriteSize() to save a bitmap

This example uses BWriteSize() to save a bitmap image in an integer matrix as a .bmp file. Each element of the bitmap array is an integer with bits 0-7 holding the blue colour, bits 8-15 holding the green and bits 16-23 holding the red and the rest of the data bits set to 0 (0x00rrggbb). If you want an Alpha channel (using bits 24-31 to specify opaqueness, with 0 for transparent and 0xff for opaque, 0xaarrggbb), add 2 to flags%. You can choose to have a bottom to top or top to bottom bitmap.

```
'Write a .bmp file based on rgb colours in integer array
'bmpPath$ Path to the file to use. Ideally the name should end in .bmp
'bmp%     A matrix bmp%[nx][ny] with the nx=columns, ny is rows
'flags%   If bit 0 is set, bmp%[0][0] is top left, else bottom left
'         If bit 1 is set, use RGBA coding, else RGB
'return   0 if OK or a negative error code
func WriteBmp(bmpPath$, const bmp%[][], flags% := 0)
var xpix% := Len(bmp%[][0]);          ' pixels in the x direction
var ypix% := Len(bmp%[0][]);          ' pixels in the y direction

var binf% := FileOpen(bmpPath$, 9, 1); ' open binary output file
if (binf% < 0) then return binf% endif;

'Write out the bitmap header. This is 14 bytes long.
BWriteSize(2, "BM");                  ' all bitmaps files start 'BM'
BWriteSize(4, xpix%*ypix%*4 + 14 + 108);' Size of .bmp file in bytes
BWriteSize(4, "");                    ' 4 app-defined bytes (0,0,0,0)
BWriteSize(4, 14 + 108);              ' offset to where the data starts

' Construct DIB header (BITMAPV4HEADER - this is 108 bytes long)
BWriteSize(4, 108);                   ' size of the DIB header
BWriteSize(4, xpix%);                 ' Width of the bitmap in pixels
BWriteSize(4, flags% band 1 ? -ypix% : ypix%); ' Height of the bitmap in pixels
BWriteSize(2, 1);                     ' number of color planes. Must be 1
BWriteSize(2, 32);                    ' number of bits per pixel
BWriteSize(4, flags% band 2 ? 3 : 0);   ' BI_BITFIELDS = 3 or BI_RGB = 0. No compression
BWriteSize(4, xpix%*ypix%*4);         ' bytes of bitmap data (or 0 as BI_RGB)
BWriteSize(4, 2835);                  ' horizontal pix/m = 72 pix/inch
BWriteSize(4, 2835);                  ' vertical pix/m = 72 pix/inch
BWriteSize(4, 0);                     ' # of palette colours. No palette used.
BWriteSize(4, 0);                     ' Important palette colours, 0 as unused

' Next 4 32-bit words set the Red, Green, Blue and Alpha mask in BI_BITFIELDS
if (flags% band 2) then
```

```
    BWriteSize(4, 0x00ff0000);        ' Red mask    You can swap the RGB
    BWriteSize(4, 0x0000ff00);        ' Green mask  masks around, if you
    BWriteSize(4, 0x000000ff);        ' Blue mask   need other arrangements.
    BWriteSize(4, 0xff000000);        ' Alpha mask  Set 0 if no Alpha channel
else
    BWriteSize(16, "BGRsBGRsBGRsBGRs"); ' Unused in BI_RGB mode
endif
    BWriteSize(52, "");               ' fill the rest with 0s
' We have no palette, so the next thing in the file is the RGB(A) values
BWriteSize(4, bmp%);                  ' Write the bitmap
FileClose();
return 0;
end
```

# C

# Ceil()

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

**`Func Ceil(x|x[] {[]…});`**

x    A real number or a real array.

Returns  When the argument is an array, the function replaces the array with the next higher integral number of all the points and returns either a negative error code or 0 if all was well.

When the argument is not an array the next higher integral number.

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

# Chan...()

# Chan()

This function converts a string that starts with a single channel specification in the current data or XY view into a channel number. This deals with memory and virtual channels, as well as the ordinary channels from a CFS file. Channels referred to do not need to exist. This allows you to know the channel number of a memory channel `Chan("m1")` or virtual channel `Chan("v1")` before you create it. This function was added at Signal version 7.02.

**`Func Chan(spec${, &index%});`**

spec$  A string holding a channel number, a memory channel specification or a virtual channel specification. The specification can be prefixed or followed by white space. If the `index%` argument is omitted, the string must not be followed by any character that could not be part of the channel (apart from white space), so `"m2"`, `" m2"` and `" m2 "` are fine, but `"m2,"` is not.

index%  An optional integer variable that is returned holding the index into `spec$` at which the first character was found that was not part of a channel specification or -1 if all characters were used. If this is omitted, all characters of `spec$` must be used when decoding the channel number. The idea it to allow you to parse an arbitrary string that starts with a channel specification. If the string held `" m2 ,"`, `index%` would be returned holding 3 as the space was the first character that could not be part of the channel specification.

Returns The single argument version either returns a channel number or a fatal error code. The two argument version returns a channel number or 0 if the channel was not recognised; in either case `index%` is -1 if all characters were used or it is the index of the first character that could not be part of a channel specification.

Unlike `ChanList()`, this routine does not care if the channel exists or not, except in the case of duplicate channels (where a channel number cannot be predicted from the source channel and the duplicate code). This allows you to use this to find out which channel a particular memory or virtual channel would use if it existed.

**See also:**
`Chan$(), ChanList()`

# Chan$()

This function converts a channel number or a list of channel numbers into a string, note that virtual and memory channels do not appear as a simple number but as "v1" or "m2". If a channel does not exist in the current view, it is represented as a number.

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

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

Returns A channel specification string, for example "1,3,5..8,v1,m6".

**See also:**
`ChanList()`

# ChanAdd()

This will add the data from a specified channel to one or more other channel's data.

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

`cSpc` A channel specifier for the channels to add data to.

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

Returns Zero.

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

# ChanColour()

**Deprecated**, use ChanColourSet() and ChanColourGet(). This returns and optionally sets the colour of a channel in a file or memory view. This colour overrides the application colour set for the drawing mode of the channel.

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

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

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

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

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

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

# ChanColourGet()

Returns a channel item RGB colour in a data or XY view. This command was added at Signal version 5.02.

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

`chan%`   A channel in the data or XY view.

`item%`   The colour item to get; 0=background, 1=primary, 2=secondary colour for time and result views, the fill colour for an XY channel. XY channels do not have individual background colours, use `ViewColourGet()` to get the background colour for all XY channels.

`r g b`   If present, these variables are returned holding the colour as red, green and blue values in the range 0 to 1.0.

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

**See also:**
`Colour dialog, ChanColourSet(), ColourGet(), ViewColourGet()`

# ChanColourSet()

Sets a channel item RGB colour in a data or XY view. A change to a colour item that would make visible difference will cause the affected view to become invalid and it it will repaint at the next opportunity. This command was added at Signal version 5.02.

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

`chan%`   A channel in the data or XY view.

`item%`   The colour item to get; 0=background, 1=primary, 2=secondary colour for time and result views, the fill colour for an XY channel. XY channels do not have individual background colours, use `ViewColourSet()` to set the background colour for all XY channels.

`r g b`   If present, these arguments set the colour as red, green and blue values in the range 0 to 1.0. If omitted, the item colour is set to the application default.

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

**See also:**
`Colour dialog, ChanColourGet(), ColourSet(), ViewColourSet()`

# ChanCount()

This counts channels in a data or XY view.

`Func ChanCount({chan%});`

`chan%`   If present, this specifies the channels to count (this is ignored for XY views), if omitted, the total channel count is returned. Supported `chan%` values are:

  -1   All channels
  -2   All visible channels
  -3   All selected channels
  -4   Waveform channels
  -5   Marker channels
  -6   Selected waveform channels or visible if none selected
  -7   Visible waveform channels
  -8   Selected waveform channels
  -9   Idealised trace channels
 -10  All selected channels or all visible if none selected
 -11  Real marker channels
 -12  Marker or real marker channels

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

**See also:**
ChanList()

# ChanDecorate()

This command is equivalent to the Channel Decoration dialog. It currently is used with real marker data channels in a time view, but may be extended in the future to cover other channel types.

```
Func ChanDecorate(chan%, mode%, flags%{, ind1%{, ind2%}});
```

chan%  A channel in the time view. This need not be a real marker channel, but has no visible effect on other channel types.

mode%  The channel decoration mode: 0=none, 1=error bars, 2=low-high range. Negative values of mode% return information and do not change the decoration mode: -1= mode%, -2= flags%, -3=ind1%, -4=ind2%.

flags%  This is the sum of:
1    The bar is drawn in the marker colour (otherwise in the channel secondary colour).
2    Draw a centre circle at the channel value.

ind1%  The 0-based index of the data item in the real marker channel to use as the error value in mode 1 and the low value in mode 2. If this argument is omitted, no change is made to the index. If you set an index that does not exist, no decoration is drawn.

ind2%  The 0-based index of the data item in the real marker channel to use as the high value when drawing a range bar. If omitted, no change is made. If you set an index that does not exist, no decoration is drawn.

Return  Unless mode% is negative, the command returns the channel decoration mode at the time of the call. Otherwise, it returns the information requested by mode%.

**See also:**
DrawMode()

# ChanDelete()

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

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

chan%  The channel to delete.

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

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

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

# ChanDiff()

This differentiates the data in specified waveform channels of the current frame in the current view. If the specified channel is not a waveform then the command has no effect. If the frame buffer is being shown, the frame buffer data is used instead. All data points in the channels used are modified. It is an error to use this command on log-binned data.

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

cSpc    A channel specifier for the channels to differentiate.

Returns  0 or a negative error code.

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

# ChanDiv()

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

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

cSpc    A channel specifier for the channels to divide the source channel into.

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

Returns  Zero.

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

# ChanFit()

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

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

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

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

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

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

Returns  0 if the command succeeded.

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

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

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

opt%    This is the sum of:
   1    Estimate the coefficients before fitting, else use current values.
   2    Draw the fit over the user-defined range, not the fit range.
   4    Use maximum likelihood fitting – obeyed for exponential fits only, and then only when fitting open/closed time and burst duration histograms where the original idealised trace data is still accessible.

frm%    Frame number or a negative code as follows:
   -1   All frames in the file.
   -2   The current frame.
   -3   Only tagged frames.
   -6   Only untagged frames.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

| opt% | Returns | opt% | Returns |
|---|---|---|---|
| 0 | Fit type of next fit | 1 | Fit order of next fit |
| -1 | 1=a fit exists, 0=no fit exists | -2 | Type of existing fit or 0 |
| -3 | Order of existing fit | -4 | Chi or least-squares error |
| -5 | Fit probability (estimated) | -6 | X axis value at fit start |
| -7 | X axis value at fit end | -8 | Reference x value |
| -9 | User-defined x draw start | -10 | User-defined x draw end |
| -11 | 1=chi-square, 0=least-square | -12 | Last fit result code |
| -13 | Number of fitted points | -14 | Number of fit iterations used |
| -15 | R-Square value for fit | -16 | Adjusted R-Square value |

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

### See also:
Fit Data dialog, More about fitting, ChanFitCoef(), ChanFitShow(), ChanFitValue(), FitPoly(), FitExp(), FitGauss(), FitSine()

# ChanFitCoef()

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

**Set and get coefficients**

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

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

chan%    The channel number of the fit to access.

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

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

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

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

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

**Get and set the hold flags**

This command variant sets the hold flags (equivalent to the Hold check boxes in the Fit Data dialog Coefficients page).

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

chan%    The channel number of the fit to access.

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

Returns  This always returns 0.

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

# ChanFitShow()

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

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

chan%    The channel number of the fit to access.

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

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

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

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

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

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

# ChanFitValue()

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

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

chan%   The channel number of the fit to access.

x       The x axis value at which to evaluate the current fit, this can be either a number or a string to be interpreted. You should be aware that some of the fitting functions can overflow the floating point range if you ask for x values beyond the fitted range of the function.

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

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

# ChanImage()

You can set a background image behind each channel in a data or XY view. Images can be read from `.bmp`, `.jpg/jpeg`, `.png` and `.tif/tiff` files on disk. See the View menu Channel Image command for details. There are three command variants:

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

chan%   A channel number in the current view. In an XY view, all channels share the same bitmap.

path$   The file name that holds the image. The first command variant sets the image, the third variant reads back the name of the file. If you set an empty name, this releases any memory used to cache the bitmap within the program. Setting an image does not change the display mode; use the second variant to make sure that the image is visible. Set the path to `"<CB>"` to use whatever image happens to be on the clipboard at the time of the function call.

mode%   There are three display modes that you can set in the second command variant: 0=no display, 1=fill background, 2=fill rectangle. You can also set mode -1, meaning no mode change.

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

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

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

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

We do not provide any method to size a channel area so a bitmap becomes a specific size. However, you can use `ChanPixel()` to find the scaling between x and y axes and pixels. It is possible to then iteratively change the window size (or adjust the channel weight with `ChanWeight()`) toget the desired effect.

**See also:**
ChanColour(), ChanPixel(), ChanWeight()

# ChanIndex()

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

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

`chan%`  A channel number. Currently, this function is useful only with real marker channels.

`index%` If present and positive, this sets the zero-based channel index. If omitted or -1, the command returns the current index. If -2, the command returns the number of index values for the channel - the number of floating point values attached to each marker.

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

**See also:**
MemChan(), MarkInfo()

# ChanIntgl()

This integrates the data in specified waveform channels of the current frame in the current view. If the specified channel is not a waveform then the command has no effect. If the frame buffer is being shown, the frame buffer data is used instead. All data points in the channels used are modified. It is an error to use this command on log-binned data.

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

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

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

Returns  0 or a negative error code.

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

# ChanItems()

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

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

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

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

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

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

**See also:**
ChanRange(), ChanPoints(), XYRange()

# ChanKey()

Channel keys display addition information about a channel. Currently only XY views have a key to identify the displayed channels. This command is used to control keys in a generic way and may be extended (it is taken from a more generally applicable function in Spike2, and has been kept as similar as possible to the Spike2 function). It is currently only usable with XY views but this will be extended in the future. There is also the `XYKey()` command that is XY key-specific. There are four command variants:

```
Func ChanKey(chan%, type$, flags%, x, y{, s{, w{, h}}});
Func ChanKey(chan%, 0);
Func ChanKey(chan%, set%, val{, val$));
Func ChanKey(chan%, get%{, &get$});
```

The first variant creates a key or modifies an existing key and returns the previous `flags%` value. The remaining commands will not work unless the key already exists. The second variant hides the key, but does not change any other settings, and returns the previous `flags%` value. The third variant sets key-specific values and the last variant reads back key-specific values.

`chan%`   A channel number in the current view. In an XY view this is ignored and should be set to 0.

`type$`   The type of the key to create. This is case insensitive, we currently only support `"XY"`.

`flags%` The sum of a set of values that control the appearance (or non-appearance) of a key. If you set `flags%` to 0 you can omit the remaining arguments to hide the key. The values are:

| Value | Meaning |
|---|---|
| 1 | Make the key visible. If this value is not added, the key is hidden. |
| 2 | Controls the key background. If set the key is transparent (XY). If not set, the key uses the channel background colour. |
| 4 | If set, the key has a border. If unset, there is no border. |
| 8 | Sets the key orientation (if applicable). If set, the key is vertical, otherwise it is horizontal. Not applicable to XY keys. |
| 16 | If set, the x,y position is relative to the entire view. If clear, it is relative to the channel. |
| 32 | Currently unused in Signal |
| 64 | If set, only the top left corner of the key is forced to be visible, the size is ignored. This flag value was added in version 7.06. |

`x,y`   The key position as a percentage of the available space (space less the key size). The x space runs from left to right, and is the space used by the x axis. The y space runs from top to bottom and if `flags%` has 16 added it is the entire view height, otherwise it is the channel height. The scaling is arranged so that 0,0 has the key at the top left and 100,100 has the key at the bottom right.

`s`   The font size in points to use for text in the key and for scaling the size of the key. If omitted or set to 0, a default value is used that is based on the size of the text in the view that the key belongs to. You can request any size you like, but there is a limit on the maximum and minimum font sizes that will be used. Negative sizes are reserved and currently have no effect.

`w,h`   These are not currently used. They are reserved for future channel keys where the size is not calculated from the key content.

`set%`   Some keys will have special settable parameters. This field identifies the parameter to set and `val` is the value. The parameters are numbered from 1 upwards. You can only set a value when the key is visible. Setting a value that the key does not recognise has no effect. Currently defined values are:

Key ˢᵉ Use

XY 1   Set non-zero to hide the example trace drawn in the XY key for each channel. Set 0 to show it.

`val`   A value to set as determined by `set%`. If you are setting a string with `val$` (no key does this yet), set this to 0.

`val$`   A string value to set, as determined by `set%`. This is reserved for future use.

`get%`   This returns a value set by `ChanKey(chan%, set%, val{, val$})`. Negative values read back channel key settings. The returned value is usually 0 if there is no key defined for the channel:

| get% | Returned value |
|---|---|
| 1..n | The key-specific value set by `ChanKey(chan%, set%, val)` |
| 0 | This hides the key and returns the `flags%` value before the key was hidden |
| -1 | Return the key type in get$, returns 0 if no key, 1 if a key |
| -2 | The current `flags%` value |
| -3, -4 | The current key x and y position |

|  |  |
|---|---|
| -5 | The current key scale |
| -6, -7 | The current key width and height (if controllable), otherwise 0. |

get$    Used to retrieve string values as required by `get%`, matching `val$`, again reserved for future use.

**See also:**
`XYKey()`

# ChanKind()

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

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

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

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

| | | | | | |
|---|---|---|---|---|---|
| 0 | Waveform | 1 | Marker | 3 | XY channel |
| 5 | Idealised trace | 6 | Real marker | | |

**See also:**
`ViewKind()`

# ChanList()

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

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

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

types%    This argument specifies which channels to return. If omitted or set to zero, all channels are returned. The values are as follows:

| | | |
|---|---|---|
| 1 | 0x1 | Waveform or result view channel |
| 2 | 0x2 | Marker channels |
| 4 | 0x4 | Idealised traces |
| 8 | 0x8 | Real marker channels |

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

| | | |
|---|---|---|
| 128 | 0x80 | Exclude CFS data file channels (means memory and virtual channels only) |
| 256 | 0x100 | Exclude virtual channels |
| 512 | 0x200 | Exclude memory channels |
| 1024 | 0x400 | Exclude visible channels |
| 2048 | 0x800 | Exclude hidden channels |
| 4096 | 0x1000 | Exclude selected channels |
| 8192 | 0x2000 | Exclude non-selected channels |

This argument is ignored in an XY view where all channels are the same type.

str$    A channel specification such as `"1..10,20,m1,v1..v3"`. Only channels that exist in the current view are returned in `list%`. If `types%` is provided, only channels that match both the string and `types%` are returned in `list%`.

Returns    The number of channels that would be returned if the array was of unlimited length, 0 if the view is not a data or XY view or -1 if the `str$` parameter was badly formed or illegal.

**See also:**
```
Channel lists, Channel specifiers, ChanShow(), ChanCount(), ChanDelete(), DlgChan(),
XYSetChan()
```

# ChanMean()

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

**Func ChanMean(chan%, start, finish{, stDev});**

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

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

finish  The last position in x axis units.

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

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

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

# ChanMeasure()

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

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

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

type%   The type of measurement to take, see the documentation of the Cursor Regions window for details of these measurements. Trying to use the Curve area measurement on log-binned data will halt the script. The possible type% values are:

| | | | |
|---|---|---|---|
| 1 Curve area | 6 Modulus | 11 Standard deviation | 16 Standard error |
| 2 Mean | 7 Maximum | 12 Extreme | 17 RMS error |
| 3 Slope | 8 Minimum | 13 Peak | |
| 4 Area | 9 Peak to peak | 14 Trough | |
| 5 Sum | 10 RMS amplitude | 15 Point count | |

start   The start position for the measurement in seconds. If start is greater than finish, the result is 0.

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

end     The end position in seconds.

end$    The end position as a string, again allowing time expressions.

Returns  The function returns the requested measurement value.

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

# ChanMult()

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

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

cSpc    A channel specifier for the channels to multiply.

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

Returns  Zero.

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

# ChanNegate()

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

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

cSpc    A channel specifier for the channels to negate.

Returns  0 or a negative error code.

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

# ChanNumbers()

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

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

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

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

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

# ChanOffset()

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

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

cSpc    A channel specifier for the channels to offset.

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

Returns  0 or a negative error code.

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

# ChanOrder()

This command changes the order of channels in a data view, groups channels with a y axis so that they share a common y axis,  sets the channel sorting order and gets list of visible channels in screen order from top to bottom. There are four command variants.

**Move channels**
The first moves a group of channels relative to a channel or to a channel position in the list of all channels in the view:

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

dest%    The destination channel number or a value less than 1 indicating the position in the list of all channels including hidden channels but excluding channels in cSpc. When used as a list position, 0 is the top

channel, -1 is the next, and so on. If there are n channels, values of -n or less are taken to mean the bottom channel.

pos%     The drop position relative to the destination channel: -1=drop above, 0=drop on top, 1=drop below. If you drop between grouped channels, the dropped channels become members of the group (as long as they have a y axis).

cSpc     A channel specifier for the channels to move.

Returns  The number of moved channels.

### Get information and ungroup channels

The second command variant gets grouping information and can convert all the channels in a group to ungrouped channels.

```
Func ChanOrder(dest%, opt%);
```

dest%    A channel number in the view that identifies a group. A group is either a single channel, or a set of channels that share a single y axis.

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

Returns  The number of channels in the group, the nth channel in the group, or the number of changed channels, depending on opt%.

### Sort channels

The third command variant sorts channels into numerical order.

```
Func ChanOrder(order%);
```

order%   This form of the command sorts all the channels into numerical order. Set -1 for low, 1 for high numbered channels at the top and 0 to use the default channel ordering set by the Display page of the Preferences dialog.

Returns  -1 if low-numbered channels were previously at the top, 1 if high-numbered channels were previously at the top.

### Get ordered channel lists

The fourth variant gets channel lists, ordered from top to bottom of all channels, all channels that start a group, and all channels within the nth group.

```
Func ChanOrder(list%[]{, sel%});
```

list%    An array that is returned with a channel list. list%[0] holds the number of items that follow in the list, so if list%[0] holds n, there are valid channel numbers in list%[1] through list%[n]. The returned channel numbers are always in the order of the channels on the screen, with channels at the top of the screen first.

sel%     Optional, taken as 0 if omitted. If 0, the list contains all the visible channels in the view. If -1, the list contains the first channels of each group, so if there are no groups, list%[0] holds 0. If sel% is greater than 0, say n, the list is returned holding all the channels in the nth group.

Returns  The number of channels that match the option. This can be greater than list%[0] if the list% array is too small.

### See also:

Channel specifiers, ChanList(), ChanWeight(), ViewStandard(), YAxis(), YAxisLock()

# ChanPenWidth()

This command sets and gets the pen width for a channel in a Data view. The pen width for channels in an XY view is handled by the XYDrawMode() command. This script command is the equivalent of the View menu channel Pen Width dialog, it was added in Signal version 5.00.

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

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

new    If present, the new width of the pen to use for the specified channels, in points. A point is 1/72 of an inch, which is approximately 1 pixel (on most displays in 2010). If you set a negative width, the channel will use the pen width set in the Display page of the Preferences dialog for data. A width of zero will set the thinnest pen (1 pixel) possible.

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

**See also:**

Channel specifiers, Edit menu preferences, Pen width dialog, XYDrawMode()

# ChanPixel()

This command gets screen pixel information about the current file, memory or XY view. All positions are given with respect to a rectangle that has the 0,0 point at the top, left corner with the x co-ordinates increasing to the right and the y co-ordinates increasing downwards. In the following description, the rectangles that we refer to are:

Desktop    The rectangle that encloses all your monitors.

View       The rectangle that encloses the drawing area of the current time, result or XY view. This does NOT include the frame around the view.

Channel    The rectangle that encloses all the data for a particular channel.

Superview  The channel superview is the rectangle that encloses all the channel displays (not including any x or y axes). In the future we may allow channels to be arranged in a grid, in which case the superview for a channel will be the rectangle that includes all the channels in the column of the grid that includes the nominated channel.

```
Func ChanPixel(chan%, &x, &y{, mode%{, &w|xp{, &h|yp}}});
```

chan%  A channel in the current file, memory or XY view. This must be supplied as a valid channel for mode 2, but can be set to 0 for modes 0 and 1 if you do not want the y information.

x      This is a real variable that is set depending on the value of mode%.

y      This is a real variable that is set depending on the value of mode%.

mode%  An optional variable, taken as 0 if omitted, that determines the returned values:

0    x is set to the number of x axis units equivalent to a move of one pixel to the right. y is set to the number of user units that are equivalent to a movement of one pixel up. If there is no y axis or chan% does not exist, it is set to 0. Normally the x and y axes are in linear mode, but they can be set to logarithmic mode, in which case the returned values are the log increment that corresponds to one pixel.

     You will need this information if you use the ToolbarMouse() or DlgMouse() function to get the mouse position and want to locate data elements that are close to the mouse pointer.

1    x and y (if chan% exists) are set to the pixel positions relative to the channel superview of the axis positions xp, yp in channel chan%. You can find the position of the channel superview with respect to the view with mode% set to 3.

2    x, y, w and h are returned holding the channel rectangle of channel chan%, which must exist. x,y are relative to the channel superview.

3    x, y, w and h are returned holding the channel superview position and size. The position is relative to the view. Currently there is only one superview, so chan% is ignored. In the future we may allow channels to be arranged in a grid, and the channel number will matter as it will determine which superview is required.

4    x, y, w and h are returned holding the view position and size. The position is relative to the Windows desktop. chan% is ignored and should be 0.

5    x, y, w and h are returned holding the x axis position and size. The position is relative to the view. Currently there is only one x axis, so chan% is ignored and should be 0. In the future we may allow channels to be arranged in a grid, and the channel number will matter as it will select the x axis to use.

6    x, y, w and h are returned holding the y axis position and size for the channel `chan%` (which must exist). The position is relative to the view.

7    x, y are returned set to the sizes (in pixels) of a representative character that will be used for a cursor label. This is to allow you to position a vertical cursor label above or below a specific value. The channel is currently ignored.

w,h    Real variables that are set to the width and height of the rectangle selected by `mode%`.

xp,yp    The x and y positions in axis units to map to pixels when `mode%` is 1.

Returns    A set of flags indicating which values were returned and if the units were modified, or -1 if the command could not be used. Flag values are:

1    The x axis value is set (this should always be set).

2    The y axis value is set (not set if the channel does not exist, is invisible or does not have a y axis).

4    In mode 0, the x axis is in log mode, so the x value is a log increment per pixel.

8    In mode 0, the y axis is in log mode, so the y value is a log increment per pixel.

16    In mode 1, the `xp` value is outside the visible channel rectangle on screen.

32    In mode 1, the `yp` value is outside the visible channel rectangle on screen.

**See also:**
`DlgMouse(), ToolbarMouse()`

# ChanPoints()

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

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

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

Returns    The number of data points in a frame.

**See also:**
`ChanRange(), ChanItems(), XYCount()`

# ChanRange()

This function finds the number of data items within a given x axis range.

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

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

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

`finish`    The end position in x axis units.

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

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

**See also:**
`ChanPoints(), ChanItems()`

# ChanRectify()

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

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

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

Returns 0 or a negative error code.

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

# ChanScale()

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

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

cSpc    A channel specifier for the channels to scale.

val     The value to multiply by.

Returns 0 or a negative error code.

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

# ChanSearch()

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

```
Func ChanSearch(chan%, mode%, &sT, eT{, sp1{, sp2{, wid{, sp3}}}});
```

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

mode%   This sets the search mode, as for the active cursors. See the cursor mode dialog documentation for details of each mode. Only the first three modes can be used for log-binned data.

| | | |
|---|---|---|
| 1 Maximum | 9 Steepest rising | 17 Turning point |
| 2 Minimum | 10 Steepest falling | 18 Slope percentage |
| 3 Extreme | 11 Slope peak | 19 Repolarisation % |
| 4 Peak find | 12 Slope trough | 20 Expression |
| 5 Trough find | 13 Slope threshold | 21 Outside dual levels |
| 6 Threshold | 14 Rising slope threshold | 22 Within dual levels |
| 7 Rising threshold | 15 Falling slope threshold | 23 Data points |
| 8 Falling threshold | 16 Absolute max slope | |

sT      The start time for the search, the parameter will be returned containing the result (the position of the feature) of a successful search. Note that as this argument is a reference parameter it has to be a real variable (so that it can be updated), values such as 0 or MinTime() will not be accepted by the script compiler.

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

sp1     This is the threshold level for threshold crossings and the reference level for extreme mode. It is in the y axis units of the search channel (y axis units per second for slopes). For data points mode it is the number of points. If omitted, the value 0.0 is used. For repolarisation % mode it is the time (in seconds) at which the 100% repolarisation level is measured (the 0% value is measured at the start time for the search). Set it to 0 if sp1 is not required for the mode.

sp2     This is the minimum required amplitude for peak and trough searches, the hysteresis for threshold crossings, and the percentage for slope percentage searches. If omitted, the value 0.0 is used. Set it to 0 if sp2 is not required for the mode.

wid     This is the width in seconds for all slope measurements, the maximum peak width for peak and trough searches, and the post-transition delay for threshold crossings. If omitted, the value 0.0 is used. Set it to 0 if width is not required for the mode or you do not want to set a maximum width or post-transition delay.

sp3    This is the second level for dual-level searches.

Returns  0 if the search succeeds or -1 if the search fails or a negative error code.

**See also:**
Active cursors, Cursor mode dialog, `ChanMeasure()`, `ChanValue()`, `CursorActiveGet()`, `CursorActiveSet()`

# ChanSelect()

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

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

cSpc    A channel specifier for the channels to select.

new%    If present it sets the state: 0 for unselected, not 0 for selected. If omitted, the state is unchanged. Attempts to change invisible channels are ignored.

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

**See also:**
`ChanList()`, `ChanOrder()`, `ChanVisible()`, `ChanWeight()`

# ChanShift()

This shifts the data in specified waveform channels of the current frame in the current view a specified number of points right or left. The data is actually rotated so that points that 'fall off' one end are shifted back in at the other. If the frame buffer is being shown, the frame buffer data is used instead. All data points in the channels used are modified.

`Func ChanShift(cSpc, shift%);`

cSpc    A channel specifier for the channels to shift.

shift%  The number of point to shift the data. A negative value shifts points left, a positive value shifts points right.

Returns  0 or a negative error code.

**See also:**
`ShowBuffer()`, `ChanShow()`, `ChanZero()`, `ArrAdd()`

# ChanShow()

This function displays or hides a channel, or a list of channels, in a data or XY view. Showing a channel that is on or doesn't exist has no effect.

`Func ChanShow(cSpc{, yes%});`

cSpc    A channel specifier for the channels to display or hide.

yes%    If this is non-zero it turns the specified channels on and if it is zero it turns them off. If `yes%` is omitted no changes are made.

Returns  If you set `chan%` to a positive channel number the function returns the channel state at the time of the call, 1 for visible, 0 for invisible. Otherwise it returns -1.

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

# ChanSmooth()

This function smooths the data in specified waveform channels of the current frame in the current view by replacing each point by the average of n adjacent points, where n can be 3 or 5. If the frame buffer is being shown, the frame buffer data is used instead. All data points in the channels used are modified.

```
Func ChanSmooth(cSpc, width);
```

cSpc     A channel specifier for the channels to smooth.

width   This sets the width to smooth over, either 3 or 5.

Returns   0 or a negative error code.

**See also:**
ShowBuffer(), ChanShow(), ChanZero(), ArrFilt()

# ChanSub()

This function subtracts the data in one channel from the data in one or more other channels.

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

cSpc     A channel specifier for the channels to subtract from.

src%    The number of the channel containing the data to subtract.

Returns   Zero.

**See also:**
ChanAdd(), ChanDiv(), ChanMult()

# ChanSubDC()

This function subtracts any DC offset present in the data in specifed waveform channels of the current frame in the current view. The DC offset is measured over the time range specified, all data points in the channels are modified. If the frame buffer is being shown, the frame buffer data is used instead.

```
Func ChanSubDC(cSpc, start, finish);
```

cSpc     A channel specifier for the channels to subtract from.

start   The start position for measurement of the DC level.

finish The end position for measurement of the DC level.

Returns   0 or a negative error code.

**See also:**
ShowBuffer(), ChanShow(), ChanZero(), ArrSum()

# ChanTitle$()

This function returns the title for a channel in a data or XY view, it can also set the channel title. For XY views only, it can also be used to get or set the Y axis title. In an XY view the channel titles are visible in the Key window.

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

chan%   The channel number (1 to n). For an XY view only, a channel number of zero can be used to access the Y axis title.

new$    If present, in a data or XY view this string holds the new channel title. If the string is too long, it is truncated.

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

**See also:**
`ChanUnits$(), XTitle$(), XYKey(), YAxis()`

# ChanUnits$()

This returns the units for a waveform channel in a data view or the Y axis units in an XY view. In a memory, XY, or data view it can also set the units.

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

`chan%`   A channel number (1 to n). This is ignored in an XY, where it operates on the Y axis units only.

`new$`    If present, in a data, XY or memory view, this string holds the new channel title. If the string is too long, it is truncated.

Returns   It returns the original units for the channel. If the channel does not exist or is not of a suitable type, the function does nothing and returns an empty string.

**See also:**
`ChanTitle$(), XUnits$(), YAxis()`

# ChanValue()

This returns the value on a given channel at a given position. It returns a value in the y axis units of the channel display mode. If the display mode has no y axis the value is the x axis position of the next item on the channel.

This returns the value corresponding to an x axis value. Use the `View(v,c).[bin]` notation or `BinToX(bin)` to access view data by bin number.

**`Func ChanValue(chan%, pos{, &data%{, mode%{, binsz}}});`**

`chan%`   A channel number (1 to n).

`pos`     The x axis position for which the value is needed.

`data%`   This is returned as 1 if there was data at the position, 0 if there was not. For example on a waveform channel with time on the x axis, if there was no waveform point within `BinSize(chan%)` of the time set by `pos`, this would be set to 0.

`mode%`   For a waveform channel, if this is present and set to 1 then linear interpolation between points is used to generate a more accurate value, otherwise the nearest waveform point is used. If present for a marker channel, this sets the display mode to use for extracting a value from a view. If an inappropriate mode is requested or if `mode%` is absent, the actual display mode is used. The modes for marker channels are:

    0    The current mode for the channel. Any additional arguments are ignored.

    1    Dots mode for markers, returns the position of the marker at or after `pos`.

    2    Lines mode for markers, result is the same as mode 1.

    3    Rate mode for markers. If the `binSz` argument is present it sets the width of each bin, otherwise the bin width is set to 1.0.

    9    Plot mode for real marker channels only using the currently selected channel data index selected for that channel. Linear interpolation is always carried out unless `pos` is before the first marker point or after the last, when the nearest point is used.

`binSz`   If present when `mode%` specifies rate mode for markers, this sets the width of the rate histogram bins in x axis units.

Returns   It returns the value or zero if no data is found. For display modes with a y axis, if there is no data within `BinSize(chan%)` of the position, the value is zero. This is the same value returned by the Cursor Values menu for the channel.

If `data%` is not provided, any error stops the script. Errors include: no current window, current window not a data view, no data at `pos`, and `pos` beyond range of x axis. If `data%` is present, errors cause `data%` to be set to 0.

For example, to get data value on channel 1 at the position of the cursor number 1 in the view on data file, `mydata.cfs`:

```
vdata%:=ViewFind("mydata.cfs");   'view handle for data
FrontView(vdata%);                'focus on the data window
ampl:=ChanValue(1,Cursor(1));     'get data value at cursor
```

**See also:**
BinToX(), Cursor(), ChanMeasure(), DrawMode(), Interact(), ChanIndex()

# ChanVisible()

This returns the shown state of the channel as 1 if the channel is visible and 0 if it is hidden. If you use a silly or non-existant channel number, the result is 0 (not displayed).

**Func ChanVisible(chan%);**

chan%   The channel number (1 to n) to report on.

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

**See also:**
ChanShow()

# ChanWeight()

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

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

cSpc    The specification for the list of channels to process.

new     If present, a value between 0.001 and 1000.0 that sets the weight for all the channels in the list. Values outside this range are limited to the range.

Returns  The command returns the channel weight of the first channel in the list.

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

# ChanZero()

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

**Func ChanZero(cSpc);**

cSpc    A channel specifier for the channels to zero.

Returns  0 or a negative error code.

**See also:**
ShowBuffer(), ChanShow(), ArrConst()

# Chr$()

This function converts a code to a character and returns it as a single character string.

**Func Chr$(code%);**

code%   The code to convert. Codes that have no character representation will produce unpredictable results when printed or displayed.

Returns  A string holding one character to represent the code.

**See also:**
`%c` format in `Print()`, `Asc()`, `DelStr$()`, `LCase$()`, `Left$()`, `Len()`, `Mid$()`, `Print$()`, `Right$()`, `Str$()`, `UCase$()`, `Val()`

# Colour()

**Deprecated**. This is provided for backwards compatibility with old scripts. Use `ColourGet()` and `ColourSet()` in new scripts.

This function gets and/or sets the colours of items. Colours are set in terms of the colour palette displayed in the Colour menu, not directly in terms of colours. XY view channels are coloured using XYColour().

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

`item%`   This is the item number, being the position of the item in the Colour menu, as follows:

| | | | |
|---|---|---|---|
| 0 | Channel numbers | Rate histogram fill | Convoluted trace |
| 1 | Data view background | Text labels | Closed state |
| 2 | Waveform as line | Cursors & cursor labels | XY channels |
| 3 | Waveform as dots | Controls (not used) | XY fill |
| 4 | Waveform as skyline | Data display grid | XY key labels |
| 5 | Waveform as histogram | Axis markings and text labels | Convoluted trace (fitted) |
| 6 | Waveform histogram fill | Tagged frames background | Open state |
| 7 | Makers as lines | Frame list traces | |
| 8 | Markers as dots | XY view background | |
| 9 | Marker text | Error bars | |
| 10 | Rate histogram outline | Fitted data | |

`col%`   If present, this sets the index of the colour in the colour palette to be applied to the item. There are 40 colours in the palette, numbered 0 to 39. The first 7 colours in the palette are set to grey scales from black to white, and the rest can be selected or mixed from basic colours.

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

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

# ColourGet()

Get the RGB colour for an item in the palette, the main colour table or the marker colour table. This can also be used to get the colour scales used for colour mapping.

**Get system colour**
This function was added to Signal at version 5.02 to encourage users to use RGB colours, rather than a colour palette (see the deprecated `Colour()` command).

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

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

`item%`   The item number in the table selected by `table%`. See `ColourSet()` for a description of the item numbers.

r g b   If present, returned as the red, green and blue colour values in the range 0.0 to 1.0.

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

**Get colour scale**

This was added at version 7.02 to give script users access to the colour scales used by the `ArrMapImage()` script command. Colour scales can be viewed, created and deleted in the Colour Scale dialog. The colour scales are saved in the registry.

```
Func ColourGet(n%, &name${, map[][]});
```

n%   The item number of the colour scale. Item numbers 0 through 10 are reserved for preset colour scales (of which items 0 through 7 are implemented). Items 10 through 20 are available for user-defined colour scales. The preset map numbers are:

| Map# | Name | Description |
|------|------|-------------|
| 0 | Monochrome | Runs linearly from black to white. |
| 1 | Rainbow | An approximation to the rainbow. |
| 2 | Thermal | Black to red to yellow to white to blue. |
| 3 | Geography | Black to blue to green to brown to grey to white. |
| 4 | Brightness | Runs from dark blue through mid-red through yellow to white. |
| 5 | The Blues | Black through shades of blue and on to white. |
| 6 | The Greens | Black through shades of green to white. |
| 7 | The Reds | Black through shades of red to white. |
| 8-9 | | Reserved for future use. |

name$   This is used to return the name associated with the colour scale.

map   This optional argument is a two-dimensional array. The first dimension must be at least 4, the second should be large enough to collect all the control points (10 is sufficient for all current colour scales). If the second dimension is too short, the array is filled and the extra data is discarded. See *How colour scales are implemented* for a description of the array values that are returned.

Returns   The number of positions in the colour scale or 0 if n% is valid but the slot is empty or -1 if you request an impossible value of n% or -2 if the map array is present and its dimensions are too small to be useful.

**See also:**

Colour dialog, `ChanColourGet()`, `ColourSet()`, `ViewColourGet()`

# How colour scales are implemented

The `Spline2D()` and `ArrMapImage()` script commands also use colour scales when generating bitmaps. The `ColourSet()` and `ColourGet()` script command create, delete and access colour scales.

Colour scales are implemented as a set of from 2 to 10 control points, each of which has a colour. These control points are positioned along a line that can be though of as running between 0 and 1. The first control point is at position 0, the last at position 1 and any others are positioned along the line. The colour at any position on the line is obtained by linearly interpolating the colours of the control points either side of the position.

In terms of the script language, this set of control points is implemented through a two dimensional array. The first dimension has size 4, the second dimension is the size of the number of control points. If we have `map[4][n%]`, representing n% control points, elements `map[0][]` hold the Red contribution to the colour as a value in the range 0.0 (no red) to 1.0 (as much red as possible), elements `map[1][]` hold the green contribution and elements `map[2][]` holds the blue contribution. Elements `map[3][]` hold the position of the element on the line. Element positions must increase with the element number. We store the positions such that map[3][0] always holds 0.0 and the last element position is always 1.0. We allow you to define a map in `ColourSet()` with arbitrary positions (as long as they increase with element index), but we convert them into the equivalent of 0 to 1 for storage.

We use floating point numbers for colours to avoid saying anything about how colours are actually implemented for future-proofing reasons.

A very simple array could be:

```
var monoChrome[4][2] := {{0,0,0,0}, {1,1,1,1}};
```

In this case the colours go from black at position 0 to white at position 1 with a grey scale in between. If you want to see all the colour scales defined on your machine you can run this script:

```
var map[4][10];      ' [r, g, b, pos][positions]
var name$;           ' the name (if set) of each colour scale
var n%, sz%;         ' index and number of positions
for n% := -2 to 20 do
    sz% := ColourGet(n%, name$, map);
    if (sz% > 0) then
        PrintLog("Colour scale: %d, name: %s\n%.4f\n\n", n%, name$, map[][:sz%]);
    endif
next
```

If you make a change to a colour scale with script recording enabled, Signal will generate the code to implement your change. For example, here is the result of creating the reverse of the Rainbow scale:

```
var v11%:= LogHandle();
FrontView(v11%);
SetColourScaleReverseRain(); 'Set colour scale

Proc SetColourScaleReverseRain()
const ReverseRain[4][5] :=
{
    {1.000, 0.000, 0.000, 0.000},
    {1.000, 1.000, 0.000, 0.247},
    {0.000, 1.000, 0.000, 0.498},
    {0.000, 0.000, 1.000, 0.757},
    {0.251, 0.000, 0.251, 1.000}
};
ColourSet(10, "ReverseRain", ReverseRain);
end
```

You could shorten this further if you do not mind having the ReverseRain array as a global by deleting the procedure call, the Proc line and the end statement.

# ColourSet()

Set the RGB colour for an item in the palette, the Application colour table or the marker colour table. If you change the colour of any item that is visible, this will make the display invalid and the item will repaint at the next opportunity. You can also use this command to set the colour scale used for colour mapping. This command was added at Signal version 5.02.

### Set system colour
These functions were added to Signal at version 5.02 to encourage users to use RGB colours, rather than a colour palette (see the deprecated Colour() command).

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

table% -1 to select the colour palette, 0 for the main colour table, 1 for the overdraw cycling colours table.

size% Used to set the size of the overdraw cycling colours table (table% = 1) in the range 8 to 100. The initial, default value is 8 for colours 0 to 7. If you increase the size, the new table elements are set to black. Setting a size of 0 resets the table to the default size. Setting a size of -1 resets the table size and sets default colours. You can use -1 with any table; setting the table size only works for the overdraw cycling colours table.

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

| | | |
|---|---|---|
| 0 Channel numbers | 11 Rate histogram fill | 22 Convoluted trace |
| 1 Data view background | 12 Text labels | 23 Closed state |
| 2 Waveform as line | 13 Cursors & cursor labels | 24 XY channels |

| 3 Waveform as dots | 14 Controls (not used) | 25 XY fill |
|---|---|---|
| 4 Waveform as skyline | 15 Data display grid | 26 XY key labels |
| 5 Waveform as histogram | 16 Axis markings and text labels | 27 Convoluted trace (fitted) |
| 6 Waveform histogram fill | 17 Tagged frames background | 28 Open state |
| 7 Makers as lines | 18 Frame list traces | 29 Information window text |
| 8 Markers as dots | 19 XY view background | 30 Information window background |
| 9 Marker text | 20 Error bars | |
| 10 Rate histogram outline | 21 Fitted data | |

For the overdraw cycling colours table it is an index from 0 up to the table size-1.

r g b    Sets the red, green and blue values in the range 0 to 1.0 for the selected item in the selected table.

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

## Hierarchy of colours

The colour of items in a channel is determined by the following sequence:

1. If there is a colour set for the channel (see `ChanColourSet()`), this is used unless this is a sample document and the data is not being saved, in which case the *Not saving to disk* (Application colour item 24) colour is used.

2. If there is a colour set for the view (see `ViewColourSet()`), this is used.

3. Otherwise the item is looked up in the Application colour table and that colour is used.

You can think of this as colours being inherited from a hierarchy of Application->View->Channel with each item able to override the colour it inherits.

## Set colour scale or delete user-defined item

This was added at version 7.02 to give script users access to the colour scales used in colour mapping. Colour scales can be viewed, created and deleted in the Colour Scale dialog. The colour scales are saved in the registry. This command allows you to write the non-preset colour scales and to delete user-defined scales.

```
Func ColourSet(n%, name${, const map[][]});
```

n%       The item number of the colour scale. Item numbers 0 through 10 are reserved for preset colour scales (of which items 0 through 7 are implemented). Items 10 through 20 are available for user-defined colour scales.

name$    This sets the name associated with the colour scale. If this name is blank and this a user-defined colour scale, the user-defined entry is deleted. You cannot delete the preset entries or items with negative numbers.

map      This argument is a two-dimensional array. The first dimension must be at least 4, the second should be of size from 2 to 10). See *How colour scales are implemented* for a description of the array values that you should fill in. Note that if you use `ColourSet()` followed by `ColourGet()`, you may not get back exactly the same values. The current system quantises the colour and position values to one part in 256. From version 7.02 you can omit this when `name$` is blank.

Returns  0 for success or -1 if you request an impossible value of `n%` or -2 if the `map` array is present and its dimensions are too small or too large to be usable, -3 if the position values in `map[3][]` are not in ascending order and -4 if there was any other problem. If you set Red (`map[0][]`), Green (`map[1][]`), or Blue (`map[2][]`) values that are outside the range 0 to 1, they are limited to this range. The position values do not need to be in the range 0.0 to 1.0 as these are scaled by the command so that the first position is 0.0 and the last is 1.0; the scaling is linear.

## See also:

Colour dialog, `ChanColourSet()`, `ColourGet()`, `ViewColourSet()`, `ViewUseColour()`

# Conditioner commands

The `Cond…` family of commands control external signal conditioners. At the time of writing, these commands support the CED 1902 programmable signal conditioner, the Power1401 programmable gain option, the Digitimer D360 and the Axon Instruments CyberAmp. Other conditioners may be added in the future.

These commands do not define which serial port is used by the conditioner nor the type of conditioner supported. When you install Signal you must choose the conditioner type and set the serial port. All these commands require a port% argument. This is the physical waveform input port number that the conditioner channel is attached to. It is not a channel number in a view.

You can access the built-in interactive support for the conditioner from the sampling configuration channel setup dialog. This can be a useful short-cut to getting the lists of gains and signal sources available on your conditioner(s).

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

# CondFeature()

This command gets and sets special signal conditioner features that are not general enough to have dedicated commands to support them. See the `CondSet()` command for more details of conditioner operation. There are four command variants:

**Get feature count**
```
Func CondFeature(port%);
```

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

Returns  The number of special features supported by the signal conditioner.

**Get feature information**
The command supports two types of features: those that have a set of discrete values such as ["None", "Rectify"], and those that support a continuous range of floating point values, such as 10.0 to 25.6, for example. To determine if the feature is continuous or discrete, call the function with only the first 3 or 4 arguments.

```
Func CondFeature(port%, feat%, &name${,&flags%{, list$[]}});
Func CondFeature(port%, feat%, &name${,&flags%{, &low{, &high}}});
```

feat%    The feature number, from 1 to the number of features available

name$    Returned set to the name of the feature.

flags%   Returned set to the feature flags. Currently, none are defined, so this will be 0.

list$    Returned set to an array of the possible settings (strings) for discrete type.

low      Returned as the low limit for a feature with continuous values.

high     Returned as the upper limit for a feature with continuous values

Returns  The number of discrete feature values, or 0 if the feature supports continuous values over the range returned in `low` and `high`.

**Set feature value**
```
Func CondFeature(port%, feat%{, val});
```

val      If present, it sets the value for the feature set by `feat%` and `port%`. If this is a continuous feature, `val` sets the new value. If the feature has n discrete setting, `val` should be 0 to n-1 to select the feature corresponding to the feature description in `list$[val]`. If `val` exceeds the allowed range, a continuous feature is set to the nearest allowed value and a discrete feature is unchanged.

Returns  The feature value at the time of the call (before any change). This is an integer index for features with discrete values otherwise it is the feature value.

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

# CondFilter()

This sets or gets the frequency of the low-pass or high-pass filter of the signal conditioner. See the `CondSet()` command for more details of conditioner operation.

```
Func CondFilter(port%, high%{, freq{, type%}});
```

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

high%   This selects which filter to set or get: 0 for low-pass, 1 for high-pass.

freq    If present, this sets the desired corner frequency of the selected filter. See the `CondSet()` description for more information. Set 0 for no filtering. If omitted, the frequency is not changed. The high-pass frequency must be set lower than the frequency of the low-pass filter, if not the function returns a negative code.

type%   Optional, taken as 1 if omitted. The filter type to use when setting the filter.

Returns The cut-off frequency of the selected filter at the time of call, or a negative error code. A return value of 0 means that there is no filtering of the type selected.

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

# CondFilterList()

This function gets a list of the possible filter frequencies of the conditioner. Conditioners that support continuous frequency ranges also supply a list of frequencies to match the list of frequencies shown in the conditioner control panel. See the `CondSet()` command for more details of conditioner operation.

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

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

high%   Selects which filter to get: 0 for low-pass, 1 for high-pass.

freq[]  An array of reals holding the cut-off frequencies of the selected filter. There is always a value of 0 meaning no filtering.

type%   Optional, taken as 1 if omitted. The filter type in the range 1 to the number of types (as returned by `CondFilterType()`).

Returns The number of filter frequencies (including 0) or a negative error code.

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

# CondFilterType()

This function returns information about the filter types supported by the conditioner for the low pass and high pass filters. For example, the CED 1902 mk IV supports a choice of filter types. There are three command variants:

### Get number of filter types
```
Func CondFilterType(port%, high%);
```

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

high%   Selects which filter to return information for: 0 for low-pass, 1 for high-pass.

Returns The number of filter types.

**Get filter type in use**

```
Func CondFilterType(port%, high%, 0);
```

Returns   The currently selected filter type, from 1 to the number of filter types.

**Get filter type information**

```
Func CondFilterType(port%, high%, type%{, &name${, &lower{, &upper}}});
```

type%   The filter number, from 1 to the number of available filters.

name$   If present, returned holding the name of the filter selected by `type%`.

lower   If present, returned as the lowest filter frequency (excluding 0, meaning 'off').

upper   If present, returned as the highest supported filter frequency.

Returns   The number of frequency values the filter can be set to (including 0) or 0 if the filter corner frequency can be set to any value in the range `lower` to `upper`.

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

# 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(), CondGet(), CondOffset(),
CondOffsetLimit(), CondRevision$(), CondSet(), CondSourceList(), CondType()
```

# CondGet()

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

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

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

`in%`    Returned as the zero-based index of the input signal source (see `CondSet()`).

`gain`    Returned as the ratio of output to input signal amplitude (ignoring filtering).

`offs`    A value added to the input waveform to move it into a more useful range. Offset is specified in user units and is only meaningful when DC coupling is used.

`low`    Returned as the cut-off frequency of the low-pass filter. A value of 0 means that there is no low-pass filtering enabled on this channel.

`hi`    Returned as the cut-off frequency of the high-pass filter. A value of 0 means that there is no high-pass filtering enabled on this channel.

`notch%`  Returned as 0 if the mains notch filter is off, and 1 if it is on.

`ac%`    Returned as 1 for AC or 0 for DC coupling.

`typeL%`  Optional integer variable returned holding the low-pass filter type number as described for `CondFilterType()`.

`typeH%`  Optional integer variable returned holding the high-pass filter type number.

Returns  0 if all well or a negative error code.

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

# 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 an empty string if there is no conditioner for the port.

<div style="background:#ccc">

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

</div>

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

Returns  A string describing the conditioner. Strings defined so far include:  "1902ssh", where ss is the 1902 ROM software version number and h is the hardware revision level; and "CYBERAMP 3n0 REV x.y.z" where n is 2 or 8.

**See also:**
```
CondFeature(), CondFilter(), CondFilterList(), CondFilterType(), 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 command sets the closest possible value to that requested. If it is important to know what has actually been set you should read back the values with CondGet() after setting them, or use the functions for reading specific values.

<div style="background:#ccc">

**Func CondSet(port%, in%, gain, offs {,low, high, notch%, ac%{, typeL%{, typeH%}}});**

</div>

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

in%  A conditioner has one or more signal sources. For example, the CED 1902 supports Grounded, Single ended, Normal Diff, Inverted Diff, etc. Conditioners of the same type may have different sources. To select a source, set in% to its zero-based index in the list returned by CondSourceList().

gain  This is the desired ratio of output to the input signal amplitude (ignoring the effect of any filtering). The actual gain depends on the capabilities of the signal conditioner, see CondGainList(). The gain range may be altered by the choice of signal source. For example, the 1902 Isolated Amp input has a build-in gain of 100. This command sets the nearest gain to the requested value.

offs  This is the desired value in user units to add to the input waveform to move it into a more useful range. Offsets are only meaningful with DC coupling. Different conditioners have different offset ranges, and the offset range may be altered by the choice of signal source, see CondOffsetLimit(). The command will set the nearest offset it can to the desired value.

low  If present and greater than 0, it is the desired corner frequency of the low-pass filter. Low-pass filters are used to reduce the high frequency content of the signal, both to satisfy the sampling requirement, and in case where it is known that no useful information is to be found in the signal above a certain frequency. If omitted, or 0, there is no low-pass filtering. The actual filter value set depends on the capabilities of the signal conditioner.

high  If present and greater than 0, it is the high-pass filter corner frequency. High-pass filters reduce the low-frequency content of the signal. This must be set lower than the frequency of the low-pass filter; if not, the function returns a negative code. If omitted, or set to 0, there is no high-pass filtering.

Different signal conditioners have different ranges of frequency filtering. To find out the real filter frequency set, use CondFilter(). CondFilterList() returns the list of possible filter frequencies.

notch%  Some signal conditioners have a mains-frequency notch filter (usually 50 Hz or 60 Hz) used to reduce the effect of mains interference on low level signals. This filter will remove the fundamental 50 Hz or 60 Hz signal; it will not remove higher harmonics (for example 150 Hz). If notch% is present with a value greater than 0, the notch filter is on. If omitted, or 0, the notch filter is off.

ac%     The 1902 supports both AC and DC signal coupling. If you set AC coupling you should probably set the offset to zero. If `ac%` is greater than 0, the signal conditioner is AC coupled. If omitted or 0, the signal conditioner is DC coupled.

typeL%  Optional value, taken as 1 if omitted, that sets the low-pass filter type as described for `CondFilterType()` in the range 1 to the number of filter types.

typeH%  Optional value, taken as 1 if omitted, that sets the high-pass filter type.

Returns  0 if all well or a negative error code.

**See also:**
```
CondFilter(), CondFilterList(), CondFilterType(), 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 on that port or 1 for a CED 1902, 2 for an Axon Instruments CyberAmp, 3 for a Power1401 with ADC gain controls and 4 for a Digitimer D360.

**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 -2pi to 2pi.

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

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

# Cosh()

This calculates the hyperbolic cosine of one value or an array of values.

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

x       The value, or a real array of values.

Returns When the argument is an array, the function replaces each value with its hyperbolic cosine and returns 0. When the argument is not an array the function returns the cosh of the argument.

**See also:**
```
Abs(), ATan(), Cos(), Exp(), Frac(), Ln(), Log(), Max(), Min(), Pow(), Rand(),
Round(), Sinh(), Sqrt(), Tanh(), Trunc()
```

# Cursor...()

# Cursor()

This function returns the x axis position of a vertical cursor, and can also move the cursor to a new position.

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

num%    The cursor number to use, cursor numbers run from 0 to 10.

where   If present, the new position of the cursor. If the new position is out of range of the x axis, it is limited to the x axis.

Returns The old cursor position or 0 if the cursor doesn't exist.

Examples:
```
Cursor(1,2.0);              'Set cursor 1 at position 2.0
where := Cursor(1);         'Get cursor position
```

**See also:**
```
ChanValue(), CursorDelete(), CursorLabel(), CursorLabelPos(), CursorNew(),
CursorRenumber(), CursorSet(), CursorValid(), HCursor()
```

# CursorActive()

This function sets and retrieves the parameters used by an active vertical cursor in searching for a feature in view data, it replaces the less elegant and now deprecated `CursorMode()`, `CursorActiveSet()` and `CursorActiveGet()` functions. There are two forms of the function, the first one sets the active cursor mode and parameters, the second is used to read the mode and parameters back.

```
Func CursorActive(num%{, mode%{, chan%, start|start$, end|end$ {,n%|thresh|
ref$|expr$ {,hyst {,width {,def|def$|min {,thresh2}}}}}}});
```

This form of the function is used to set active cursor modes and parameters. Note that the use of some of the parameters varies according to the cursor mode that is set. The function parameters are:

num%    The vertical cursor number for which we are setting the active mode parameters, from 0 to 10. If only this argument is provided the function returns the current cursor mode and makes no changes.

mode%   The active cursor mode (see the main active cursor documentation for a complete description of the modes). The possible values of `mode%` are:

| | | |
|---|---|---|
| 0 Static | 8 Falling threshold | 16 *Extreme slope* |
| 1 *Maximum* | 9 *Maximum slope* | 17 Turning point |

| | | |
|---|---|---|
| 2 *Minimum* | 10 *Minimum slope* | 18 *Percentage slope* |
| 3 *Extreme* | 11 Peak in slope | 19 *Repolarisation %* |
| 4 Peak find | 12 Trough in slope | 20 Expression |
| 5 Trough find | 13 *Slope threshold* | 21 Outside dual levels |
| 6 *Threshold* | 14 Slope +ve thresh | 22 Within dual levels |
| 7 Rising threshold | 15 Slope -ve thresh | 23 Point count |

Modes shown in italics are not available for use with cursor zero and are forced to Static mode. Modes above 3 apart from Expression and Point count are not allowed for log-binned data.

chan%     The channel number on which the cursor searches for features. This parameter is ignored for Static and Expression modes.

start     The start time for the feature search in seconds.

start$    The start time for the search as a string. Expressions such as `"XLow()+0.2"` can be used, numbers used in the string are assumed to be in seconds unless a specific modifier is used thus: `"XLow()+0.2ms"`.

end       The end time for the feature search, in seconds.

end$      The end time for the search as a string.

n%        This is the point count for Points count mode (mode 23).

thresh    The threshold level for searches for appropriate modes. This can also be given as a string expression where items such as `"HCursor(1)"` or `"Mean(0.1, 0.2)"` can be used.

ref$      The reference level for Extreme searches. For Repolarisation % searches (mode 19) this is the time expression that defines the time at which the 100% value is measured (this can also be given as a number in seconds). The 0% value is measured at the start time for the search.

expr$     The time expression string for Expression mode (mode 20).

hyst      The hysteresis for threshold crossing searches or the minimum amplitude value for peak and trough searches, for Percentage slope (mode 18) and Repolarisation % (mode 19) searches this is the percentage value.

width     The measurement or slope width value used in the feature search. For peak and trough searches, this sets the maximum width for the peak while for threshold crossings it is the delay after crossing parameter.

def       The default position for the cursor - this is the position the cursor moves to if the search fails. Default positions only apply to cursors numbers higher than zero and are not used for Expression mode. If you do not set a default position and the search fails, the cursor will become invalid (which prevents it being used for measurements), so you can use the default position to ensure that measurements are taken even if the active mode cursor search fails.

def$      The default position as a time expression string. Default positions only apply to cursors numbers higher than zero.

min       The minimum step - only used if this is cursor 0.

thresh2   The second threshold for Outside dual levels and Within dual levels searches (modes 21 and 22 respectively). This can also be given as a string expression.

Returns   The active cursor mode at the time that the function was called.

---

**Func CursorActive(num%{, item% {, &val$}});**

This form of the function is used to read back the active cursor mode and parameters. Note that the use of some of the parameters varies according to the cursor mode that is set. The function parameters are:

num%      The vertical cursor number for which to retrieve information, from 0 to 10.

item%     This specifies the parameter for which to retrieve the current value:

| | | | | | |
|---|---|---|---|---|---|
| -1 mode% | -2 chan% | -3 start$ | -4 end$ | -5 n% | -6 |

| -7 def$ | or -8 ref$ | -9 hyst | -10 width | -11 min | -12 |
|---------|-----------|---------|-----------|---------|-----|
| expr$ | | | | | |

the value returned when `item%` is equal to -7 is `expr$` if the cursor mode is Expression, `def$` otherwise. If `item%` is omitted it is treated as -1.

`val$`  This optional string variable will be updated with the parameter value as a string when `item%` is -3, -4, -6, -7, -8 or -12.

Returns  The numerical value of the selected value. For items -3, -4, -6, -7, -8 and -12 this value is generated by parsing the associated string so it will reflect the position of any cursors and data values used, if the string parsing fails zero is returned.

**See also:**
`ChanSearch(), Cursor(), CursorDelete(), CursorNew(), HCursorActive()`

## CursorActiveGet()

This function returns the parameters used by an active cursor in searching for a feature in the view data. Note that the use of some parameters varies according to the cursor mode set. You should check the cursor mode first since if it is Repolarisation % then the fifth argument should be a string and not a real variable. This function has been replaced by the rather more elegant `CursorActive()` which should be used for all new scripts.

```
Func CursorActiveGet(num%, chan%, start|start$, end|end$ {,n%|thresh|expr$|
ref${, hyst{, width{, def|def$|min{, thresh2}}}}});
```

`num%`  The cursor number from 0 to 10.

`chan%`  Returned holding the number of the channel on which the cursor operates.

`start`  Returned holding the start time for the feature search.

`start$`  Returned holding the start time for the search as a string. Search limits such as `"XLow() + 0.2"` can be correctly returned.

`end`  Returned holding the end time for the search.

`end$`  Returned holding the end time for the search as a string.

`n%`  The search point count when the cursor mode is 23.

`thresh`  Returned holding the threshold level used in the feature search.

`expr$`  Returned holding the expression string for expression (20) mode.

`ref$`  Returned holding the expression used in Repolarisation % (19) mode searches to define the time at which the 100% value is measured.

`hyst`  Returned holding the hysteresis value used in the feature search or the percentage value for Percentage slope (18) and Repolarisation %(19) mode searches.

`width`  Returned holding the slope width value used in the feature search.

`def`  Returned holding the default position if the search fails

`def$`  Returned holding the default position as a string.

`min`  Returned holding the minimum step for cursor 0 searches.

`thresh2` Returned holding the second threshold level for dual-level searches.

Returns  Zero.

**See also:**

Cursor(), CursorDelete(), CursorMode(), CursorNew(), CursorActive()

# CursorActiveSet()

This function sets the parameters used by an active vertical cursor in searching for a feature in view data. Note that the use of some of the parameters varies according to the cursor mode set, which means that you should take care to set the active mode first using CursorMode() and then use this function to set the active mode parameters. All time values, whether as a number or a string, are in seconds regardless of the X axis units that are in use. This function has been replaced by the rather more elegant CursorActive() which should be used for all new scripts.

```
Func CursorActiveSet(num%, chan%, start|start$, end|end$ {, n%|thresh|expr$|
ref${, hyst{, width{, def|def$|min{,thresh2}}}}});
```

num%  The vertical cursor number, from 0 to 10.

chan%  The channel number on which the cursor searches for features.

start  The start time for the feature search in seconds.

start$  The start time for the search as a string. Expressions such as "XLow()+0.2" can be used, again in seconds. In Signal version 3 this parameter was also used to set the expression in Expression mode, so to avoid incompatibilities with old scripts this parameter is also copied to the expression string if Expression mode is in use for the cursor in question.

end  The end time for the feature search, in seconds.

end$  The end time for the search as a string.

n%  This is the point count for Points count mode (mode 23).

thresh  The threshold level for searches for appropriate modes. This can also be given as a string.

expr$  The expression string for Expression mode (mode 20).

ref$  The expression used in Repolarisation % searches (mode 19) to define the time at which the 100% value is measured, this can also be given as a number. The 0% value is measured at the start time for the search.

hyst  The hysteresis for threshold crossing searches or the minimum amplitude value for peak and trough searches, for Percentage slope (mode 18) and Repolarisation % (mode 19) searches this is the percentage value.

width  The measurement or slope width value used in the feature search. For peak and trough searches, this sets the maximum width for the peak while for threshold crossings it is the delay after crossing parameter.

def  The default position for the cursor - this is the position the cursor moves to if the search fails. Default positions only apply to cursors numbers higher than zero. If you do not set a default position and the search fails, the cursor will become invalid (which prevents it being used for measurements), so you can use the default position to ensure that measurements are taken even if the active mode cursor search fails.

def$  The default position as a string. Default positions only apply to cursors numbers higher than zero.

min  The minimum step - only used if this is cursor 0.

thresh2  The second threshold for Outside dual levels and Within dual levels searches (modes 21 and 22 respectively). This can also be given as a string.

Returns  Zero.

**See also:**
ChanSearch(), Cursor(), CursorDelete(), CursorMode(), CursorNew(), CursorActive()

# CursorDelete()

Deletes a cursor. It is not an error to delete an unknown cursor (which has no effect).

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

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

Returns The number of the cursor deleted or 0 if no cursor was deleted.

**See also:**
```
Cursor(), CursorLabel(), CursorLabelPos(), CursorNew(), CursorRenumber(),
CursorSet(), HCursorDelete()
```

# CursorExists()

This function tests if a given vertical cursor exists at the time of the call.

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

num%    The cursor number from 0 to 10. For cursor zero, which can only be hidden not deleted, this will always return 1.

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

**See also:**
```
Cursor(), CursorDelete(), HCursorExists()
```

# CursorLabel()

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

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

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

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

form$   A user-defined label string with replaceable fields %p, %n and %v(chan) for position, number and channel value; chan is the channel number whose value you require. %w.dp and %w.dv(chan) formats are allowed where w and d are numbers that set the field width and number of decimal places.

Returns  A cursor style before any change as 0-4 if a single cursor is selected, or the view cursor style as 0-3. If style% is omitted or not 0-3, the current view cursor style is not changed.

**See also:**
```
Cursor(), CursorDelete(), CursorLabelPos(), CursorNew(), CursorRenumber(),
CursorSet(), HCursorLabel()
```

# CursorLabelPos()

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

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

num%    The cursor number. If the cursor does not exist the function does nothing and returns -1.

pos     If present, the command sets the label position as the percentage of the distance from the top of the cursor. Out-of-range values are set to the appropriate limit.

Returns  The cursor position before any change was made, or -1 if the cursor does not exist.

**See also:**
```
Cursor(), CursorDelete(), CursorLabel(), CursorNew(), CursorRenumber(), CursorSet(),
HCursorLabelPos()
```

# CursorMode()

This function lets you set and read the active cursor mode for a cursor. This function has been replaced by the rather more elegant `CursorActive()` which should be used for all new scripts.

**`Func CursorMode(num%{, mode%});`**

`num%`    The cursor number. If the cursor does not exist the function does nothing and returns -1.

`mode%`   If present, the command sets the new cursor mode (see the main active cursor documentation for a description of the modes). The possible values of `mode%` are:

| | | |
|---|---|---|
| 0  Static | 8  Falling threshold | 16  *Extreme slope* |
| 1  *Maximum* | 9  *Maximum slope* | 17  Turning point |
| 2  *Minimum* | 10  *Minimum slope* | 18  *Percentage slope* |
| 3  *Extreme* | 11  Peak in slope | 19  *Repolarisation %* |
| 4  Peak find | 12  Trough in slope | 20  Expression |
| 5  Trough find | 13  *Slope threshold* | 21  Outside dual levels |
| 6  *Threshold* | 14  Slope +ve thresh | 22  Within dual levels |
| 7  Rising threshold | 15  Slope -ve thresh | 23  Point count |

Modes shown in italics are not available for use with cursor zero and will be forced to static. Modes above 3 apart from Expression and Point count are not allowed for log-binned data.

Returns  The cursor mode before any change was made, or -1 if the cursor does not exist.

**See also:**
`ChanSearch(), Cursor(), CursorActive(), CursorNew(), CursorSet()`

# CursorNew()

This command adds a new cursor to the view at the designated position. A new cursor is created in Static mode (not active).

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

`where`   Where to position the cursor. In a file or memory view it is a time in seconds. In an XY view it is in x axis units. The position is limited to the x axis range. If the position is omitted, the cursor is placed in the middle of the window.

`num%`    If this is omitted, or set to -1, the lowest numbered free cursor is used. If this is a cursor number, that cursor is created. This must be a legal cursor number or -1.

Returns  It returns the cursor number as an integer, or 0 if all cursors are in use.

**See also:**
`Cursor(), CursorDelete(), CursorLabel(), CursorLabelPos(), CursorRenumber(), CursorVisible(), CursorSet(), HCursorNew()`

# CursorOpen()

This command reports on the open state or opens the cursor values and cursor regions dialogs for the current data or memory view. These dialogs open in the last used position unless not usefully on a monitor, when they open centred on the application window. If the dialog is open you can use this command to get the dialog handle and change settings. Use `FileClose()` to close an opened dialog. This function was added in Signal version 5.00.

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

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

`mode%`   Set 1 to open the dialog and show it, 0 or omitted to open and hide it. You may wish to open a window invisibly so that you can position it before it is displayed.

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

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

type% This sets the measurement type for the cursor regions dialog. Set 1-17 to set a measurement as for `ChanMeasure()`. Omit or set -1 or 0 for no change. See the documentation of the Cursor Regions window for details of these measurements.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | Curve area | 6 | Modulus | 11 | Standard deviation | 16 | Standard error |
| 2 | Mean | 7 | Maximum | 12 | Extreme | 17 | RMS error |
| 3 | Slope | 8 | Minimum | 13 | Peak | | |
| 4 | Area | 9 | Peak to peak | 14 | Trough | | |
| 5 | Sum | 10 | RMS amplitude | 15 | Point count | | |

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

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

# CursorRenumber()

This command renumbers the cursors from left to right in the view. There are no arguments.

`Func CursorRenumber();`

Returns  The number of cursors found in the view.

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

# CursorSearch()

This function causes active cursors in a data 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 `CursorSearch` causes all other vertical cursors to search if they are active.

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

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

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

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

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

# CursorSet()

This command deletes any existing vertical cursors, then positions a specified number of new cursors, equally spaced in the view and numbered in order from left to right. If any positions are given, they are applied. The cursor labelling style is not changed.

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

num%    The number of cursors in the range 0 to the maximum allowed. 0 turns off all the cursors. It is a run-time error to ask for more than the maximum or less than 0 cursors.

where   Optional cursor positions in x axis units. Positions that are out of range are set to the nearest valid position.

Examples:
```
CursorSet(0);        'Delete all cursors
CursorSet(2,20,30); 'remove cursors, set 2 at 20 and 30 on x axis.
```

**See also:**
```
BinToX(), Cursor(), CursorDelete(), CursorLabel(), CursorVisible(), CursorLabelPos(),
CursorNew(), CursorRenumber()
```

# CursorValid()

Use this function to test if the last search of a cursor 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.

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(), 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 or -1 for all vertical cursors.

show%   If present set this to 0 to hide the cursor and non-zero to show it, if not present no change is made.

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

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

# CursorX()

This command gets the position of a cursor before the last move. It is particularly useful with Active cursors. The same mechanism is available as a dialog expression. This command was added at Signal version 6.05.

```
Func CursorX(num%);
```

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

Returns The cursor position before the last move (however it was caused) or -1 if the cursor doesn't exist. Use Cursor(num%) to get the current position.

**See also:**

```
BinToX(),  XToBin(),  Cursor(),  CursorDelete(),  CursorLabel(),  CursorLabelPos(),
CursorRenumber(), CursorVisible(), CursorSet(), CursorNew()
```

# D

# Date$()

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

**Func Date$({dayF%, {monF%, {yearF%, {order%, {sep$}}}}});**

dayF%   This sets the format of the day field in the date. This can be written as a day of the week or the day number in the month, or both. The options are:

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

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

If you add 8 or 16, 1 is added automatically. If you request both the weekday name and the number of the day, the name appears before the number.

monF%   The format of the month field. This can be returned as either a name or a number. If this argument is omitted, the value 3 is used. The options are:

 0   No month field.
 1   Show name of the month: "`April`".
 2   Show number of month: "`04`"
 **3**   Show an abbreviated name of month: "`Apr`"
 4   Show number of month with no leading zeros: "`4`"

yearF%  The format of the year field. This can be returned as a two or four digit year.

 0   No year is shown
 **1**   Year is shown in two digits: "`98`".
 2   Year is shown in two digits with an apostrophe before it: "`'98`".
 3   Year is shown in four digits: "`1998`".

order%  The order that the day, month and year appear in the string.

 **0**   Operating system settings
 1   month/day/year
 2   day/month/year
 3   year/month/day

sep$   This string appears between the day, month and year fields as a separator. If this string is empty or omitted, Signal 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(), TimeDate(), Time$()
```

# Debug()

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

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

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

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

This command was included for use in situations such as student use, where it is important that the user cannot break out of a script by accident. It is advisable to test your script carefully before using this option. Once set, you cannot stop a looping script except by forcing a fatal error. Make sure you save your script before setting this option.

**See also:**
Eval()

# DebugHeap()

This script command is provided for use by CED engineers to help to debug system problems. It was added at version 4.02. It reports on the state of the application heap, used to dynamically allocate memory. The heap is a list of memory sections. Each section is described by its start address, its size, and if it is in use by the application or is free (available for use). When the application wants more memory, it asks the heap for it. If there is no suitable memory in the heap, the heap requests more memory from the system and uses this to create more heap sections.

The command has the following variants:

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

```
Func DebugHeap();
```

Returns   0 = heap OK, -1 = _HEAPBADBEGIN initial header information is bad or cannot be found, -2 = _HEAPBADNODE bad node or the heap is damaged, -3=_HEAPBADPOINTER a pointer into the heap is invalid, -4 = _HEAPEMPTY the heap has not been initialised, -5 = unknown error.

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

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

Returns   The same values as the call with no arguments.

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

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

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

Returns The same values as the call with no arguments.

## Get heap information

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

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

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

    0    The total number of memory sections in the heap.

    1    The number of used memory sections in the heap.

    2    The total size of memory controlled by the heap.

    4    The total size of memory that is in used and controlled by the heap.

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

Returns    The same values as the call with no arguments.

## Get heap entries

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

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

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

    0    0 if the section is unused, 1 if it is used.

    1    The start address of the section.

    2    The size of the section, in bytes.

    3    If present, returns the first 4 bytes of data in this section.

    4    If present, returns the next 4 bytes of data in this section.

    n    Returns further data (if it exists) in the section.

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

## Example

This example prints out the number of system handles in use and a synopsis of the heap usage. If you suspected that your script was causing a memory leak you could insert this procedure, then call it periodically. If there is a steadily increasing use of handles or heap space, check that you are closing all the views you have created.

```
proc ReportHeap()
var info%[4], stats%[32], i%;
DebugHeap(info%, stats%);
PrintLog("Handles=%d, Heap used=%d kB in %d fragments, %d Kb free in %d fragments\n",
 App(-4), info%[3]/1000.0, info%[1], (info%[2]-info%[3])/1000.0, info%[0]-info%[1]);
for i% := 0 to 15 do PrintLog("%6d", pow(2, i%)); next;
PrintLog("\n");
for i% := 0 to 15 do PrintLog("%6d", stats%[i%]); next;
PrintLog("\n");
for i% := 16 to 19 do PrintLog("%4dkB", pow(2, i%)/1024); next;
for i% := 20 to 29 do PrintLog("%4dMB", pow(2, i%)/(1024*1024)); next;
for i% := 30 to 31 do PrintLog("%4dGB", pow(2, i%)/(1024*1024*1024)); next;
PrintLog("\n");
for i% := 16 to 31 do PrintLog("%6d", stats%[i%]); next;
PrintLog("\n");
end;
```

## See also:

```
App(-4), Debug(), Eval(), DebugList(), DebugOpts()
```

# 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. Low numbered objects are the integers 0 to 20, higher-numbered items are the instructions that the script compiles to followed by objects that represent the built-in script commands.

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

`list%`  This determines what to list and also controls the accumulation of timing information.

-3  Disable the accumulation of timing information (the normal state) for calls to built-in functions.

-2  Enable the accumulation of timing information. This is normally used as a diagnostic of slow script performance when we need to figure out where the time is being spent.

-1  Reset accumulated times and call counts to 0.

0  List a summary of the `DebugList()` options in the Log view.

1  List the names of fixed objects (constants and operators). This is usually only of interest to CED programmers.

2  List the names of permanent objects (constants, operators and built-in commands).

3  List built-in commands (things like NextTime()).

>3  List information for the object with the index `list%`.

`opt%`  This optional argument (default value 0) sets the additional object information to list and is the sum of:

1  list the index number.

2  list the object type.

4  list timing information for the built-in script commands. The timing information is three numbers: the number of times the command was called, the total time in seconds used and the time per call in microseconds.

8  Only list timing information for built-in script commands that were called at least once (added in version 6.03).

16  Only list timing information for built-in script commands that were not called (added in version 6.03). We use this to check our test script coverage.

To use the timing information:

```
DebugList(-3);              'Disable timing
DebugList(-1);              'Reset call counts and times
DebugList(-2);              'Enable timing
... your code to be timed
DebugList(-3);              'stop accumulating times
PrintLog("Command name    Total calls     Seconds     us/call Index\n");
DebugList(3, 1+4+8);            'list indices and times for used built-in commands
```

Here is some typical output. The timing is from the point where the command arguments have been prepared up to the point where the command returns control to the script language. These times were measured on an AMD Athlon 1900+ rated processor.

```
Command name    Total calls     Seconds     us/call Index
...
Asc                  537      0.000512       0.953 530
Chr$                 668      0.001169       1.751 531
DelStr$               13      0.000030       2.299 532
InStr               1076      0.001263       1.174 533
LCase$               130      0.000230       1.769 534
Left$                334      0.000742       2.221 535
Mid$                3751      0.005480       1.461 536
Right$                79      0.000155       1.959 537
...
```

The timing information is normally used by CED programmers to check that functions are working with a reasonable efficiency and when attempting to find out where all the time is going in a script. Any time not accounted for by built-in commands is general script execution time (fetching instruction, running them, allocating

variables and so on). If you should discover that a particular built-in function is using a lot of time, you may be able to optimise your use of the function to improve matters. If you think that a particular function is slow, let us know; we may be able to improve it.

**See also:**
```
App(-4), Debug(), Eval(), DebugOpts(), DebugHeap()
```

# 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()
```

# DeleteFrame()

This function deletes the current frame from the current data view. Frames can only be deleted if they were appended and have not yet been saved to disk. It is not possible to delete the last frame in a memory view.

```
Func DeleteFrame();
```

Returns  Zero or a negative error code.

**See also:**
```
AppendFrame(), FrameCount(), FrameFlag(), FrameTag()
```

# DelStr$()

This function removes a substring from a string.

```
Func DelStr$(text$, index%, count%);
```

text$   The string to remove characters from. This string is not changed.

index%  The start point for the deletion. The first character is index 1. If this is greater than the length of the string, no characters are deleted.

count%  The number of characters to delete. If this would extend beyond the end of the string, the remainder of the string is removed.

Returns `DelStr$()` returns the original string with the indicated section deleted.

**See also:**
```
InStr(), LCase$(), Len(), Mid$(), Right$(), UCase$()
```

# Dialogs

You can define your own dialogs to get information from the user. You can define dialogs in a simple way, where each item of information has a prompt and the dialog is laid out automatically, or you can build a dialog by specifying the position of every item. A simple dialog has the structure shown in the diagram:

The dialog is arranged in terms of items. Unless you specifically request otherwise, the dialog items are stacked vertically above each other with buttons arranged at the bottom.

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

This form of dialog is very easy to program. There is no need to specify any position or size information, the system works it all out. Some users require more complicated dialogs, with control over the field positions. This is also possible, but more involved to program. You are allowed up to 1000 fields in a dialog, the DlgShow() script function is limited (as are all script functions) to 20 parameters so if you want more than 20 items in a dialog some have to be handled via array parameters to DlgShow.

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

## Steps to program a dialog

Generating and using a dialog using the script language is a three step process:

1.  Use DlgCreate() to start designing a dialog. The dialog starts with an OK and a Cancel button only, but you can change that in step 2.

2.  Add fields to the dialog with commands like DlgInteger(), DlgChan() and so on. You can add and modify buttons with DlgButton(). You can also configure how the dialog behaves with DlgAllow() and set mouse interactions with DlgMouse().

3.  Use DlgShow() with a list of arguments, one per dialog field, to get and set the editable field contents. As there is a limit on the number of arguments allowed in the script language, you can also use arrays as arguments; an array with n elements matches n fields in the dialog.

## Dialog units

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

## Simple dialog example

The simple example dialog shown above can be created by this code:

```
var ok%, item1%, item2%, item3, item4$:= "more...", item5;
DlgCreate ("Title for the dialog");          'start new dialog
DlgInteger(1,"Item &1 prompt",0,10,0,0,1); 'range 0-10,spinner
DlgChan    (2, "Item &2 prompt", 1);          'Waveform channel list
DlgReal    (3, "Item &3 prompt", 1.0, 5.0); 'real, range 1.0-5.0
DlgString  (4, "&More prompts...", 6);        'string, any characters
DlgCheck   (5, "A &check box with a prompt");'a check box item
DlgButton  (2, "&Extra");                     'extra button, number 2

ok% := DlgShow(item1%, item2%, item3, item4$, item5); 'show dialog
```

## Prompts, & and tooltips

In the functions that set an item with a prompt, if you precede a character in the prompt with an ampersand (&), the following character is underlined and is used by Windows as a short-cut key to move to the field or activate the button. All static dialog items except a group box allow you define a tooltip by appending a vertical bar followed by the tooltip text to the text$ argument. For example:

```
DlgReal(3, "Rate|Enter the sample rate in Hz", 100, 500);
```

Buttons allow you to specify an additional activation key and an optional tooltip by adding a vertical bar followed by the key code and then another vertical bar followed by the tooltip. See the `label$` argument of `ToolbarSet()` for details of key codes.

### More complex example

This example shows how to respond to user actions within a dialog. In this case we use a check box to enable and disable a group of items and a button that displays the current values of dialog items. The numbered fields are:

1   An integer, range 0-10 with a spinner
2   A drop list of 4 items
3   A check box, used to enable items 4 and 5
4   A real number with a spinner
5   A string with a drop down list of items

We have added button 2 (buttons 0 and 1 are Cancel and OK) and a group box around items 4 and 5. To make room for the group box, the y positions of items 4 and 5 are set explicitly.

With `DlgAllow()` we have set `Func Change%(item%)` to be called whenever the user changes a selection or check box or when an editable field loses the input focus. The `item%` argument is set to the item number that changed or to 0 if the dialog is appearing for the first time. We are interested in item 3, the check box, and we use the state to enable or disable the group box and the items inside it.

`Func Current%()` is linked to the `"Current"` button. In this case it is used to display a message box that lists the current values of items in the dialog.

```
var ok%, item1%, item2%, item3%, item4, item5$:= "Text", gp%;
DlgCreate("Dialog with user placement",0,0,40,7.5);
DlgInteger(1,"Integer 0 to 10",0,10,0,1,1);      'Int with spinner
DlgList(2,"List item","List 0|List 1|List 2|List 3", 4, 0, 2);
DlgCheck(3, "check box enabling items",0,3);      'check box item
DlgReal(4, "Real 1 to 5",1.0,5.0,0,4.5,0.5);      'Real with spinner
DlgString(5, "String length 10",10,"",0,5.5,      'String item with
            "String 1|String 2|String 3");        'drop down list
DlgButton(2, "Current", Current%);                'button+function
DlgAllow(0x3ff, 0, Change%);  'Allow all, no idle, change function
gp% := DlgGroup("Extra items",1,3.8,-1,2.9);      'Group box
ok% := DlgShow(item1%,item2%,item3%,item4,item5$);
Halt;

Func Change%(item%)
var v%;
docase
  case ((item% = 3) or (item% = 0)) then    '0 is initial setup
    v% := DlgValue(3);          'get check box state
    DlgEnable(v%, gp%, 4, 5); 'enable groupbox+items 4, 5
endcase;
return 1;                      'Return 1 to keep dialog running
end;

Func Current%()
var v1%, v2%, v3%, v4, v5$;
v1% := DlgValue(1);            'Retrieve the current values
v2% := DlgValue(2); v3% := DlgValue(3);
v4  := DlgValue(4); v5$ := DlgValue$(5);
Message("Values are %d, %d, %d, %g and %s",v1%,v2%,v3%,v4,v5$);
return 1;                      'Return 1 to keep the dialog running
end;
```

### See also:
DlgAllow(), DlgButton(), DlgChan(), DlgCheck(), DlgCreate(), DlgEnable(), DlgGroup(), DlgInteger(), DlgLabel(), DlgList(), DlgReal(), DlgShow(), DlgSlider(), DlgString(), DlgText(), DlgValue(), DlgVisible(), DlgXValue()

# DlgAllow()

Call this function after `DlgCreate()` and before `DlgShow()` to enable dialog idle time processing, advanced call-back features and dynamic access to the dialog fields. There are no restrictions on what call-back functions can do. However, it is not sensible to place time-consuming code in an idle call-back function or to do anything other than check dialog fields and possible display a warning message in a dialog-item-change function. Call-back functions use `DlgValue()`, `DlgEnable()` and `DlgVisible()` to manipulate the dialog fields. You can also call `DlgAllow()` after `DlgShow()` in which case it will operate on the most recently created dialog. `DlgAllow()` is new in version 3.

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

allow% A code that specifies the actions that the user can and cannot take while interacting with Signal. The code is the sum of possible activities:

|      |        |                                                   |
|-----:|--------|---------------------------------------------------|
|    1 | 0x0001 | User may swap to other applications               |
|    2 | 0x0002 | User may change the current window                |
|    4 | 0x0004 | User may move and resize windows                  |
|    8 | 0x0008 | User may use File menu                             |
|   16 | 0x0010 | User may use Edit menu                             |
|   32 | 0x0020 | User may use View menu                             |
|   64 | 0x0040 | User may use Analysis menu                         |
|  128 | 0x0080 | User may use Cursor menu and add cursors           |
|  256 | 0x0100 | User may use Window menu                           |
|  512 | 0x0200 | User may use Sample menu                           |
| 1024 | 0x0400 | User may not double click y axis                  |
| 2048 | 0x0800 | User may not double click the x axis or scroll it |
| 4096 | 0x1000 | User may not change channel of horizontal cursors |
| 8192 | 0x2000 | User may not change to another frame              |

A value of 0 would restrict the user to inspecting data and positioning cursors in a single, unmoveable window, but being able to switch frames. A value of 8192 is the same but without changing frames.

id%() This is an integer function with no arguments. Use the name with no brackets, for example `DlgAllow(0,Idle%);` where `Func Idle%()` is a script function. When `DlgShow()` executes, the function is called repeatedly in system idle time, as for the `ToolbarSet()` idle function unless a change function, a button function or a mouse function is active.

If the function return value is greater than 0, the dialog remains open. A zero or negative return value closes the dialog and `DlgShow()` returns the same value.

If this argument is omitted or 0, there is no idle time function.

ch%() This is an integer function with one integer argument, for example `Func Changed%(item%)`. You would use `DlgAllow(0,0,Changed%);` to link this function to a dialog. Each time the user changes a dialog item, Signal calls the function with the argument set to the changed item number. There is an initial call with the argument set to 0 when the dialog is about to be displayed.

A field is deemed to change when the user clicks a check box or changes a selection in a list or moves the focus from an editable item after changing the text. For real and integer values, the new value must be in range.

If the change function returns greater than 0, the change is accepted. If the return value is zero, the change is resisted and the focus set back to the changed item. If the return value is negative, the dialog closes and `DlgShow()` returns this value and the arguments are not updated.

There is an example of use here.

**See also:**

DlgCreate(), DlgEnable(), DlgShow(), DlgValue(), DlgValue$(), DlgVisible(),
Interact(), ToolbarSet()

# DlgButton()

Dialogs created by `DlgCreate()` have Cancel and OK buttons; this function adds, deletes and changes buttons. You can link a script function to a button and use the function return value to decide if the dialog should close. Use this function after `DlgCreate()` and before `DlgShow()`. You can also call this function with no arguments to get the number of the last button that was pressed (added at version 4.06). There are two command variants:

```
Func DlgButton(but%, text${, func ff% {, x, y}});
Func DlgButton();
```

but%   The button number from 0 to 199. Button 0 is the cancel button, 1 is the OK button. Button numbers higher than 1 create new buttons.

text$   This sets the button label. Set an empty string to delete a button. You cannot delete button 1; the label is set back to OK if you try. The label text can be followed by an optional key code and an optional tooltip separated by vertical bars. See the `label$` argument of `ToolbarSet()` for details of the format.

If you set a key code, the button can be activated even when the dialog does not have the input focus as long as it is the topmost user dialog and you have not created a toolbar or interact bar from a function linked to the dialog. This allows you to drag cursors in a window, then use the key code without the need to click in the dialog to activate it first.

ff%()   This is an integer function with no arguments that is called when the button is used. Set the argument to zero or omit it if you don't want a button function, in which case clicking the button closes the dialog, `DlgShow()` returns the button number and the `DlgShow()` arguments are updated for all buttons except 0.

If you supply a function, it is called each time the button is used and the function return value determines what happens next:

<0   The button acts as Cancel. The dialog closes, `DlgShow()` returns this value and its arguments are not updated.

0   The button acts as OK. The dialog closes and the `DlgShow()` return value is the button number and its arguments are updated.

>0   The dialog continues to display.

The button function can use `DlgEnable()`, `DlgValue()` and `DlgVisible()` and can also create and show subsidiary dialogs.

x,y   Set the button position in dialog units, both or neither of these must be supplied. If x is positive it sets the position of the left edge of the button relative to the left side of the dialog, if negative it sets the position of the right edge of the button relative to the right side of the dialog. If the button position is not supplied it will be automatically positioned at the bottom of the dialog in rows based on the button numbers.

Returns   The call with no arguments returns the number of the last button that was pressed. This allows you to service multiple buttons with the same user-defined button function. The call that creates buttons returns 0.

**Example**
```
func DoButton%()
Message("Button %d", DlgButton()); 'report button number
return 1; ' leave dialog open
end;

DlgCreate("Example of button replacement");
DlgText("This dialog demonstrates using a button", 0, 1);
DlgButton(0, "");    ' remove the Cancel button
DlgButton(2, "My &Button||Press to say hello", DoButton%);
DlgShow();  ' no arguments as no variable fields defined
```

There is another example of use here.

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

# DlgChan()

This function defines a dialog entry that lists channels that meet a specification to use for selection of a channel of a particular type. For simple dialogs, the `wide`, `x` and `y` arguments are not used. Channel lists are checked or created when the `DlgShow()` function runs. If the current view is not a data or XY view, the list will be empty.

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

`item%`   This sets the item number in the dialog.

`text$`   The text to display as a prompt. If the prompt contains a vertical bar, "|" any following text will be used as a tooltip and displayed when the mouse pointer is held over the item.

`wide`   This is an alternative to the prompt. It sets the width of the box in which the user selects a channel. If the width is not given the number entry box has a default width of the longest channel name in the list or 12, whichever is the smaller.

`mask%`   This is an integer code that determines the channels to be displayed. It is ignored for XY views. You can select channels of particular types by adding together the following codes:

| | | |
|---|---|---|
| 1 | 0x1 | Waveform or result view channels |
| 2 | 0x2 | Marker channels |
| 4 | 0x4 | Idealised trace channels |
| 8 | 0x08 | Real marker channels |

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

| | | |
|---|---|---|
| 128 | 0x80 | Exclude CFS data file channels (means idealised trace and virtual channels only) |
| 256 | 0x100 | Exclude virtual channels |
| 1024 | 0x400 | Exclude visible channels |
| 2048 | 0x800 | Exclude hidden channels |
| 4096 | 0x1000 | Exclude selected channels |
| 8192 | 0x2000 | Exclude non-selected channels |

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

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

`list%`   As an alternative to a mask, you can pass in a channel list (as constructed by `ChanList()`). This must be an array of channels, with the first element of the array holding the number of channels in the list.

`x`   If omitted or zero, the selection box is right justified in the dialog box, otherwise positive values set the position of the left end of the channel selection box in dialog units and negative values set the position of the right hand end relative to the right edge of the dialog. When positioning the left edge using a positive value, a channel item will be positioned approximately 1 dialog unit further to the left than you might expect - this is to take account of the control border.

`y`   If omitted or zero, this takes the value of `item%`. It is the position 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:**
DlgAllow(), DlgButton(), DlgCheck(), DlgCreate(), DlgInteger(), DlgLabel(),
DlgList(), DlgReal(), DlgValue(), DlgValue$(), 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 prompt to display, optionally followed by a vertical bar and tooltip text.

`x,y`    The position of the check box in dialog units. If omitted, x is set to 2 and y to `item%` and the check box behaves exactly like other dialog items. If x is positive is sets the position of the left side of the check box and if negative it sets the position of the right hand end of the prompt text relative to the right edge of the dialog.

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

There is a simple example here.

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

# DlgCreate()

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

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

`title$` A string holding the title for the dialog.

`x,y`    Optional, taken as 0 if omitted. The position of the top left hand corner of the dialog. The positions are in percentages of the screen size. The value 0 means centre the dialog. Values out of the range 0 to 95 are limited to the range 0 to 95.

`wide`    The width of the dialog in dialog units. If this is omitted, or set to 0, Signal works out the width for itself, based on the items in the dialog.

`high`    The height of the dialog in dialog units. If omitted, or set to 0, Signal works it out for itself, based on the dialog contents.

`help`    This is a string or numeric identifier that identifies the help page to be displayed if the user requests help when the dialog is displayed. String identifiers should correspond to entries in the Signal help file index.

`scr%`    Optional screen selector. See `Window()` command for details.

`rel%`    Omit or set 0 for application window relative, 1 for screen or desktop relative.

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

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

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

**See also:**
DlgAllow(), DlgButton(), DlgChan(), DlgCheck(), DlgInteger(), DlgLabel(), DlgList(),
DlgReal(), DlgShow(), DlgString(), DlgText(), Window()

# DlgEnable()

This function is used from a dialog call-back function to enable or disable dialog items, it cannot be used in other circumstances. Disabled items are drawn in grey, and will not respond to mouse clicks, keypresses or any other actions. There are two versions of this command. With a single argument, it returns the enabled state of an item; with two or more arguments it sets the enabled state of one or more dialog items. When you enable or disable an item, any prompt or spin controls associated with the item are also enabled or disabled. When applied to an image item, disable hides the image and enable shows it.

```
Func DlgEnable(en%, item%|item%[]{, item%|item%[]...});
Func DlgEnable(item%);
```

en%     Set 0 to disable list items, 1 to enable them and 2 to enable and give the first item the input focus. Input focus changes should be used sparingly to avoid user confusion; they can cause button clicks to be missed.

item%   An item number or an array of item numbers of dialog elements. The item number is either the number you set, or the number returned by `DlgText()` or `DlgGroup()`, or `-button`, where `button` is the button number. You cannot access prompts separately from their items as this makes no sense.

Returns  When called with a single argument it returns the enabled state of the item, otherwise it returns 0.

There is an example of use here.

**See also:**
`DlgAllow()`, `DlgButton()`, `DlgCreate()`, `DlgShow()`, `DlgValue()`, `DlgValue$()`, `DlgVisible()`

# DlgFont()

Before Signal version 6, the font used in user-defined dialogs was the system ANSI_VAR_FONT, which is usually MS Sans Serif 8pt. This is a proportionally spaced raster font, which does not look as nice as a TrueType or ClearType font as it does not take advantage of the ability to use fractional pixel positions on modern displays. However, by sticking with this font, user-defined dialogs look the same and layout the same across all windows platforms.

The `DlgFont()` script command gives you two more choices of font for user-defined dialogs (including `Message()` and `Query()`), but with the penalty that you cannot be certain that complex user-defined dialogs will layout in the same way in different versions of Windows or with different user options. The font used by a user-defined dialog is set when the `DlgShow()` command is executed, so `DlgFont()` must be run before this. Each time a script starts the default font is reset, equivalent to DlgFont(0). It is possible that in the future we will provide a Preferences option to change the default.

```
DlgFont({font%})
```

font%   An optional argument that if present sets the desired font. If omitted, no change is made. There are three font options:

0   This selects the original ANSI_VAR_FONT. This is automatically set each time you run a script.

1   This sets the same font that Signal uses for dialogs (such as the About Signal dialog). This is usually an 8 pt TrueType font (often Tahoma), so will look sharper than option 0 and will usually have a similar spacing.

2   This sets the font that Windows is using for system dialogs, it is fairly similar to the Signal dialog font (option 1) but appears to have a greater line spacing.

Returns  The font selection at the time of the function call.

Setting option 1 will usually not cause too much difference in layout and the result should look better than option 0. Setting option 2 is more problematic as the default since Vista is Segoe UI 9 pt, and this is likely to cause layout incompatibilities, especially if you make complex dialogs.

**See also:**
`DlgCreate()`, `DlgShow()`, `Query()`, `Message()`

# DlgGetPos()

This can only be used from a dialog call-back function to get the position of the top left corner of the dialog. The call-back functions are set by `DlgAllow()` and `DlgButton()`. To get the final dialog position you need to have call-backs for any button press that would close the dialog, or in an idle routine. This function was added at Signal version 4.01.

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

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

scr%     Optional screen selector for views, dialogs and the application window. If omitted or -1, positions are relative to the application window. Otherwise, 0 selects the entire desktop rectangle and greater values select a particular screen rectangle (but see rel%). See System() for more screen information.

rel%     Ignored unless scr% is 0 or greater. Set 0 or omit for positions relative to the intersection of the rectangle set by scr% and the application window, 1 for positions relative to the scr% rectangle. If there is no intersection, x and y are both returned holding zero.

Returns  1 if the position was returned, or -1 if the rectangle set by scr% and rel% is of zero size.

**See also:**
DlgCreate(), DlgButton(), DlgEnable(), DlgShow(), DlgValue(), DlgValue$(),
DlgVisible()

# DlgGroup()

This routine creates a group box with a dialog; a rectangular frame with a text label at the top left corner. You can use this after `DlgCreate()` and before `DlgShow()`. There is nothing for the user to edit in this item, so you do not supply an item number and there is no matching argument in `DlgShow()`. However, the returned number is an item number (higher than the values used to match items to `DlgShow()` arguments) that you can use in call-back functions to identify the group box.

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

text$    The text to display at the top left of the group box.

x,y      The position of the top left corner of the group box.

width    If positive, the width of the group box. If negative, this is the offset of the right hand side of the group box from the right hand edge of the dialog.

height   The height of the group box.

Returns  The routine returns an item number so that you can refer to this in call-back functions to use DlgVisible(), DlgValue$() and DlgEnable().

There is an example of use here.

**See also:**
DlgCreate(), DlgEnable(), DlgShow(), DlgValue$(), DlgVisible()

# DlgImage()

Call this function after `DlgCreate()` and before `DlgShow()` to add an image to a dialog.

```
Func DlgImage(bmp$, x, y, width, height);
```

bmp$     The path to a suitable file holding an image. Images are read from `.bmp`, `.jpg`, `.jpeg`, `.png`, `.tif` or `.tiff` files on disk. The image is stretched to fit in the rectangle set by the x, y, width and height arguments.

x,y      The position of the top left corner of the bitmap.

width    If positive, the width of the image in dialog units. If negative, this is the offset of the right hand side of the image from the right hand edge of the dialog.

`height`  The height of the bitmap in dialog units.

Returns  The routine returns an item number so that you can refer to this in call-back functions to use `DlgVisible()` and `DlgEnable()`.

**See also:**
`DlgCreate(), DlgEnable(), DlgShow(), DlgVisible()`

# DlgInteger()

This function defines a dialog entry that edits an integer with an optional spin control or drop down list of selectable items. The numbers you enter may not contain a decimal point. For simple dialogs, the `wide`, `x`, `y`, `sp%` and `li$` arguments are not used.

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

`item%`  This sets the item number in the dialog in the range 1 to the number of items.

`text$`  The prompt to display, optionally followed by a vertical bar and tooltip text.

`wide`  This is an alternative to the prompt. It sets the width in dialog units of the box in which the user types the integer. If the width is not given the number entry box has a default width of 11 digits (or width needed for the number range?).

`lo%`  The start of the range of acceptable numbers.

`hi%`  The end of the range of acceptable numbers.

`x`  If omitted or zero, the number edit is right justified in the dialog box, otherwise positive values set the position of the left end of the number edit in dialog units and negative values set the position of the right hand end relative to the right edge of the dialog. When positioning the left edge using a positive value, an integer item will be positioned approximately 1 dialog unit further to the left than you might expect - this is to take account of the edit field border.

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

`sp%`  If present and non-zero, this adds a spin box with a click increment of `sp%`.

`li$`  If present, this argument is a list of items separated by vertical bars that can be selected into the integer field, for example `"1|10|100"`.

The variable passed into `DlgShow()` should be an integer. The field starts with the value of the variable if it is in the range. Otherwise, it is limited to the nearer end of the range.

**See also:**
`DlgAllow(), DlgButton(), DlgChan(), DlgCheck(), DlgCreate(), DlgLabel(), DlgList(), DlgReal(), DlgValue(), DlgValue$(), DlgShow(), DlgSlider(), 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 prompt to display, optionally followed by a vertical bar and tooltip text.

`x`  If omitted or zero, the label is right justified in the dialog box, otherwise positive values set the position of the left end of the label in dialog units and negative values set the position of the right hand end relative to the right edge of the dialog.

`y`  If omitted or zero, this takes the value of `item%`. It is the position of the label 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(), DlgValue$(),
DlgShow(), DlgString(), DlgText(), DlgXValue()
```

# DlgList()

This defines a dialog item that is 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$|wide, const list$[]|str${, n%{, x{, y}}});
```

`item%`  This sets the item number in the dialog.

`text$`  The text to display as a prompt. If the prompt contains a vertical bar, "|" any following text will be used as a tooltip and displayed when the mouse pointer is held over the item.

`wide`  This is an alternative to the prompt. It sets the width of the box in which the user selects an item. If the width is not given the number entry box has a default width of the longest string in the list or 18, whichever is the smaller.

`list$`  An array of strings. These hold the items to be presented in the list. Each string should not be over 18 characters long, or it will be truncated.

`str$`  An alternative way to define the items to be presented in the list. The single string holds all of the items, items are separated by the vertical bar character (|). Again, items should not be more than 18 characters long.

`n%`  The number of entries in the list. If this is omitted, or if it is larger than the array, then the size of the array is used. If the `str$` form of the command is used, the number of items in the string sets the maximum number.

`x`  If omitted or zero, the list selection box is right justified in the dialog box, otherwise positive values set the position of the left end of the list selection box in dialog units and negative values set the position of the right hand end relative to the right edge of the dialog. When positioning the left edge using a positive value, a list item will be positioned approximately 1 dialog unit further to the left than you might expect - this is to take account of the control border.

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

The result obtained from this is the index into the list of the list element chosen. The first element is number 0. The variable passed to `DlgShow()` for this item should be an integer. If the value of the variable is in the range 0 to n %-1, this sets the item to be displayed, otherwise the first item in the list is displayed.

The following example shows how to set a list:

```
var list$,ok%,which%:=0;       'string list, test for OK, result
list$ := "zero|one|two|three"; 'these are the choices
DlgCreate("List example");     'Start the dialog
DlgList(1,"Make your choice", list$);
ok% := DlgShow(which%);        'Display dialog, wait for user
```

**See also:**
```
DlgAllow(), DlgButton(), DlgChan(), DlgCheck(), DlgCreate(), DlgInteger(),
DlgLabel(), DlgReal(), DlgValue(), DlgValue$(), DlgShow(), DlgString(), DlgText()
```

# DlgMouse()

There are two variants of the command; the first sets the initial mouse pointer position when the dialog opens - use after DlgCreate() and before DlgShow().  If you do not use this command variant, the mouse pointer is not moved when the dialog opens. This command can be particularly useful if you are using multiple screens.

```
Proc DlgMouse(item%);
```

item%  The item number of an element of the dialog at which to position the mouse pointer when the dialog opens. This is either the number you set for the dialog item, or the number returned by DlgText() or DlgGroup(), or -button, where button is the button number.

The second variant gives you access to the mouse positions and left button mouse clicks in Data and XY views when the mouse is over a data channel while a dialog is active. This functions in exactly the same way as the ToolbarMouse() command, so see there for full details and a description of the mouse down, up and move functions. The command arguments are:

```
Proc DlgMouse(vh%, ch%, mask%, want%, Func Down%{, Func Up%{, Func Move%}});
```

**See also:**

DlgButton(), DlgCreate(), DlgGroup(), DlgShow(), DlgText(), DlgVisible(), ToolbarMouse()

# DlgProgress

This routine creates a progress bar within a dialog; a rectangular area that is normally used to give a visual impression of what fraction of a task has been completed. You can use this after calling DlgCreate() and before DlgShow(). There is nothing for the user to edit in this item, so you do not supply an item number and there is no matching argument in DlgShow().

The returned value is an item number (above the values used to match items to DlgShow() arguments) that you can use in call-back functions to identify the progress bar. The progress bar appearance is purely controlled by the values it is set-to using the DlgValue() function. To make sensible use of this progress bar you will probably need to have an idle time function that makes calls to DlgValue() to indicate progress through the operation.

This command was added in Signal version 7.01.

```
Func DlgProgress(lb, rt, flags%, x, y, width{, height});
```

lb      Sets the values corresponding to the left/bottom end of the progress bar. lb can be greater than rt but must not be the same. When you use DlgValue() to set the progress, values in the range lb to rt cause the progress bar to fill from left to right.

rt      Sets the values corresponding to the right/top end of the progress bar. lb can be greater than rt but must not be the same.

flags%      This sets display options and is the sum of:

         1      Allow tooltip value readout of the bar position when the mouse hovers over the progress bar.

         2      Use an old-style progress bar display where the bar is broken up into a series of blocks; this appears to have no effect in some versions of Windows.

         4      Treat all progress bar values as integers, so ignore fractional parts of lb, rt and values passed with DlgValue().

         8      Set a progress bar that indicates progress vertically from bottom to top; without this it runs horizontally from left to right.

x      This sets the horizontal position of the left of the progress bar relative to the left-hand side of the dialog in dialog units. If this value is negative it sets the position of the right edge of the bar relative to the right-hand side of the dialog

y      This sets the vertical position of the top of the progress bar in dialog units.

width      If positive, this sets the width of the progress bar. If negative, this is the offset of the right hand side of the progress bar from the right hand edge of the dialog.

height      The height of the progress bar, if this is omitted then a reasonable default value will be used.

Returns      The routine returns an item number so that you can refer to this in call-back functions in order to use DlgVisible() and DlgValue().

**Tooltips**

If you enable the tooltip value readout option in flags%, the current position of the progress bar appears in the tip display when the mouse is over the bar.

**See also:**

```
DlgChan(), DlgCheck(), DlgCreate(), DlgInteger(), DlgLabel(), DlgList(), DlgReal(),
DlgShow(), DlgString(), DlgText(), DlgValue(), DlgValue$(), DlgXValue()
```

# DlgReal()

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

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

item%   This sets the item number in the dialog in the range 1 to the number of items.

text$   The prompt to display, optionally followed by a vertical bar and tooltip text.

wide    This is an alternative to the prompt. It sets the width in dialog units of the box in which the user types a real number. If `wide` is not given the box has a default width of 12 digits.

lo,hi   The range of acceptable numbers.

x       If omitted or zero, the number edit is right justified in the dialog box, otherwise positive values set the position of the left end of the number edit in dialog units and negative values set the position of the right hand end relative to the right edge of the dialog. When positioning the left edge using a positive value, a real item will be positioned approximately 1 dialog unit further to the left than most other controls - this is to take account of the edit field border.

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

sp      If present and non-zero, this adds a spin box with a click increment of `sp`. You can change this value dynamically with `DlgValue()`.

li$     If present, this argument is a list of items separated by vertical bars that can be selected into the editing field, for example `"1.0|10.0|100.0"`.

pre%    If present this sets the number of significant figures to use to represent the number, in the range 6 (the default) to 15.

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

**See also:**
```
DlgChan(), DlgCheck(), DlgCreate(), DlgInteger(), DlgLabel(), DlgList(), DlgShow(),
DlgSlider(), DlgString(), DlgText(), DlgValue(), DlgValue$(), DlgXValue()
```

# DlgShow()

This function displays the dialog you have built and returns values from the fields identified by item numbers, or makes no changes if the dialog is cancelled. Once the dialog has closed, all information about it is lost. You must create a new dialog before you can use this function again.

```
Func DlgShow(&item1|item1[], &item2|item2[], &item3|item3[] ...);
```

item    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. You can have up to 20 item parameters, if you want more than 20 items in your dialog some item parameters must be arrays; an array with n elements matches n fields in the dialog.

        The item 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 type of the items in the dialog.

Returns  0 if the user clicked on the Cancel button, or 1 if the user clicked on OK.

If the user clicks on OK, all the variables are updated to their new values. If the user clicks on Cancel, the variables are not changed.

**See also:**

DlgChan(), DlgCheck(), DlgCreate(), DlgInteger(), DlgLabel(), DlgList(), DlgReal(),
DlgString(), DlgText(), DlgXValue()

# DlgSlider()

This command adds a slider control that sets an integer value between two user-defined limits. The slider is horizontally orientated for simple use but can be vertically orientated. This command was new in Signal version 5.02.

```
Proc DlgSlider(item%, text$|wide, lb, rt{, iTick{, flags%{, x{, y}}}});
```

item%  This sets the item number in the dialog in the range 1 to the number of items.

text$   A left-justified prompt to display, optionally followed by a vertical bar and tooltip text. If text$ is used, the slider will be horizontal and fill the space from the end of the prompt to the right hand side of the dialog.

wide   This is an alternative to the prompt. If positive, the slider is horizontal and the value sets the slider width in dialog units; a zero value uses the entire dialog width. A negative value sets the height of a vertical slider and you will need to provide the x and y values to complete the positioning.

lb,rt   Sets the values corresponding to the left/bottom and right/top end of the slider. lb can be greater than rt but must not be the same.

iTick   Optional, with a default value of 0. Enables ticks if not zero, values greater than 0 set the tick spacing and negative values set tick auto-scaling in a 1, 2, 5 sequence.

flags%  This sets display options and is the sum of: 1=tooltip value readout during drag, 2=change notifications during drag operation, 4=round all slider values to integers.

x       If omitted or zero, the slider is right justified in the dialog. Otherwise positive values set the position of the left side of the slider relative to the left-hand side of the dialog in dialog units and and negative values set the right side position relative to the right-hand side of the dialog.

y       If omitted or zero, this takes the value of item%. It is the vertical position of the slider control in dialog units.

The variable passed into DlgShow() as argument number item% should be a real. The slider starts with the value of the variable if it is in the range. Otherwise, it is limited to the nearer end of the range. Note that the range of values that a slider can return is quantised by the pixel positions that the slider occupies but the returned variable only changes if the slider position is moved or if the original value is out of the slider range.

### Tooltips
If you enable the tooltip value readout option in flags%, the position of the slider in the control appears in any tip when the mouse is over the slider. If you have set a tip with the text$ argument, this tip appears when the mouse is over the prompt. If you add 4 to flags%, all displayed positions are integral.

### Change function
If you have used DlgAllow() to set a change function, you can choose if this is called every time the slider position changes during a drag operation, or only when the drag operation ends.

### Keyboard control
The cursor left, right, up and down keys move the slider when it is has the keyboard focus.

### See also:
DlgChan(), DlgCheck(), DlgCreate(), DlgInteger(), DlgLabel(), DlgList(), DlgReal(),
DlgShow(), DlgString(), DlgText(), DlgValue(), DlgValue$(), DlgXValue()

# DlgString()

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

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

item%   This sets the item number in the dialog in the range 1 to the number of items.

text$   The prompt to display, optionally followed by a vertical bar and tooltip text.

wide    This is an alternative to the prompt. It sets the width in dialog units of the box in which the user types the string. If the width is not given the number entry box has a default width of `max%` or 60, whichever is the smaller.

max%    The maximum number of characters allowed in the string.

legal$  A list of acceptable characters. See `Input$()` for a full description. If this is omitted, or an empty string, all characters are allowed.

x       If omitted or zero, the string edit is right justified in the dialog box, otherwise positive values set the position of the left end of the string edit in dialog units and negative values set the position of the right hand end relative to the right edge of the dialog. When positioning the left edge using a positive value, a string item will be positioned approximately 1 dialog unit further to the left than most other controls - this is to take account of the edit field border.

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

sel$    If this string is present, it should hold a list of items separated by vertical bars, for example `"one|two| three"`. The field becomes an editable combo box with the items in the drop down list.

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.

There is an example of use here.

**See also:**
```
DlgChan(), DlgCheck(), DlgCreate(), DlgInteger(), DlgLabel(), DlgList(), DlgReal(),
DlgShow(), DlgValue(), DlgValue$(), DlgText(), DlgXValue(), Input$()
```

# DlgText()

This places non-editable text in the dialog box. This is different from `DlgLabel()` as you do not supply an item number and it does not require a variable in the `DlgShow()` function. It returns an item number (higher than item numbers for matching arguments in `DlgShow()`) that you can use to identify this field in call-back functions, for example `DlgVisible()`.

```
Func DlgText(text$, x, y{, wide});
```

text$   The prompt to display, optionally followed by a vertical bar and tooltip text.

x,y     The position of the bottom left hand corner of the first character in the string, in dialog units. Set `x` to 0 for the default label position (the same as `DlgLabel()`). Otherwise positive x values set the position of the left side of the text relative to the left-hand side of the dialog and and negative values set the right side position relative to the right-hand side of the dialog..

wide    Normally, the width of the field is set based on `text$`. This optional argument sets the width in dialog units. This allows you to replace the text with a longer string from a call-back function.

Returns  An item number to identify this field for call-back functions.

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

# DlgValue() and DlgValue$()

These functions are used from a dialog call-back function to get and optionally set the value of an item, spinner increment, item prompt or button text, they cannot be used in other circumstances.

```
Func DlgValue(item%{, val});
Func DlgValue$(item%{, val$});
```

item%      This identifies the dialog item. For items with arguments in `DlgShow()`, use the `item%` value you set to create the field. For items created with `DlgText()` and `DlgGroup()`, use the returned item number. For buttons use minus the button number to access the button label. To access the prompt for an item add 1000 to the item number. Add 2000 to the item number to access the step value of a spin control linked to a numeric field.

val|$      This optional argument holds the new item value. If you omit this argument the item is not changed. You can use numeric values on numeric fields or to set a check box or a list. You can use a text value to change a prompt or button label or to change the text of an editable control or to select the first matching item in a list box. It is up to you to make sure the text is acceptable for editable items.

Returns    The returned value is the current value of the item. You can use `DlgValue$()` on any item to get the current contents of the field, check box text prompt, button label or item prompt as a text string. Use `DlgValue()` to collect numeric or check box values.

If there is a problem running the command, for example if the item does not exist, or an argument type is not appropriate for an item, the result is an empty string or the value 0.

There is an example of use here.

**See also:**
`DlgCreate()`, `DlgAllow()`, `DlgEnable()`, `DlgShow()`, `DlgVisible()`

# DlgVisible()

This function is used from a dialog call-back function to show or hide dialog items, it cannot be used in other circumstances. There are two versions of this command. The version with a single argument returns the visible state of an item; the version with two or more arguments sets the visible state of one or more dialog items. When you show or hide an item, any prompt or spin control associated with the item is also shown or hidden. Using `DlgVisible()` to hide a button does not disable responding to associated keypresses, use `DlgEnable()` as well if you want this.

```
Func DlgVisible(show%, item%|const item%[]{, item%|const item%[]...});
Func DlgVisible(item%);
```

show%      Set this to 1 to show the items in the list and to 0 to hide them.

item%      An item number of an element of the dialog or an integer array containing a list of item numbers. The item numbers are either the number you set, or the number returned by `DlgText()` or `DlgGroup()`, or −button, where `button` is the button number. You cannot access prompts separately from their items as this makes no sense.

Returns    When called with a single argument it returns the visible state of the item, otherwise the return value is 0.

**See also:**
`DlgAllow()`, `DlgCreate()`, `DlgEnable()`, `DlgShow()`, `DlgValue()`, `DlgValue$()`

# DlgXValue()

This creates an editable combo box to collect an x axis value for the current file, memory or XY view. The combo box drop down list is populated with cursor positions and other window values when `DlgShow()` runs. If the current view is not suitable, the list is empty. This control accepts expressions, for example: `(Cursor(1) +Cursor(2))/2`. The matching `DlgShow()` argument is a real number to hold a time in seconds for a file view, or an x axis value for other views. This procedure is new in version 3.

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

item%      This sets the item number in the dialog in the range 1 to the number of items.

text$      The text to display as a prompt. If the prompt contains a vertical bar, "|" any following text will be used as a tooltip and displayed when the mouse pointer is held over the item.

wide       This is an alternative to the prompt. It sets the width in dialog units of the combo box. If the width is not given the combo box has a default width of 18 numbers.

x      If omitted or zero, the combo box is right justified in the dialog box, otherwise positive values set the position of the left end of the combo box in dialog units and negative values set the position of the right hand end relative to the right edge of the dialog. When positioning the left edge using a positive value, a time item will be positioned approximately 1 dialog unit further to the left than you might expect - this is to take account of the control border.

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

**See also:**
```
DlgAllow(), DlgButton(), DlgChan(), DlgCheck(), DlgCreate(), DlgInteger(),
DlgLabel(), DlgList(), DlgValue(), DlgValue$(), DlgReal(), DlgShow(), DlgString(),
DlgText()
```

# Draw()

This allows invalid regions in the current view to update and optionally sets the drawn X axis range. `Draw()` on a view that is up-to-date should make no change. The view is not brought to the front.

```
Proc Draw({from{, size}});
```

from    The left hand edge of the view in x axis units. For a text view, this sets the top line to display in the view (or as close to the top line as is possible) and scrolls the view horizontally so that the first character position is at the left of the view. For a grid view this sets the column at the top left of the view. The first column is 0. Use `YRange()` on a grid view to set the top row.

size    The width of the view in x axis units. A negative size is ignored. This argument must be omitted in a text view.

With no arguments, `Draw()` updates invalid areas in the view. With one argument, the view is scrolled to start at `from`. With two arguments, the width is set (unless it is unchanged) and then it is drawn. If `size` is negative or omitted, the same size as last time is used.

Data views run from `Mintime()` to `Maxtime()`. There are no limits set on the range of values for an XY view axis. However, huge numbers cause ugly axis labels.

**See also:**
```
DrawAll(), XRange(), YRange(), XLow(), XHigh(), Maxtime(), Mintime()
```

# DrawAll()

This routine updates all views that require redrawing. This is equivalent to iterating through all the views and performing a `Draw()` on each.

```
Proc DrawAll();
```

**See also:**
```
Draw()
```

# DrawMode()

This sets and reads the display mode for the channel in a data view. You can set the display mode for channels that are not displayed.

```
Func DrawMode(cSpc{, mode%{, dotSz%|binSz|offset{, err%|flags%{, point%{,
line%}}}}});
Func DrawMode(chan%{, item%{, value});
```

cSpc    A channel specifier for the channels to use.

mode%   If present and positive, this sets the display mode, and returns the previous mode. If an inappropriate mode is requested, no change is made. Some modes require additional parameters (for example a dot size). If additional parameters are omitted, the last known values are used. The mode values for setting draw modes in the view are:

0      The standard mode for the channel.

1      Dots for markers, using the dotSz% argument if it is provided.

| | |
|---|---|
| 2 | Lines for markers. |
| 3 | Rate for markers. |
| 4 | Histogram for waveform. |
| 5 | Line for waveform. |
| 6 | Dots for waveform, using the dotSz% argument if it is provided. |
| 7 | Skyline for waveform. |
| 8 | Cubic spline for waveform. |
| 9 | Waveform mode for real markers. |
| 10-14 | Reserved. |
| 15 | Basic for idealised trace. |
| 16 | Convolution for idealised trace. |
| 17 | Both basic and convolution of idealised trace. |

If `mode%` is present and negative, the function returns a current value without changing anything. The mode values for getting data are:

| | |
|---|---|
| -1 | Reserved, does nothing and returns -1. |
| -2 | Return the current dot size. |
| -3 | Return the current bin size for a histogram. |
| -4 | Return the basic trace offset for idealised trace drawing. |
| -5 | Return the current draw mode flags value. |
| -6 | Return the current point drawing mode for real markers in plot mode. |
| -7 | Return the current line drawing mode for real markers in plot mode. |
| -13 | Return the current drawing style for errors. |

Values in the range -8 to -12 are reserved for future use. If `mode%` is absent no changes are made.

`dotSz%` This sets the dot size to use on screen (or the point size to use on a printer) for modes 1 and 6 in units of the the pen width set for data in the Display page of the Preferences dialog. 0 is the smallest size available. The maximum size allowed is 10. Set to -1 for no change.

`binSz` This sets the width of the rate histogram bins for mode 3 only.

`offset` The y-offset of the basic idealised trace from the raw data for modes 15 and 17.

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

`flags%` This is the sum of values controlling various drawing options, the options available are:

1. To enable showing the baseline for idealised traces in Convolution or Both modes.
2. To turn on showing colours for marker codes for markers drawn as Dots or Lines and real markers in Waveform mode.
4. To turn off drawing of the centre line for marker and real marker channels in Lines mode.
8. To turn on drawing of marker codes as two hexadecimal digits only for marker and real marker channels in Dots mode.

`point%` selects the point drawing mode for real markers in plot mode, omit for no change. The point drawing modes available are:

| | | | | | |
|---|---|---|---|---|---|
| -1 | none) | 0 | dots (default) | 1 | boxes |
| 2 | plus signs + | 3 | crosses x | 4 | circles o |
| 5 | triangles | 5 | diamonds | 7 | horizontal line |
| 8 | vertical line | | | | |

`line%` selects the line drawing mode for real markers in plot mode, omit or set to -1 for no change. Styles are:

| | | | | | |
|---|---|---|---|---|---|
| -1 | none | 0 | solid (default) | 1 | dotted |
| 2 | dashed | | | | |

`chan%` A single channel number - required for the `item%` variant of this function.

`item%` Calls that use the `item%` form of the command (where `item%` is omitted or is negative) are used to read back specific information and to set values if the `value` argument is present. The values you can set with `item%` can also be set with the `mode%` command variant (the equivalent argument is in the `name` column). The allowed values of `item%` are:

| item% | name | Function |
|---|---|---|
| -1 | mode% | Or omit `item%` to return the current channel draw mode as described above. |
| -2 | dotSz% | Return or set the dot or tick size to use as described above. |
| -3 | binSz | Return or set the rate histogram bin width. |
| -4 | offset | Return or set the y-offset of the basic idealised trace from the raw data. |
| -5 | flags% | Return or set the draw mode flags as described above. |
| -6 | point% | Return or set the point drawing mode for real markers in plot mode as described above. |
| -7 | line% | Return or set the line drawing mode for real markers in plot mode as described above. |
| -13 | err% | Return or set the error drawing style as described above. |

value   Used with the `item%` variant to set new values.

Returns   The draw mode before the call if a single channel is set or a value determined by a negative `mode%` value. For multiple channels or an invalid call, it returns -1.

**See also:**
Channel draw mode dialog, `Draw()`, `ViewStandard()`, `XYDrawMode()`, `ChanValue()`

# Dup()

This gets the view handle of a duplicate of the current view, or the number of duplicates, including the original. Duplicated views are numbered from 1 (1 is the original). If a duplicate is deleted, higher numbered duplicates are renumbered. For more information on this see `WindowDuplicate()`.

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

num%   The number of the duplicate view to find, starting at 1. You can also pass 0 (or omit `num%`) as an argument, to return the number of duplicates.

Returns   If `num%` is greater than 0, this returns the view handle of the duplicate, or 0 if the duplicate does not exist. If `num%` is 0 or omitted, this returns the number of duplicates including the original. The following illustrates the use of `Dup()`.

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

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

# E

# EditClear()

In a memory view, this command zeros the data in all channels, in a text view it deletes any selected text. In a Grid view, this command clears selected cells.

```
Func EditClear();
```

Returns The function returns 0 if nothing was deleted, otherwise it returns the number of items deleted or 1 if the number is not known, or a negative error code.

**See also:**
`EditCopy(), EditCut()`

# EditCopy()

This command copies data from the current view to the clipboard or copies a string to the clipboard or creates a bitmap on the clipboard from an array. The effect depends on the type of the current view. Data and XY views copy as a bitmap, as a scaleable image, as text in the format set by `ExportTextFormat()`, `ExportChanFormat()` and `ExportChanList()`, or as binary data in a private format. Text views copy data as text. Grid views copy as text with cells in a row separated by the tab character and rows separated by end of line (`"\n"`).. The options to create a bitmap were added at Signal version 7.02.06. There are four command variants:

### Copy current view
The type of the current view determines what is available for copying. Time views copy as a bitmap, as a scalable image and as text in the format set by `ExportRectFormat()`, `ExportTextFormat()`, `ExportChanFormat()` and `ExportChanList()`. Result and XY views can copy as a bitmap, text and as a scalable image. Multimedia views copy as a bitmap. Text windows copy text. Grid views copy as text with cells in a row separated by the Tab character and rows separated by end of line (`"\n"`). Cursor range and values copy as text.

`Func EditCopy({as%});`

as% Sets how to copy data when several formats are possible. If omitted, all formats are used. It is the sum of: 1=Copy as a bitmap, 2=Copy as a scalable image (Windows metafile), 4=Copy as text, 8=Copy in CED private format.

Returns It returns 0 if nothing was placed on the clipboard, or the sum of the format specifiers for each format used.

### Copy script text
This variant copies script-defined text to the clipboard.

`Func EditCopy(text$);`

text$ A text string to place on the clipboard. This can be useful when you want to move results to a different program.

Returns 0 if nothing was copied, else 4 (text).

### Copy real array as a bitmap
This takes a real array holding red, green and blue intensities, plus an optional Alpha (opacity) channel to the clipboard as a bitmap. This was added at Signal version 7.02.

`Func EditCopy(arr[][][]{, flags%});`

arr An array of size `[3][wide%][high%]` to set Red, Green and Blue values, or an array of size `[4][wide%][high%]` to set Red, Green, Blue and opacity. All values are in the range 0 to 1.0; values outside this range are limited to the range. The matrix `arr[0][][]` holds the Red values, `arr[1][][]` holds Green values and `arr[2][][]` holds Blue values. If `arr[3][][]` exists, it holds the opacity values (0 for transparent to 1 for opaque). The colour in `arr[][0]0` correspond with the top left pixel in the bitmap unless `flags%` bit 0 is set, when it corresponds with the bottom left.

flags% A set of flags, taken as 0 if omitted. Currently only flag value 1 is defined which matches `arr[][0][0]` to the bottom left and the y direction runs upwards rather than downwards from the top.

Returns 0 if nothing was copied, else 1 (bitmap).

### Copy integer array as a bitmap
This copies an integer array that holds data already organised in bitmap format and copies it to the clipboard as a bitmap. Each integer value in the array holds colour values. This was added at Signal version 7.02.

`Func EditCopy(arr%[][]{, flags%});`

arr%   An integer array of size `[wide%][high%]` holding the bitmap image data. Each array element defines one pixel. Bits 0-7 hold the blue value, 8-15 the green and 16-23 the red (all as 0-255). You can also specify the use of an Alpha channel in bits 24-31. Bits above 31 are not used.

flags%  A set of flags, taken as 0 if omitted. 1 causes the bitmap to be created such that arr[][0][0] refers to the bottom left pixel. Without this flag bit set it refers to the top left pixel. Add 2 to use bits 24-31 of each array element as the Alpha (opacity) channel in the range 0 (transparent) to 255 (opaque).

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(),
EditClear(), EditCut(), EditPaste()
```

# EditCut()

This command cuts data from the current view to the clipboard (as for EditCopy()) and deletes the original (if this is possible).

**Func EditCut({as%});**

as%    This optional argument sets how data is cut and copied to the clipboard. Currently only text may be cut. If omitted, all allowed formats are used.

1    Cut and copy as a bitmap, has no effect.
2    Cut and copy as a scaleable image (metafile), has no effect.
4    Cut text to the clipboard.

Returns  0 if nothing was placed on the clipboard, or the sum of the format specifiers for each format that was used. The only possible values now are 0 and 4.

**See also:**
```
EditClear(), EditCopy(), EditPaste()
```

# EditFind()

In a text view, this command searches from the selection point for the next occurrence of the specified text and selects it if found. This is the same as the Find Text dialog.

**Func EditFind(find${, dir%{, flags%}});**

find$   The text to search for. May include regular expressions if 4 is added to `flags%`.

dir%    Optional. 0=search backwards, 1=search forwards (default), 2=wrap around.

flags%  Optional, taken as 0 if omitted. The sum of: 1=case sensitive, 2=find complete words only, 4=simple regular expression search (`dir%` may not be 0).

Returns  1 if found, 0 if not found.

**See also:**
```
Find text dialog, EditReplace()
```

# EditImageSave()

This script command, new at Signal version 7.02, saves a bitmap on the clipboard to a file in a user-defined format. Images may be placed on the clipboard using `EditCopy()` and `ArrMapImage()`.

**Func EditImageSave(file${, type%});**

file$   The file to save to. We expect you to provide a suitable file extension. If you do not set a file extension, the file is saved as the type set by the `type%` field, and failing that, it is saved as a bitmap.

type%   Optional (zero if omitted). If present, it forces the type of the output file:

Value   Forced type

| | |
|---|---|
| 0 | None. The file type is deduced from the file extension, else it is saved as a bitmap. |
| 1 | Standard Windows BMP file. |
| 2 | JPEG image. |
| 3 | PNG image. |
| 4 | TIFF image. |
| 5 | GIF image. |

Return   -1 if we failed, 0 for success.

**See also:**
```
ArrMapImage(),EditPaste(), EditCopy()
```

# EditPaste()

This command can be used to obtain the current clipboard text contents as a text string, or to get a bitmap as an array, or to paste the clipboard into a Text or Grid view. Some of the command features were added at Signal version 7.02.

| Target | Action |
|---|---|
| String | The text clipboard contents are copied to the string. |
| Text view | If the clipboard contains text the current text selection is replaced by the clipboard contents. |
| Grid view | If the clipboard contains text, the clipboard contents are pasted to the grid, starting at the current cell. The clipboard text can contain multiple cells and uses Tab characters to separate cells horizontally and end of line characters to separate cells vertically. Cells are pasted to the right and below the starting cell. After the paste operation, the rectangle holding the modified cells is selected. |
| Array | If the clipboard holds a bitmap, you can fill a 3d real array with the Red, Green, Blue components, and if the bitmap supports it, the opacity. |

There are several command variants:

**Paste text into Text or Grid view**
This is the simplest version of the command and is equivalent to typing `Ctrl+V` in the current view.

```
Func EditPaste();
```

Returns  The number of characters pasted into a text view or the number of cells pasted into a Grid view.

**Get the current clipboard text**
This copies any text on the clipboard into a script string.

```
Func EditPaste(&text$);
```

text$   Any text in the clipboard in returned in this string.

Returns  The return value is 1 if the clipboard holds text, 0 if it does not.

**Get clipboard information**
This variant is used to find out what information is on the clipboard and to get the size of any bitmap. This was added at Signal version 7.02.

```
Func EditPaste(what%{, &wide%, &high%});
```

what%   This determines the information you are seeking:

| | |
|---|---|
| 0 | Returns flags for the information on the clipboard in the same format as the `as%` argument of `EditCopy()`. The returned value is the sum of: 1=bitmap, 2=scalable image (Windows metafile), 4=text, 8=CED private format. |

1    Returns 1 if the clipboard holds a bitmap, 0 if it does not. If it holds a bitmap, `wide%` and `high%` are set to the width and height of the bitmap.

`wide%`    Only used when `what%` is 1. It is returned holding the width of the bitmap.

`high%`    Only used when `what%` is 1. It is returned holding the height of the bitmap.

Returns  The information as described for `what%`.

**Get clipboard bitmap image into an array**
These variants copy a bitmap from the clipboard into arrays, they were added at Signal version 7.02.

```
Func EditPaste(arr[][][]{, flags%});
Func EditPaste(arr%[][]{, flags%});
```

`arr`    An array of size `[3][wide%][high%]` to collect Red, Green and Blue values, or an array of size `[4]` `[wide%][high%]` to collect Red, Green, Blue and opacity. All returned values are in the range 0 to 1.0. The matrix `arr[0][][]` holds the Red values, `arr[1][][]` holds Green values and `arr[2][][]` holds Blue values. If the source bitmap has alpha (opacity) information and the array is large enough, `arr[3]` `[][]` holds the opacity values (0 for transparent to 1 for opaque). If the source holds no opacity information, all the returned values will be 0. Only array elements that map to pixels in the bitmap are changed. The colours in `arr[][0]0]` correspond with the top left pixel in the bitmap unless `flags%` bit 0 is set, when they correspond with the bottom left.

`flags%`  A set of flags, taken as 0 if omitted. Set 1 to cause incrementing y indices to run upwards from the bottom rather than downwards from the top. Set 2 with an integer array to collect opacity values.

Returns  1 if we successfully filled in the data, 0 if there was no suitable bitmap on the clipboard or -1 if the first array dimension size is not supported.

**See also:**
`EditCopy(), EditClear(), EditCut(), EditCopy(), EditImageSave(), ArrMapImage()`

# EditReplace()

In a text view, this command checks if the selection matches a pattern and replaces it if it does, then it searches for the pattern again. This is the same as the Replace Text dialog.

```
Func EditReplace(find${, repl${, dir%{, flags%}}});
```

`find$`    The text to search for.  May include regular expressions if 4 is added to `flags%`.

`repl$`    Optional replacement text, taken as an empty string if omitted.

`dir%`    Optional. 0=search backwards, 1=search forwards (default), 2=wrap around.

`flags%`  Optional, taken as 0 if omitted. The sum of: 1=case sensitive, 2=find complete words only, 4=regular expression search (`dir%` may not be 0).

Returns  The sum of: 1 if found a new match, 2 if replaced the original selection.

**See also:**
Replace text dialog, `EditFind()`

# EditSelectAll()

This function selects all items in the current view that can be copied to the clipboard. This is the same as the Edit menu Select All option.

```
Func EditSelectAll();
```

Returns  It returns the number of selected items that could be copied to the clipboard.

**See also:**
`EditCopy(), EditClear(), EditCut()(), MoveTo(), MoveBy(), Selection$()`

# Error$()

This function converts a negative error code returned by a function into a text string; it is able to provide a useful string describing all of the error codes generated by Signal.

```
Func Error$(code%);
```

`code%`   A negative error code returned from a Signal function.

Returns   It returns a string that describes the error.

# Eval()

This evaluates the argument and converts the result into text. The text is displayed in the Script view or the Evaluate window message area, as appropriate, when the script ends. The argument can be the value returned by a function.

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

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

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

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

Another use of `Eval()` is in the Script menu Evaluate window to see the result returned by a function or expression, as in these examples:

```
Eval(FileDelete(myfile$)); ' display 1 or a negative error code
Eval(Error$(-1531));       ' give string for error code if known
```

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

# Exp()

This function calculates the exponential function (e to the power of x, $e^x$) for a single value, or replaces the elements of a real array by their exponentials. If a value is too large, overflow will occur, causing the script to stop for single values, and a negative error code for arrays.

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

`x`         The argument for the exponential function or an array of real values.

Returns   With an array, the function returns 0 if all was well, or a negative error code if a problem was found (such as overflow). With an expression, it returns the exponential of the number.

**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 `FileExportAs()` and `EditCopy()`. It is equivalent to the data, time and headings settings for each channel type in the Text Output Configuration dialog, but note that though this is used to enable data output for a given type of channel, output for a specific channel will only be generated if the relevant type is enabled <u>and</u> the channel in question is selected for export using `ExportChanList()`. Using `ExportTextFormat()` with no arguments will reset these fields to enable the output of data, time and headings for the waveform channel type only. Output of error information (if available) can be enabled using this function but error output will only be generated if data output for the channel to which the errors apply is generated.

```
Proc ExportChanFormat(type%, data%, xval%, heads%);
```

`type%`  The type of data for which we are setting the format:

      0    Waveform

| | |
|---|---|
| 1 | Markers |
| 2 | Error values |
| 4 | Fitted curves |
| 5 | Idealised traces |
| 6 | Real markers |

`data%` Set this non-zero to enable data output for this channel type.

`xval%` Set this non-zero to enable output of x axis values for this channel type, this is ignored for marker and real marker data (where the time values are always output) errors (which is output in a column next to the associated waveform) and when `data%` is set to zero.

`heads%` Set this non-zero to enable output of column headings for this channel type. This is ignored for errors or if neither `data%` nor `xval%` are enabled.

**See also:**
```
EditCopy(), FileExportAs(), ExportChanList(), ExportFrameList(), ExportTextFormat(),
ExportTimeRange()
```

# ExportChanList()

This command sets a channel list to export for use by `FileExportAs()` and `EditCopy()` from a data view.

```
Proc ExportChanList(cSpc);
```

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

**See also:**
```
EditCopy(), FileExportAs(), ExportChanFormat(), ExportFrameList(),
ExportTextFormat(), ExportTimeRange()
```

# ExportFrameList()

This command sets a list of frames for use by `FileExportAs()` and `EditCopy()`.

```
Proc ExportFrameList(sFrm%{, eFrm%{, mode%}});
Proc ExportFrameList(frm$|frm%[]{, mode%});
```

`sFrm%` First frame to export. This option processes a range of frames. `sFrm%` can also be a negative code as follows:

| | |
|---|---|
| -1 | All frames in the file are included |
| -2 | The current frame |
| -3 | Frames must be tagged |
| -6 | Frames must be untagged |

`eFrm%` Last frame to export. If this is -1 the last frame is the last in the data view. This argument is ignored if `sFrm%` is a negative code.

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

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

`mode%` If `mode%` is present it is used to supply an additional criterion for including each frame in the range, list or specification. If `mode%` is absent all frames are included. The modes are:

| | |
|---|---|
| 0-n | Frames must have a state matching the value of `mode%` |
| -1 | All frames in the specification are processed |
| -2 | Only the current frame, if it is in the main list, is processed |
| -3 | Frames must also be tagged |
| -6 | Frames must also be untagged |

The following simple example exports all frames to `fred.cfs`.

```
ExportFrameList(-1);           'export from all frames in the view
FileExportAs ("fred.cfs", 1);  'export selected data as text
```

**See also:**
EditCopy(), FileExportAs(), ExportChanList(), ExportChanFormat(), ExportTextFormat(), ExportTimeRange()

# ExportTextFormat()

This command sets the text export format for use by `FileExportAs()` and `EditCopy()`. It is equivalent to setting decimal places, field width, string delimiter, item separator and frame header options in the Text Output Configuration dialog. The command with no arguments resets everything in the dialog to default settings: decimal places to 5, field width to 0, the string delimiter to double quotes, the separator to a tab character, the header and waveform channel units disabled. It also enables the output of data, time and headings for the waveform channel type only. See `ExportChanFormat()` to set these.

```
Proc ExportTextFormat({dDec%, tDec%, width%, lim$, sep${, flags%{, intp
%}}});
```

dDec%   Decimal places for data values.

tDec%   Decimal places for time values.

width%  Field width for all values, or zero for minimum width.

lim$    The delimiter, which is the character to place at the start and end of each text string in the output. The normal character to use is a double-quote mark.

sep$    The separator character which is used to separate multiple data items on a line. This should be one of tab, comma or space.

flags%  If this is present this sets various options, the default value is 0. Before version 6.03 this was the head% option, but as long as 0 was used for no header and 1 for header wanted, no incompatibilities will be encountered. It is the sum of:
   1   Output the frame header information
   2   Output waveform channel units in header (from version 6.03 onwards)

intp%   The interpolation method to use for waveform output. The default value is 0, the choices available are
   0   None
   1   Linear
   2   Cubic spline

**See also:**
EditCopy(), FileExportAs(), ExportChanList(), ExportFrameList(), ExportChanFormat(), ExportTimeRange()

# ExportTimeRange()

This command sets an x axis range for use by `FileExportAs()` and `EditCopy()` in a data view. This is equivalent to setting start and end times in the export setup dialogs.

```
Proc ExportTimeRange(sRange, eRange{, flags%});
```

sRange  The start of the range of data to export, in x axis units.

eRange  The end of the range of data to export, in x axis units.

flags%  This affects CFS data written by `FileExportAs()` and is the sum of:
   1   Time shift data so that it starts at zero in the output file.

**See also:**
EditCopy(), FileExportAs(), ExportChanFormat(), ExportChanList(), ExportFrameList(), ExportTextFormat()

# F

# File...()

## FileApplyResource()

This applies a resource file to the current data or XY view. If a data view is duplicated, all other duplicates are deleted, then the resource file is applied, which may create duplicates. The current view handle is not changed. Handles of duplicate views are not preserved, even if the resource file creates the same number of duplicates.

```
Func FileApplyResource(name$);
```

name$   The resource file to apply. You must include the `.sgrx` (or `.sgr` for older resource files) file extension and required path. Remember that \ in a string must be entered as \\ or use /. If the name is "" or contains a "*" or a "?", the user is prompted for a file.

Returns  The number of modified views (usually 1 unless the resource file generates duplicates), 0 if the resource file did not contain suitable information, -1 if the user cancelled the file dialog, -2 if the file could not be found or -3 if there is any other problem.

**See also:**
FileGlobalResource(), FileOpen(), FileSave(), FileSaveResource()

## FileClose()

This is used to close the current window or external file. You can supply an argument to close all views associated with the current data view or to close all the views belonging to the application.

```
Func FileClose({all%{, query%});
```

all%   This argument determines the scope of the file closing. Possible values are:

-1   Close all views except loaded scripts and debug windows

0   Close the current view. This is the same as omitting `all%`

1   Close all windows associated with the current view

query%  This determines what happens if a view holds unsaved data:

-1   Don't save the data or query the user

0   Query the user about each view that needs saving. If the user chooses Cancel, the operation stops, leaving all unclosed windows behind. This is the same as omitting `query%`.

Returns  The number of views that have not been closed. This can occur if a view needs saving and the user requests Cancel.

Note: A common fault in scripts is the use of the construct:
```
View(v%).FileClose(0);
```

This can cause problems because, if the current view is already `View(v%)`, then at the end of the function the script will attempt to switch back to `View(v%)` again, but it is now gone! This results in a "`View is wrong type`" error for no obvious reason. To avoid the problem use:

```
View(v%);
FileClose(0);
```

**See also:**
FileOpen(), FileSave(), FileSaveAs(), FileNew()

## FileComment$()

This function accesses the file comments in the file associated with the current file view or in a memory view. File comments for XY and text based views are always blank. The comment string is up to 72 characters in length.

```
Func FileComment$(num%{, new$});
Func FileComment$({new$});
```

num%     This selects the comment string number from 1 to 5, if num% is omitted then file comment string number 1 is used.

new$     If present, the command replaces the existing comment string with new$.

Returns  The comment string at the time of the call.

**See also:**
FrameComment$()

# FileConvert$()

This function converts a data file from a "foreign" format into a Signal data file. The range of foreign formats supported depends on the number of import filters in the Signal\import folder.

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

src$     This is the name of the file to convert. The file extension is used to determine the file type (unless flag% bit 0 is set). Known file extensions include: abf, cfs, cnt, cut, dat, eeg, ewb, ibw, son and uff. We expect to add more. If an empty string is used or one containing wild cards then a file selection dialog will appear.

dest$    If this is present, it sets the destination file. If this is not a full path name, the name is relative to the current directory. If you do not supply a file extension then Signal appends ".cfs". If you set any other file extension, Signal cannot open the file as a Signal data file. If you do not supply this argument, the converted file will be written to the same folder as the source file, using the original file name with the file extension changed to .cfs.

flag%    This argument is the sum of the flag values: 1=Ignore the file extension of the source file and try all possible file converters & if all else fails try the binary importer, 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.

cmd$     Optional string holding parameters that control the importer. The string is of the form: name1=value1;name2=value2;name3=value3 where the names are case insensitive. We append ;AppVer=Signal,700 (the 700 stands for the version of Signal multiplied by 100) to the end of the string. It is entirely up to the importers what parameter names they recognise. For more information about specific importers click here and then find the importer in the table. If the importer supports a command line, the cmd$ column contains Yes. Click the Yes for details of supported keywords.

Before the string is passed to an importer it is scanned for parameters that apply generally. At the moment, only one such option exists:

| Na me | Description |
| --- | --- |
| dll | You can use this option to select a specific import DLL. If you do not supply this, Signal will use the first importer it finds that supports a file extension that matches the file set by src$. Most data formats have a unique extension, but some have a wide range of extensions. For example, to force the use of the CED binary importer you would set cmd$ := "dll=binary" as the argument. You can find the DLL name from the table of importers for the File menu Import command, or record your actions. |

Returns  The full path name of the created file, or an empty string if the file was not converted.

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

# FileCopy()

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

```
Func FileCopy(src$, dest${, over%});
```

src$     The source file to copy to the destination. This file is not changed.

dest$     The destination file. If this file exists you must set `over%` to overwrite it.

over%     If this optional argument is 0 or omitted, the copy will not overwrite an existing destination file. Set to 1 to overwrite.

Returns     The routine returns 1 if the file was copied, 0 if it was not. Reasons for failure include: no source file, no destination path, insufficient disk space, destination exists and insufficient rights.

**See also:**
```
BRead(), BWrite(), FileDelete(), FileOpen(), ProgRun()
```

# FileDate$()

This function returns a string holding a date relating to the file associated with the current view. Use `FileTimeDate()` to get the date as numbers. Before Signal version 7.01 this function could only be used with a file view.

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

dayF%     This sets the format of the day field in the date, if omitted the value 2 is used. 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%     This sets the format of the month field, if omitted the value 3 is used. 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%     This sets the format of the year field, if omitted the value 1 is used. 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%     This sets thee order that the day, month and year appear in the string, if omitted the value 0 is used.

**0**     Operating system settings

1     month/day/year

2     day/month/year

3     year/month/day

sep$     This string appears between the day, month and year fields as a separator. If this string is empty or omitted, Signal supplies a separator based on system settings.

offs    This is an offset in seconds which can be added to the date retrieved from the file, most often to find the date corresponding to a time during sampling. If omitted it is treated as zero. This argument was added in Signal version 7.01.

which%   This it is a value from -1 to 2 which selects which date is read from the file. If omitted it is treated as zero except for a time view when the special value of -1 is used. This argument was added in Signal version 7.01.

- -1   For time views only, this selects the date on which sampling of the CFS file started.
- 0   This selects the date on which the file was last modified
- 1   This selects the date on which the file was created
- 2   This selects the date on which the file was last accessed

Returns   The date as a string.

**See also:**
`Date$(), FileTimeDate(), FileTime$()`

# FileDelete()

This function deletes one or more files. File names can be specified in full, or relative to the current directory.

Windows file names are of the form `x:\folder1\folder2\foldern\file.ext` or `\\machine\folder1\folder2\foldern\file.ext` across a network. If a name does not start with a \ or with `x:\` (where `x` is a drive letter), the path is relative to the current directory. Beware that \ must be written \\ in a string passed to the compiler.

**Func FileDelete(name$[]|name${, opt%});**

name$   This is either a string variable or an array of strings that holds the names of the files to delete. Only one name per string and no wildcard characters are allowed. If the names do not include a path they refer to files in the current directory.

opt%    If this is present and non-zero, the user is asked before each file in the list is deleted. You cannot delete protected or hidden or system files.

Returns   The number of files deleted or a negative error code.

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

# FileExportAs()

This function saves the current data view or the sampling configuration as a file on disk. A data view is saved either in its native format, or as text or as a picture. It is equivalent to the two File menu commands Export As and Save configuration. This cannot be used for external text or binary files as they are already on disk.

**Func FileExportAs(name${, type%{, mode%{, text${, flag%{, exp$}}}}});**

name$   The name to use for saving. If the string is empty or if the string holds wild card characters * or ?, then the File menu Save As dialog opens, otherwise it is used directly. In Windows, the wildcards select the initial list of files. If the string is used directly, a default file extension is not provided; you must provide the extension yourself.

type%   The type to save the file as (if omitted, type -1 is used):

- -2   For data and XY views only, opens a File Save As dialog which will allow the user to choose the type of file to save as.
- -1   Export in the native format for the data view. This is equivalent to using `type%` 0 for file and memory views or 12 for XY views.
- 0   Export part of the data view as set by `ExportFrameList()`, `ExportTimeRange()` and `ExportChanList()` to a new Signal data file. The file extension should be `.cfs`.
- 1   Save the contents of the current data or XY view as a text file. Signal saves the data as set by `ExportFrameList()`, `ExportTimeRange()` and `ExportChanList()` in the text format set by

ExportChanFormat() and ExportTextFormat(). If the export frame list is empty then the current frame is used. The file extension should be .txt.

5 Save data or XY view as a picture file. The file extension should be .wmf.

6 Save the sampling configuration in a configuration file. The function will only save sampling configuration data using the new-style .sgcx file format and will force the file extension used to be .sgcx. Old-style sampling configuration files can only be converted to new-style XML by using them for sampling. The function fails and returns an error if it is used to try to save an old-style sampling configuration that has not been converted to new-style.

12 For XY views only, save as an XY data file. The file extension should be .sxy.

13 Save the view as a bitmap picture.

14 As 13, but saving in Joint Photographic Expert Group (JPEG) format.

15 As 13, but saving in Portable Network Graphics (PNG) format.

16 As 13, but saving in Tagged Image File Format (TIFF).

17 Save as Grid view (Grid views only).

100 Types 100 and upwards identify external exporters installed in the export folder (where you will find additional documentation). Time and result views export the channels and time range set by ExportChanList(). XY views export the data set by flag%. Codes defined so far:

| Code | Exporter | |
|---|---|---|
| 100 | MatLab | follow this link to see the string format used for the exp$ argument (below). |

mode% This optional argument has a default value of zero and is the sum of:

1 Do not query the user if this operation would overwrite an existing file. While an existing file is open in Signal you will not be able to overwrite it.

8 Allow the operating system to select the initial directory for any Save As dialog that may be provided, otherwise use the current directory.

text$ An optional prompt displayed as part of the file dialog to prompt the user.

flag% Used when saving XY views as text or to an external exporter to override the normal behaviour, which is to save all visible channels and all data points. This optional argument has a default value of 1 and is the sum of:

1 Output only visible channels, otherwise all XY channels.

2 Output only data that is in the visible range (use 3 for visible data).

exp$ External exporter settings as a string (see the external exporter documentation for argument names) of the form: "name1=value|name2=value|name3=value".

Returns The function returns 0 if the operation was a success or a negative error code.

**See also:**
```
EditCopy(), ExportChanFormat(), ExportChanList(), ExportFrameList(),
ExportTimeRange(), ExportTextFormat()
```

# FileGetIntVar()

This function reads a CFS file variable of integer type from the file attached to the current view, which must be a file view. The CFS supports the use of file and frame variables of integer, floating point and string types. Software other than Signal may have included these when creating a data file.

```
Func FileGetIntVar(name${, &nVar%{, &units${, &nType%}}});
```

name$ The name of the variable to look for. This string is not case sensitive but every character including spaces must match exactly.

nVar% If present this returns the variable number, -1 if not found, or a negative error code.

units$ If present this returns the units for the variable.

nType% If present this returns a code for the CFS type of an integer variable:

0: INT1, 1: WRD1, 2: INT2, 3: WRD2, 4: INT4:

Returns  The function returns the value of the variable if the operation was a success, otherwise zero.

**See also:**
```
FileGetRealVar(), FileGetStrVar$(), FileVarCount(), FileVarInfo(), FrameGetIntVar(),
FrameGetRealVar(), FrameGetStrVar$(), FrameVarCount(), FrameVarInfo()
```

# FileGetRealVar()

This function reads a CFS file variable of real type from the file attached to the current view, which must be a file view.

```
Func FileGetRealVar(name${, &nVar%{, &units$}});
```

name$  The name of the variable to look for. This string is not case sensitive but every character including spaces must match exactly.

nVar%  If present this returns the variable number, -1 if not found, or a negative error code.

units$ If present this returns the units for the variable.

Returns  The function returns the value of the variable if the operation was a success, otherwise zero.

**See also:**
```
FileGetIntVar(), FileGetStrVar$(), FileVarCount(), FileVarInfo(), FrameGetIntVar(),
FrameGetRealVar(), FrameGetStrVar$(), FrameVarCount(), FrameVarInfo()
```

# FileGetStrVar$()

This function reads a CFS file variable of string type from the file attached to the current view, which must be a file view.

```
Func FileGetStrVar$(name${, &nVar%{, &units$}});
```

name$  The name of the variable to look for. This string is not case sensitive but every character including spaces must match exactly.

nVar%  If present this returns the variable number, -1 if not found, or a negative error code.

units$ If present this returns the units for the variable.

Returns  The function returns a string contents of the variable if the operation was a success, otherwise an empty string.

**See also:**
```
FileGetIntVar(), FileGetRealVar(), FileVarCount(), FileVarInfo(), FrameGetIntVar(),
FrameGetRealVar(), FrameGetStrVar$(), FrameVarCount(), FrameVarInfo()
```

# FileGlobalResource()

This function is equivalent to the Global Resources dialog. Please see the documentation for the dialog for a full explanation. The function has two forms: the first is for setting values, the second is to read them back:

```
Func FileGlobalResource(flags%, loc%, name${, path$});
Func FileGlobalResource(&loc%, &name$, &path$);
```

flags% This is the sum of the following values:

1    Enable the use of global resources. If unset, no global resources are used.
2    Only use global resources if a data file has no associated resource file.
4    Only use if the data file is within the path set by path$.

loc%   The location to search for the global resource file. 0=the system folder list, 1=search the data file folder, then the system list, 2=the data file folder only.

name$  The name of the global resource file excluding the path and .sgrx (or .sgr for older resource files) file extension.

path$ The file path within which to use the global resources. If you omit this argument when setting values, the current path does not change.

Returns When setting values, the return value is 0 or a negative error code. When reading back values, the return value is the `flags%` argument.

**See also:**
```
Global Resources dialog, FileApplyResource(), FileOpen(), FileSave(),
FileSaveResource()
```

# FileList()

This function gets lists of files and sub-directories (folders) in the current directory and can also return the path to the parent directory of the current directory. This function can be used to process all files of a particular type in a particular directory.

```
Func FileList(names$[]|&name$, type%{, mask$});
```

name$ This is either a string variable or an array of strings that is returned holding the name(s) of files or directories. Only one name is returned per string.

type% This sets the type of the objects to return information on. Allowed values are:

-3 The parent directory of the current directory. The full path is returned.

-2 Sub-directories of the current directory. No path is returned.

-1 All files in the current directory.

0 Signal data files (`*.cfs`).

1 Text files (`*.txt`).

2 Output sequence files (`*.pls`).

3 Signal script files (`*.sgs`).

6 Signal sampling configuration files (`*.sgcx` and `*.sgc`). The `mask$` parameter can be used to select files with just one file extension should this be required.

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 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. The negative `mode%` values were added in Signal versions 7.01 and 6.06.

```
Func FileName$({mode%});
```

mode% If present, determines what to return, if omitted this is taken as 0.

0 Return the full file name including the path

1 The disk drive/volume name

2 The path section, excluding the volume/drive and the name of the file

3 The file name up to and not including the last dot in the name, also excluding any trailing number

4 Any trailing numbers that were excluded from 3

5 The end of the file name from the last dot - the file name extension

-1 The disk drive/volume plus the path.

-2 The disk drive/volume, path and the file name excluding any trailing number

-3 The disk drive/volume, path and the file name including any trailing number

-4    The file name including any trailing number

-5    The complete file name including any trailing number and the file name extension

Returns  A string holding the requested name, or a blank string if there is no file.

**See also:**
```
FileList(), FilePath$(), FilePathSet(), FileDelete()
```

# FileNew()

This is equivalent to the File menu New command; it creates a new window, also called a view, and returns the view 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%});
Func FileNew(17, mode%{, cols%{, rows%}});
```

type%    The type of file to create:

0    A Signal data file based on the sampling configuration, ready for sampling. This opens a new file view which is also referred to as the sampling document view . It may also open other windows which will include the sampling control panels.

1    A text file in a window.

2    An output sequence file in a window.

3    A Signal script file in a window.

12   An XY view with one (empty) data channel. Use `XYAddData()` to add more data and `XYSetChan()` to create new channels.

17   An empty Grid view with `rows%` rows and `cols%` columns.

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

1    Make the new window(s) visible immediately. If this flag is not set the window is created, but is invisible.

2    For data files, if the sampling configuration holds information for creating additional windows, use it. If this flag is not set, data files extract enough information from the sampling configuration to set the sampling parameters for the data channels.

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

**See also:**
```
FileOpen(), FileSave(), FileSaveAs(), FileClose(), SetMemory(), SampleStart(),
XYAddDAta(), XYSetChan()
```

# FileOpen()

This is the equivalent of the File Open menu command; it opens an existing Signal data file or a text file in a window, or an external text or binary file. If the file is already opened, a handle for the existing view is returned. The window becomes the new current view. You can create windows as visible or invisible. It is often more convenient to create an invisible window so you can position it before making it visible.

```
Func FileOpen(name$, type%{, mode%{, text$}});
```

name$    The name of the file to open. This can include a path. The file name is operating system dependent, see `FileDelete()`. If the name is blank or holds wild card characters, the file dialog opens for the user to select a file. Don't forget that if you want to get a single backslash (\) character in the file name you have to use a double backslash in the string constant.

type%    The type of the file to open. The types currently defined (see `ViewKind()`) are:

0    Open a Signal data file. A new **file view** is created.

1    Open a text file. A new **text view** is created.

2    Open an output sequence file. A new **output sequence view** is created.

3   Open a Signal script file. A new **script view** is created.

6   Load a sampling configuration file. No new view is created. The function will try both the .sgc extension for old-style resources and the new .sgcx extension for XML resources if necessary. If the a `.sgc` file extension is given and the specified file does not exist the new-style `.sgcx` extension is tried. Similarly the `.sgc` extension is tried if the specified file with a `.sgcx` extension was not found. This means that old scripts will work without any changes and can be switched to using the new-style files (if both files exist) by deleting the old file. Scripts can be changed to use new-style files by default by changing the file extension specified to `.sgcx`.

8   An external text file without a window. An invisible **external text view** is created in which `Read()` or `Print()` can be used.

9   An external binary file without a window. An invisible **external binary view** is created in which `BRead()`, `BWrite()`, `BSeek()` and other binary routines can be used.

12   Open an XY data file. A new **XY view** is created.

17   Open a Grid view file. A new **Grid view** is created.

`mode%`   This optional argument determines how the window or file opens and how the associated resource file information is used. If the argument is omitted, its value is 0.

For file types 0 to 3 and 12 (plus type 6 for the 8 value only) `mode%` 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 (except to memory channels, see below). Note that if your data file contains virtual channels, opening the file without reading the resources will remove them from the file.

4   Return an error if the file is already open in Signal. If this flag is not set and the file is already in use, it is brought to the front and its handle is returned.

8   Any file open dialog that is generated starts off showing the directory selected by the OS (this is usually the place where the last file was opened from or saved to), if not set the dialog starts off showing the current directory. This also applies to sampling configuration files (type 6), unlike all of the other `mode%` values.

16   Do not load memory channel data from the resources. Memory channel data is loaded from the resources by default except if this flag is set and the value 2 flag for overall resource file information use is not set.

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.

Add 8 to this value for any file open dialog that is generated to start off showing the directory selected by the OS (this is usually the place where the last file was opened from or saved to), if not the dialog starts showing the current directory.

Add 32 to this value (for external text files) to read and pay attention to any BOM or failing that, read where appropriate as Unicode (wide UTF-16LE) characters. For writing text, use wide (UTF-16LE) characters.

`text$`   An optional prompt displayed as part of the file dialog, for all except type 6. If this is supplied then the file dialog will appear even if a complete file name is also supplied.

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

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

**See also:**
```
FileDelete(), FileNew(), FileSave(), FileSaveAs(), FileClose(), BRead(), BReadSize(),
BSeek(), BWrite(), BWriteSize(), ViewFind(), ViewList(), ViewKind()
```

# FilePath$()

This function gets the "current directory", the place on disk where file open and file save dialogs start from. It can also get the path for created data files, various special directories available for user purposes or the directory where the Signal application is installed.

```
Func FilePath$({opt%});
```

opt%    If omitted this is taken as zero. This determines which directory/folder to get:

    0    The current directory. This is the directory that incomplete path names (names that do not start with a drive specification or a backslash) are relative to.

    1    The path for sampled data files created by `FileNew()` as set by the Sampling page in the Preferences dialog.

    2    The path to the directory where the Signal application is installed. Using this location for user data is **<u>strongly deprecated</u>** as Signal may be installed within the protected `Program Files` directory tree and generally attempts to save files here or write to existing files will fail, so please **<u>do not</u>** use this opt% value, use -4 instead if possible. At some point in the future CED may enforce this by refusing to return this path and returning an empty string or a different path instead.

    3    The automatic file saving path as set by the Automation section of the Sampling Configuration dialog.

    As an experiment from version 5.00, we also support some negative values that return the path to some special folders:

    -1    The (user-specific) Desktop folder, for example: C:\Users\username\Desktop\

    -2    The users documents folder, for example: C:\Users\username\Documents\

    -3    The system folder for Signal user and application data, for example: C:\Users\username\AppData\Local\CED\Signal7\. You can use this location to save files that are specific to the current logged on user and Signal. You could also consider using the registry with the `Profile()` command for small quantities of information.

    -4    The `Signal7` folder inside the users documents folder (commonly called `My Documents`). It is recommended that you use this location for your data and files if possible. Available from version 5.08 only.

    -5    The `Signal7Shared` folder inside the documents folder for all users. Available from version 5.08 only.

    -6    The folder where Signal can store private information, normally C:\ProgramData\CED\Signal7\. It is not recommended that you make use of this directory as it may be protected.

Returns  A string holding the path or an empty string if an error is detected.

**See also:**
```
FilePathSet(), FileList(), FileName$(), SampleAutoName$()
```

# FilePathSet()

This function sets the current directory/folder, and where Signal data files created by `FileNew()` are stored until they are sent to their final resting place by `FileSaveAs()`. There are two version of the command. The first sets or optionally creates a directory based on a passed in path, the second prompts the user to choose and optionally create a directory.

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

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 or a prompt is set, a dialog opens for the user to select an existing directory/folder. Don't forget that if you want to get a single backslash (\) character in a path you have to use a double backslash in the string constant.

opt% If omitted this is taken as zero. This determines which directory/folder to set:

0 The current directory. This is the directory that incomplete path names (names that do not start with a drive specification or a backslash) are relative to.

1 The path for Signal data files created by `FileNew()` as set by the Sampling page of the Preferences dialog.

2 The path to the application. This is fixed, so an attempt to change it is an error.

3 The automatic file naming path as set by the Automation section of the Sampling Configuration dialog.

make% In the first command version, if this is zero or omitted, all elements of the path must already exist. If set to 1 and path$ is not empty, the command will create the directory/folder if all elements of the path exist except the last. In the second version of the command that displays a dialog, if this is zero the user can only select an existing path. If set to 1, the user is allowed to create a new directory/folder. If the users sets a path you can find out what was set with `FilePath$()`.

prmpt$ Optional prompt for use with the dialog. If you supply a prompt, a user dialog will appear using the current value of path$ as the starting point. If path$ is empty, the current path set by opt% is the starting point.

Returns  Zero if the path was set, or a negative error code.

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

# FilePrint()

This function is equivalent to the File menu Print command; it prints some or all of the current view to the printer that is currently set. If no printer has been set, the current system printer is used. In a file or memory view, it prints a range of data with the x axis scaling set by the display. In a text or log view, it prints a range of text lines. There is currently no script mechanism to choose a printer; you must do it interactively.

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

from The start point of the print. This is in seconds in a file or memory view and in lines in a text view. If omitted, this is taken as the start of the view.

to The end point in the same units as from. If omitted or set beyond the end of the view then the end of the view is used.

flags% 0=portrait, 1=landscape, 2=current setting. If omitted, the current value is used.

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

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

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

# FilePrintScreen()

This function is equivalent to the File menu Print Screen command; it prints all visible data, XY and text-based views to the current printer on one page. The page positions are proportional to the view positions in the Signal application window.

```
Func FilePrintScreen({head${, vTtl%{, box%{, scTxt%{, flags%{, foot$}}}}}});
```

head$ The page header. If omitted or an empty string, there is no page header.

vTtl%   Set 1 or higher to print a title above each view, omitted or 0 for no title.

box%   Set 1 or higher for a box around each view. If omitted, or 0, no box is drawn.

scTxt% Set 1 or higher to scale text differently in the x and y directions to match the original. If omitted or 0 scale both directions by the same scale factor.

flags% 0=portrait, 1=landscape, 2=current setting. If omitted, the current value is used.

foot$   The page footer. If omitted or an empty string, there is no page footer.

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

**See also:**
FilePrint(), FilePrintVisible()

# FilePrintVisible()

This function prints the current view as it appears on the computer screen to the current printer. In a text view, this prints the lines in the current selection. If there is no selection, it prints the line containing the cursor. This function is equivalent to the File menu Print visible command.

```
Func FilePrintVisible({flags%});
```

flags% 0=portrait, 1=landscape, 2=current setting. If omitted, the current value is used.

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

**See also:**
FilePrint(), FilePrintScreen()

# FileQuit()

This is equivalent to the File menu Exit command. If there is any unsaved data you are asked if you wish to save it before the application closes. If the user cancels the operation (because there were files that needed saving), the script terminates, but the Signal application is left running. Use FileClose(-1, -1) before FileQuit() to guarantee to exit.

```
Proc FileQuit();
```

**See also:**
FileClose()

# FileSave()

This function saves the current view as a file on disk. It is equivalent to the File menu Save command. You cannot use this command for a file view if it has just been sampled, use FileSaveAs() instead. If the view has not been saved previously, the File menu Save As dialog opens and the user must provide a file name. This cannot be used for external text or binary files either as they are already on disk.

```
Func FileSave();
```

Returns  The function returns 0 if the operation was a success, or a negative error code.

**See also:**
FileOpen(), EditCopy(), FileExportAs(), FileSaveAs(), FileClose()

# FileSaveAs()

This function is equivalent to the File menu SaveAs command, it is used to save the current view, with any relevant changes, under a new name. You can also use this function to save and name a Signal data file immediately after it has been sampled. Use FileExportAs() to export selected parts of a CFS data file to a new file, to export from a view under a different format or to save the current sampling configuration in a file.

```
Func FileSaveAs(name${, mode%{, text$}});
```

name$    The name to use for saving. If the string is empty or if the string holds wild card characters * or ?, then the File menu Save As dialog opens; the wildcards select the initial list of files.

mode%    This optional argument has a default value of zero and is the sum of:

    1    Do not query the user if this operation would overwrite an existing file. While an existing file is open in Signal you will not be able to overwrite it.

    8    Allow the operating system to select the initial directory for any Save As dialog that may be provided, otherwise use the current directory.

text$    An optional prompt displayed as part of the file dialog to prompt the user.

Returns    The function returns 0 if the operation was a success, or a negative error code.

**See also:**
```
FileSave(), EditCopy(), FileExportAs()
```

# FileSaveResource()

This saves a resource file for the current data or XY view. If a data view is duplicated, resources for all the duplicates are saved.

```
Func FileSaveResource({name$|glob%})
```

name$    The resource file to save to. If the name is "" or contains a "*" or a "?", the user is prompted for a file. Any extension in the file name is ignored and the extension is set to .sgrx. Remember that \ in a string must be entered as \\ or use /. If the name is "" or contains a "*" or a "?", the user is prompted for a file.

glob%    An alternative to name$. 0=prompt for a file name, 1=save resources to the resource file associated with the view (this may be a global resource file), 2=save to the global resource file only (if enabled). If both name$ and glob% are omitted, the effect is the same as setting glob% to 1.

Returns    0 if all was OK, -1 if the user cancelled the File Save dialog, -2 if the file could not be saved for any other reason.

This command will add information to an existing resource file, or create a new one.

**See also:**
```
FileGlobalResource(), FileApplyResource(), FileOpen(), FileSave()
```

# FileSize()

This function returns the size of the data file associated with the current data view. During sampling this allows for data that is buffered, but not yet written. If you use this during sampling, you will see the file size increase each time a new frame is written, that is, it will not increase by the size of data items but by the size of data frames. This function did not exist before Signal version 4.07.

```
Func FileSize();
```

Returns    The size of the file, in bytes, as a real number as it can exceed integer range.

**See also:**
File size limit

# FileTime$()

This function returns a string holding a time relating to the file associated with the current view. Use FileTimeDate() to get the time as numbers. Before Signal version 7.01 this function could only be used with a file view.

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

tBase%   Specifies the time base to show the time in. You can choose between 24 hour or 12 hour clock mode. If this argument is omitted, 0 is used.

**0** Operating system settings    2  12 hour format

1  24 hour format

show%   Specifies the time fields to show. Add the values of the required options together and use that as the argument. If this argument is omitted or a value of 0 is used, 7 (1+2+4) is used for 24 hour format and 15 (1+2+4+8) for 12 hour format.

1    Show hours        4    Show seconds

2    Show minutes     8    Remove leading zeros from hours

amPm%   This sets the position of the "AM" or "PM" string in 12 hour format and has no effect in 24 hour format. If omitted, a value of zero is used. The string that gets printed ("AM" or "PM") is specified by the operating system.

**0**    Operating system settings    2  Show to the left of the time

1    Show to the right of the time  3  Hide the "AM" or "PM" string

sep$   This string appears between adjacent time fields. If sep$ = ":" then the time will appear as 12:04:45. If an empty string is entered or sep$ is omitted, the operating system settings are used.

offs   This is an offset in seconds which can be added to the time retrieved from the file, most often to find the absolute time corresponding to a time during sampling. If omitted it is treated as zero. This argument was added in Signal version 7.01.

which%   This it is a value from -1 to 2 which selects which time is read from the file. If omitted it is treated as zero except for a file view when the special value of -1 is used. This argument was added in Signal version 7.01.

-1   For file views only, this selects the time at which sampling of the CFS file started.

0   This selects the time at which the file was last modified

1   This selects the time at which the file was created

2   This selects the time at which the file was last accessed

Returns  The time as a string, or an empty string if we failed to get a useful time.

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

# FileTimeDate()

This function returns a time and date relating to the file associated with the current view as numbers. Use FileTime$() and FileDate$() to get the result as strings. Before Signal version 7.01 this function could only be used with a file view.

```
Func FileTimeDate(&s%{, &m%{, &h%{, &d%{, &mon%{, &y%{, &wDay%{, which
%}}}}}}});
Func FileTimeDate(td%[]{, which%})
```

s%   If s% is the only argument, it is set to the number of seconds since midnight. Otherwise it is set to the number of seconds since the start of the current minute.

m%   If this is the last argument, it is set to the number of minutes since midnight. Otherwise it is set to the number of full minutes since the start of the hour.

h%   If present, the number of hours since Midnight is returned in this variable.

d%   If present, the day of the month is returned as an integer in the range 1 to 31.

mon%   If present, the month number is returned as an integer in the range 1 to 12.

y%   If present, the year number is returned here. It will be an integer such as 2002.

wDay%   If present, the day of the week will be returned here as 0=Monday to 6=Sunday.

which% This it is a value from -1 to 2 which selects which time and date is read from the file. If omitted it is treated as zero except for a file view when the special value of -1 is used. This argument was added in Signal version 7.01.

-1 For file views only, this selects the time and date at which sampling of the CFS file started.

0 This selects the time and date at which the file was last modified

1 This selects the time and date at which the file was created

2 This selects the time and date at which the file was last accessed

td%[] If an array is the first argument, the first seven elements are filled with time and date data. The array can be less than seven elements long. Element 0 is set to the seconds, 1 to the minutes, 2 to the hours, and so on.

Returns 0 if we failed to get a time and date, 1 for success. This function was a procedure with no value returned prior to version 7.01.

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

# FileTimeDateSet()

This command sets the sampling start time for the file associated with the current time view, this time has been stored in all Signal data files since version 1.00. If you create a file using the script language and do not use this command, the sampling start time is set to the time of creation of the file.

```
Func FileTimeDateSet(s%, m%, h%, d%, mon%, y%);
Func FileTimeDateSet(td%[])
```

s% Sets the seconds in the range 0-59.

m% Sets the minutes in the range 0-59.

h% Sets the hours in the range 0-23.

d% Sets the day of the month in the range 1 to 31.

mon% Sets the month of the year in the range 1 to 12.

y% Sets the year in the range 1970 to 2069. This limited range will be removed in a later version of Signal.

td%[] If an array is the first and only argument, the first six elements provide the new time and date for the file. Element 0 sets the seconds, 1 sets the minutes, 2 sets the hours, 3 sets the days, 4 sets the month and 5 sets the years. The values should be in the same range as for the s% to y% arguments.

Returns If all the values are in the correct ranges and the file is not read only, the new data is set and the return value is 0. If there is a problem, the return value is a negative error code.

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

# FileVarCount()

This function counts CFS file variables in the data file. File variables are extra values attached to a CFS data file that are used by Signal for various purposes. A Signal script can read the values of these variables but is not allowed to change them.

```
Func FileVarCount();
```

Returns The number of file variables in the data file associated with this view.

**See also:**
FileGetIntVar(), FileGetRealVar(), FileGetStrVar$(), FileVarInfo(), FrameGetIntVar(), FrameGetRealVar(), FrameGetStrVar$(), FrameVarCount(), FrameVarInfo()

# FileVarInfo()

This function reads the name, type and optionally the units of a CFS file variable. File variables are extra values attached to a CFS data file that are used by Signal for various purposes. A Signal script can read the values of these variables but is not allowed to change them.

```
Func FileVarInfo(nVar%, &name${, &units$});
```

nVar%    This is the variable number.

name$    This is returned holding the name of the variable, which can be used in the commands for reading the file variables.

units$   If provided, this is returned holding the units for the variable.

Returns  The function returns the type of the variable or -1 if the variable was not found or is of unknown type. The type code is as follows:

    0   An integer variable which can be read using `FileGetIntVar()`

    1   A floating point variable which can be read using `FileGetRealVar()`

    2   A string variable which can be read using `FileGetStrVar$()`

**See also:**
```
FileGetIntVar(), FileGetRealVar(), FileGetStrVar$(), FileVarCount(),
FrameGetIntVar(), FrameGetRealVar(), FrameGetStrVar$(), FrameVarCount(),
FrameVarInfo()
```

# FiltApply()

This function applies a set of filter coefficients or a filter in the filter bank to a set of waveform channels in the current file or memory view.

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

```
Func FiltApply(n%|coef[], cSpc, frm%|frm%[]|frm$);
```

n%       Index of the filter in the filter bank to apply in the range -1 to 11, or

coef[]   An array holding a set of FIR filter coefficients to apply to the waveform.

cSpc     A channel specifier for the channels to filter.

frm%     Frame number or a negative code as follows:

    -1   All frames in the file

    -2   The current frame

    -3   Only tagged frames

    -6   Only untagged frames

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

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

Returns  The number of the last channel to be filtered or a negative error code. A negative error code is also returned if the user clicks Cancel from the progress bar that may appear during a long filtering operation.

**See also:**
```
Interactive FIR filtering, Filter banks, Filter types, Technical details of FIR
filters, FiltAtten(), FiltCalc(), FiltComment$(), FiltCreate(), FiltInfo(),
FiltName$(), FiltRange()
```

# FiltAtten()

This set the desired attenuation for a filter in the filter bank. When `FiltApply()` or `FiltCalc()` is used, the number of coefficients needed to achieve this attenuation (up to a maximum of 255) will be generated. A value of zero sets the attenuation back to the default (-65 dB).

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

`index%`  Index of the filter in the filter bank to use in the range -1 to 11.

`dB`      If present and negative, this is the desired attenuation for stop bands in the filter.

Returns  The desired attenuation for a filter at the time of the call.

**See also:**
```
Interactive FIR filtering, Filter banks, Filter types, Technical details of FIR
filters, FiltApply(), FiltCalc(), FiltComment$(), FiltCreate(), FiltInfo(),
FiltName$(), FiltRange()
```

# FiltCalc()

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

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

`index%`  Index of the filter in the filter bank to use in the range -1 to 11.

`sInt%`   The sample interval of the waveform you are about to filter. This is the value returned by `BinSize()` for a waveform channel.

`coeff`   An array to be filled with the coefficients used for filtering. If the array is too small, as many elements as will fit are set. The maximum size needed is 2047.

`dBGot`   If present, returns the attenuation attained by the filter coefficients.

`nCoef%`  If present, sets the number of coefficients used in the calculation (use an even number for a full differentiator and an odd number for all other filter types).

Returns  The number of coefficients generated by the filter.

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

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

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

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

- The transition gap and the width of a pass or stop band cannot be smaller than 0.005 of the sampling frequency.
- The transition gap cannot be larger than 0.12 of the sampling frequency.

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

1. If the sampling frequency is such that, to produce the filter, the transition gap and/or pass and stop band widths are outside their limits, then the widths are set to the limits before calculating the filter. In our 50 Hz low pass

filter example, if we calculate it with respect to a 12 kHz sampling frequency, the minimum pass band width is 12000*0.005 = 60 Hz. So, the filter would be changed to a 60 Hz low pass filter.

2.  If half the sampling frequency (the Nyquist frequency) is less than an edge of a pass or stop band, certain attributes of the filter are lost. In our 50 Hz low pass filter example, if we tried to calculate with a sampling frequency of 80 Hz, we would see that the Nyquist frequency is 40 Hz. No frequency above 40 Hz can be represented in a waveform sampled at 80 Hz, so a 50 Hz low pass filter is equivalent to an "All pass" filter. The filter specification will be altered to reflect this before calculating.

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

**See also:**
Interactive FIR filtering, Filter banks, Filter types, Technical details of FIR filters, 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 -1 to 11.

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

Returns  The previous comment for the filter at the index.

**See also:**
Interactive FIR filtering, Filter banks, Filter types, Technical details of FIR filters, 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 -1 to 11. This action replaces any existing filter at this index. Note that, for this function and all the other FiltXXXX function, an index of -1 refers to special filter information that is not part of the saved filter banks.

`type%`   The type of the filter desired (see table).

`trW`     The transition width of the filter. This is the frequency interval between the edge of a stop band and the edge of the adjacent pass band.

`edgeN`   This is a list of edges of pass bands in Hz. (see table).

Returns  0 if there was no problem or a negative error code if the filter was not created.

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

| type% | Name | trW | edge1 | edge2 | edge3 | edge4 |
|---|---|---|---|---|---|---|
| 0 | All stop | | | | | |
| 1 | All pass | | | | | |
| 2 | Low pass | Yes | High | | | |
| 3 | High pass | Yes | Low | | | |
| 4 | Band pass | Yes | Low | High | | |
| 5 | Band stop | Yes | High(1) | Low(2) | | |
| 6 | Low pass differentiator | Yes | High | | | |

| 7 | Differentiator | | | | | |
|---|---|---|---|---|---|---|
| 8 | 1.5 Band Low pass | Yes | High(1) | Low(2) | High(2) | |
| 9 | 1.5 Band High pass | Yes | Low(1) | High(1) | Low(2) | |
| 10 | 2 Band pass | Yes | Low(1) | High(1) | Low(2) | High(2) |
| 11 | 2 Band stop | Yes | High(1) | Low(2) | High(2) | Low(3) |

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

**See also:**
```
Interactive FIR filtering, Filter banks, Filter types, Technical details of FIR
filters, FiltApply(), FiltAtten(), FiltCalc(), FiltComment$(), FiltInfo(),
FiltName$(), FiltRange()
```

# FiltInfo()

This function retrieves information about a filter in the bank.

**Func FiltInfo(index%{, what%});**

`index%` Index of the filter in the filter bank to use in the range -1 to 11.

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

-2 Maximum `what%` number allowed

-1 Desired attenuation

0 type (if you supply no value, 0 is assumed)

1 Transition width

2-5 `edge1-edge4` given in `FiltCreate()`

Returns The information requested as a real.

**See also:**
```
Interactive FIR filtering, Filter banks, Filter types, Technical details of FIR
filters, FiltApply(), FiltAtten(), FiltCalc(), FiltComment$(), FiltCreate(),
FiltName$(), FiltRange()
```

# FiltName$()

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

**Func FiltName$(index%{, new$});**

`index%` Index of the filter in the filter bank to use in the range -1 to 11.

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

Returns The previous name of the filter at that index.

**See also:**
```
Interactive FIR filtering, Filter banks, Filter types, Technical details of FIR
filters, FiltApply(), FiltAtten(), FiltCalc(), FiltComment$(), FiltInfo(),
FiltCreate(), FiltRange()
```

# FiltRange()

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

**Proc FiltRange(index%, &minFr, &maxFr);**

`index%` Index of the filter in the filter bank to use in the range -1 to 11.

`minFr` Returns the minimum sampling frequency you can calculate the filter with respect to, so that no transition width is greater than the maximum allowed and no attributes of the filter are lost.

maxFr    Returns the maximum sampling frequency you can calculate the filter with respect to, without the transition (or band) widths being smaller than allowed.

It is possible to create a filter which cannot be applied to any sampling frequency without being changed. This will be apparent because `minFr` will be larger than `maxFr`.

**See also:**
```
Interactive FIR filtering, Filter banks, Filter types, Technical details of FIR
filters, FiltApply(), FiltAtten(), FiltCalc(), FiltComment$(), FiltInfo(),
FiltCreate(), FiltName$()
```

# FIRMake()

This function creates FIR filter coefficients and places them in an array ready for use by `ArrFilt()`. This command is very similar in operation to the DOS program `FIRMake` and has similar input requirements. Unless you need precise control over all aspects of filter generation, you may find it easier to use `FiltCalc()` or `FIRQuick()`. You will need to read the detailed information about FIR filters in the description of the Digital Filter dialog to get the best results from this command.

```
Proc FIRMake(type%, param[][], coef[]{, nGrid{, extFr[]}});
```

type%    The type of filter file to produce: 1=Multiband filter, 2=Differentiator, 3=Hilbert transformer, 4=Multiband pink noise (Multiband with 3 dB per octave roll-off).

param    This is a 2-dimensional array. The size of the first dimension must be 4 or 5. The size of the second dimension (n) should be the number of bands in your filter. You pass in 4 values for each band (indices 0 to 3) to describe your filter:

Indices 0 and 1 are the start and end frequency of each band. All frequencies are given as fraction of a sampling frequency and so are in the range 0 to 0.5.

Index 2 is the function of the band. For all filter types except a differentiator, this is the gain of the filter in the band in the range 0 to 1 (the most common values are 0 for a stop band and 1 for a pass band.) For a differentiator, this is the slope of the filter in the band, normally not more than 2. The gain at any frequency *f* in the band is given by *f\*function*.

Index 3 is the relative weight to give the band. The weight sets the relative importance of the band in multiband filters. The program divides each band into frequency points and optimises the filter such that the maximum ripple times the weight in each band is the same for all bands. The weight is independent of frequency, except in the case of the differentiator, where the weight used is weight/frequency.

If there is an index 4 (the size of the first dimension was 5), this index is filled in by the function with the ripple in the band in dB.

coef    An array into which the FIR filter coefficients are placed. The size of this array determines the number of filter coefficients which are calculated. It is important, therefore, to make sure this array is exactly the size that you need. The maximum number of coefficients is 512.

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:**
More about FIRMake() filter types, Interactive FIR filtering, Filter banks, Filter types, Technical details of FIR filters, 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 512. This is the maximum number of coefficients that can be created and this function reserves the right to return as many as it feels necessary, up to that value to create a decent filter.

type%  This sets the type of filter to create. 0=Low pass, 1=High pass, 2=Band pass, 3=Band stop and 4=Differentiator.

freq   This is a fraction of the sampling rate in the range 0 to 0.5 and means different things depending on the type of filter.:

For Low pass, High pass and Differentiator types, this represents the cut-off frequency. This is the frequency of the higher edge of the first frequency band.

For Band pass and Band stop filters, this is the midpoint of the middle frequency band: the pass band in a Band pass filter, the stop band in a Band stop filter.

width  For Low pass, High pass and Differentiator filters, this is the width of the transition gap between the stop band and the pass band. The default value is 0.02 and there is an upper limit of 0.1 on this argument.

For a band Pass or Band stop filter, `width` is the width of the middle band. E.g. if you ask for a Pass band filter with the `freq` parameter to be 0.25 and the width to be 0.05, the middle pass band will be from 0.2 to 0.3. For these types of filter, you still need a positive transition width. This transition width is 0.02 and cannot be changed by the user.

atten  The desired attenuation in the stop band in dB. The default is 50 dB. This is analogous to the desired attenuation in the `FiltAtten()` command.

Returns The number of coefficients calculated. If the array is not large enough the coefficient list is truncated (and the result is useless).

**See also:**
More about FIRMake() filter types, Interactive FIR filtering, Filter banks, Filter types, Technical details of FIR filters, ArrFilt(), FiltApply(), FiltCalc(), FIRMake(), FIRResponse()

# FIRResponse()

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

**Proc FIRResponse(resp[], const coef[]{, as%{, type%}});**

resp   The array to hold the frequency response. This array will be filled regardless of its size. The first element is the amplitude response at 0 Hz and the last is the amplitude response at the Nyquist frequency. The remaining elements are set to the response at a frequency proportional to the element's position in the array.

coef   The coefficient array calculated by `FIRMake()`, `FIRQuick()` or `FiltCalc()`.

as%     If this is 0 or omitted, the response is in dB (0 dB is unchanged amplitude), otherwise as linear amplitude (1.0 is unchanged).

type%   If present, informs the command of the filter type. The types are the same as those supplied for `FIRQuick()`: 0=Low pass, 1=High pass, 2=Band pass, 3=Band stop and 4=Differentiator. If a type is given , the time to calculate the response is halved. If you are not sure what type of filter you have, or you have type not covered by the `FIRQuick()` types, then do not supply a type to this command.

**See also:**
```
Interactive FIR filtering, Filter banks, Filter types, Technical details of FIR
filters, ArrFilt(), FiltCalc(), FIRMake(), FIRQuick()
```

# FitCoef()

This command gives you access to the fit coefficients for the next `FitData()` fit. You can return the values from any type of fit and set the initial values and limits and hold values fixed for iterative fits. There are two command variants:

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

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

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

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

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

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

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

### Get and set the hold flags
This command variant allows you to hold some coefficients at their current values during the next fit.

```
Func FitCoef(hold%[]);
```

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

Returns  This always returns 0.

**See also:**
```
FitData(), FitValue(), FitExp(), ChanFitCoef()
```

# FitData()

This function, together with `FitCoef()` and `FitValue()`, lets you apply the same fitting functions that are available for channel data to data in arrays. You supply arrays of x and y data points and an optional array holding the standard deviation of the input data point y values. There are three command variants:

### Initialise fit information
The first variant sets the type of fit. If you select an iterative fit, the initial values of the fitting coefficients are reset to standard values and any "hold" flags set by `FitCoef()` are cleared. You can set your own initial values with the `FitCoef()` command or make a guess at the initial values when performing the fit.

```
Func FitData(type%, order%);
```

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

order%   If positive, this is the order of the fit, if negative it is minus the number of fitted coefficients. See the information about each fit for the allowed values for each fit type. If type% is 0 this argument is ignored and should be 0.

Returns  The number of fit coefficients for the fit or a negative error code.

### Exponential fit

This fits multiple exponentials by an iterative method. The data is fitted to the equation:

$y = \mathbf{a}_0 \exp(-x/\mathbf{a}_1) + \mathbf{a}_2 \exp(-x/\mathbf{a}_3)\ldots$   For an even number of coefficients
$y = \mathbf{a}_0 \exp(-x/\mathbf{a}_1) + \mathbf{a}_2 \exp(-x/\mathbf{a}_3)\ldots + \mathbf{a}_n$   For an odd number of coefficients

You can set up to 10 coefficients or orders 1 to 5. If you use a fit order, the number of coefficients is the order times 2. See the FitExp() command for more information. Coefficient estimates are effective for orders 1 and 2.

### Polynomial fit

This fits $y = \mathbf{a}_0 + \mathbf{a}_1 x + \mathbf{a}_2 x^2 + \mathbf{a}_3 x^3 \ldots$ to a set of $(x,y)$ data points. The fitting is by a direct method; there is no iteration. The fit order is the highest power of x to fit in the range 1 to 10. The number of coefficients is the fit order plus 1.

### Gaussian fit

This fits multiple Gaussians by an iterative method. The data is fitted to the equation:

$y = \mathbf{a}_0 \exp(-\frac{1}{2}(x-\mathbf{a}_1)^2/\mathbf{a}_2^2) + \mathbf{a}_3 \exp(-\frac{1}{2}(x-\mathbf{a}_4)^2/\mathbf{a}_5^2) + \ldots$

The fitted parameters (coefficients) are the $\mathbf{a}_i$. You can fit up to 3 Gaussians (order 1 to 3). The number of coefficients is given by the fit order times 3. Coefficient estimates become less useful as the order increases.

### Sine fit

This fits multiple sinusoids by an iterative method. The data is fitted to the equation:

$y = \mathbf{a}_0 \sin(\mathbf{a}_1 x + \mathbf{a}_2) + \mathbf{a}_3 \sin(\mathbf{a}_4 x + \mathbf{a}_5) + \ldots \{+ \mathbf{a}_{3n}\}$

The fitted coefficients are the $\mathbf{a}_i$. Angles are evaluated in radians. You can fit up to 3 sinusoids (order 3) and an optional offset. Although the function is given in terms of sine functions, you can easily convert to cosines by subtracting $\pi/2$ from the phase angle ($\mathbf{a}_2$, $\mathbf{a}_5$, $\mathbf{a}_8$) after the fit. The coefficient count can be set to 3, 4, 6, 7, 9 or 10. If you use orders, the number of coefficients is order times 3 and you cannot set an offset. A useful coefficient estimate is made for a single sinusoid fit.

### Sigmoid fit

This fits a single Boltzmann sigmoid by an iterative method. The data is fitted to the equation:

$y = \mathbf{a}_0 + (\mathbf{a}_1 - \mathbf{a}_0)/(1 + \exp((\mathbf{a}_2 - x)/ \mathbf{a}_3)$

In terms of the fitted result, $\mathbf{a}_0$ and $\mathbf{a}_1$ are the low and high fitting limits, $\mathbf{a}_2$ is the X50 point and $\mathbf{a}_3$ is related to the reciprocal of the slope at the X50 point. The slope at the X50 point, where x is $\mathbf{a}_2$, is $(\mathbf{a}_1 - \mathbf{a}_0)/(4 \mathbf{a}_3)$. You can set order 1 only, or 4 coefficients.

### Perform the fit

This variant of the command does the fit set by the previous variant. Use the FitCoef() command to preset fit coefficients and to read back the result of the fit.

```
Func FitData(opt%, y[], x[]{, s[]|s{, &err{, maxI% {,&iTer%{, covar[]
[]}}}}}});
```

opt%    1=Estimate the coefficients before fitting, 0=use current values. Note that the estimates are usually only useful for a small number of coefficients.

y[]     An array of y values to be fitted.

x[]     A corresponding array of x values.

s[]|s     A corresponding array of standard deviations for the data points defined by y[] and x, or a single value, being the standard deviation of each point. If this value is omitted or set to 1.0, the result is a least squares fit. If standard deviations are supplied, the result is a chi-squared fit.

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

maxI%     If present, this changes the maximum number of iterations from 100.

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

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

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

## Get fit information

This variant of the command returns information about the current fit.

```
Func FitData({opt%});
```

opt%     This determines what to return. opt% values match ChanFit(), where possible. The returned information for each value of opt% is:

| opt% | Returns | opt% | Returns |
|---|---|---|---|
| 0 | Fit type of next fit | 1 | Fit order of next fit |
| -1 | 1=a fit exists, 0=no fit exists | -8 | Not used |
| -2 | Type of last fit or 0 | -9 | Not used |
| -3 | Number of coefficients | -10 | Not used |
| -4 | Chi or least-squares error | -11 | 1=chi-square, 0=least-square |
| -5 | Fit probability (estimated) | -12 | Last fit result code |
| -6 | Lowest x value fitted | -13 | Number of fitted points |
| -7 | Highest x value fitted | -14 | Number of fit iterations used |

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

**See also:**
ChanFit(), FitCoef(), FitValue()

# FitExp()

This command fits multiple exponentials to arrays of x,y data points, with an optional weight for each point. Fitting is by an iterative method. The data is fitted to the equation:

$y = \mathbf{a}_0 \exp(-x/\mathbf{a}_1) + \mathbf{a}_2 \exp(-x/\mathbf{a}_3)\dots$       For an even number of coefficients

$y = \mathbf{a}_0 \exp(-x/\mathbf{a}_1) + \mathbf{a}_2 \exp(-x/\mathbf{a}_3)\dots + \mathbf{a}_n$       For an odd number of coefficients

The fitted coefficients are the $\mathbf{a}_i$; odd numbered $\mathbf{a}_i$ are assumed to be positive. You can fit up to 5 exponentials or 4 exponentials and an offset. However, experience shows that trying to fit more than two exponentials requires care. The fit from even two exponentials should be viewed with caution, especially if the odd coefficients are similar. The commands to implement this are:

## Set up the problem

The first command sets the number of exponentials to fit and the data set to be fitted. You must call this function before you call any of the others.

```
func FitExp(nCoef%, const y[], const x[]{, const s[]|s});
```

nCoef%   The number of coefficients to fit in the range 2 to 10. If this is even, the first form of the function above is used. If it is odd, the final coefficient is an offset.

y[]   An array of y data values. The length of the array must be at least `nCoef%`.

x[]   An array of x data values. The length of the array must be at least `nCoef%`.

s   An optional argument which is either an array with one value for each y data point or a single value for all data points. If the value is the standard deviation, then the error value returned when you iterate to find the best fit is the chi-squared value and the fit is a chi-squared fit.

   If this value is proportional to the error in the y values, then the fit is still a chi-squared fit, and the error returned is proportional to the chi-squared value. If you omit this array, the fit is a least-squares error fit, and the error value returned is the sum of the squares of the errors in the y values.

Returns   The function returns 0. There is no other return value as all errors stop the script.

The number of data points is set by the smallest of the sizes of the `y[]`, `x[]` or `s[]` (if present) arrays. The number of data points must be at least the number of coefficients.

## Set coefficient values and ranges

This variant of the function sets the initial value of each coefficient and optionally sets the range of allowed values. You can call this function at any time after the setup call and before the iterate call (below) has returned 0 indicating that the fit is completed.

It is important that you give reasonable initial values for the coefficients, especially when you fit more than one exponent. You should limit the odd coefficient values (the time constants) so that they cannot be zero and make sure that multiple exponents do not have overlapping ranges. If two exponents get similar values, the fit is degenerate and will wander around forever without getting anywhere. However, setting too rigid a range may damage the fitting process as sometimes the minimisation process has to follow a convoluted n-dimensional path to reach the goal, and the path may need to wander quite a bit. Let experience be your guide.

If you do not give starting values, the command will make a simplistic guess at the fitting values. As we expect that you know more about the "right" answer than the command does, we suggest that you set the values you want.

```
Func FitExp(coef%, val{, lo, hi});
```

coef%   The coefficient to set. The first coefficient is number 0, the last is `nCoef%`-1.

val   The initial value to assign to the coefficient. If you have set low and high limits, and the value is outside these limits, it is set to the nearer limit.

lo,hi   If present, these two values set the acceptable range of values for this coefficient. If omitted, or if both values are set to the same value, there is no limit. The value of the coefficient is tested against these limits after each iterative step, and if it exceeds a limit, it is set to that limit.

## Iterate to a solution

Once you have set up the problem and given initial values to your coefficients, you can start the iteration process that will move the coefficients from their starting values to new values that minimise the error (optionally scaled by the `s` argument).

```
func FitExp(a[], &err{, maxI%{, &iTer%{, covar[][]}}});
```

a[]   An array of size at least `nCoef%` that is returned holding the current set of coefficient values. The first amplitude is in `a[0]`, the first exponent in `a[1]`, the second amplitude in `a[2]`, the second exponent in `a[3]` and so on.

err   A real variable returned as the sum over the data points of $(yx[i]-y[i])^2/s[i]^2$ if `s[]` is used or holding the sum of $(yx[i]-y[i])^2$ if `s[]` is not used, where yx[i] is the value predicted from the coefficients at the x value `x[i]`.

maxI%   This is the maximum number of fitting iterations to do before returning from the function. If you omit this value, the function sets 1000. You can set any value from 1 to 10000. If you set more than 10000, the number is limited to 10000.

iTer%   An optional integer variable that is returned holding the number of iterations done before the function returned.

covar   An optional two dimensional array of size at least `[nCoef%][nCoef%]` that is returned holding the covariance matrix when the fit is complete. It is not changed unless the function return value is 0 or -1.

Returns   1 if the number of iterations are exhausted and the fit has not converged, 0 if the fit has converged, -1 if the fit is not improving but the result may be OK. Other negative numbers indicate failure.

### Select coefficients to fit

Normally the command fits all the coefficients, but you can use this variant to select the coefficients to fit. You can use this command at any time after you have set the problem until the iteration variant returns 0 or -1.

```
func FitExp(const fit%[]);
```

fit%[]   An array of at least nCoef% integers. If `fit%[i]` is 0, coefficient i is held constant, otherwise it is fitted. If all elements are 0, all arguments are fitted.

The effect of this command persists until either the iteration variant returns a value less than 1, or you set up a new problem, or you call this variant again.

### An example

This is a template for using these commands to fit all the coefficients:

```
const nData%:=50;         'set number of data elements
var x[nData%], y[nData%]; 'space for our arrays
var coefs[4],err;         'coefficients and squared error
...                       'in here goes code to get the data
FitExp(4, y[], x[]);      'fit two exponentials (no sigma array)
FitExp(0, 1.0, 0.2, 4);   'set first amplitude and limit range
FitExp(1, .01, .001, .03);'set first time constant and range
FitExp(2, 2.0, 0.1, 6);   'set second amplitude and limit range
FitExp(3, .08, .03, .15); 'set second time constant and range
repeat
   DrawMyData(coefs[], x[], y[]);  'Some function to show progress
until FitExp(coefs[], err, 1) < 1;
DrawMyData(coefs[], x[], y[]);     'Show the final state
```

### See also:

More about curve fitting, `ChanFit()`, `FitGauss()`, `FitLinear()`, `FitNLUser()`, `FitPoly()`, `FitSigmoid()`, `FitSin()`

# FitGauss()

This command fits multiple Gaussians to x,y data points with an optional weight for each point. Fitting is by an iterative method. The input data is fitted to the equation:

$$y = \mathbf{a}_0 \exp(-\tfrac{1}{2}(x-\mathbf{a}_1)^2/\mathbf{a}_2^2) + \mathbf{a}_3 \exp(-\tfrac{1}{2}(x-\mathbf{a}_4)^2/\mathbf{a}_5^2) + \ldots$$

The fitted parameters (coefficients) are the $\mathbf{a}_i$. You can fit up to 3 Gaussians. The commands to implement this are:

### Set up the problem

The first command sets the number of Gaussians to fit and the data set to be fitted. You must call this function before you call any of the others.

```
func FitGauss(nCoef%, const y[], const x[]{, const s[]|s});
```

nCoef%   The number of coefficients to fit. The legal values are 3, 6 and 9 for one, two and three Gaussians.

y[]   An array of y data values. The length of the array must be at least nCoef%.

x[]   An array of x data values. The length of the array must be at least nCoef%.

s   An optional argument which is either an array with one value for each y data point or a single value for all data points. If the value is the standard deviation, then the error value returned when you iterate to find the best fit is the chi-squared value and the fit is a chi-squared fit.

   If this value is proportional to the error in the y values, then the fit is still a chi-squared fit, and the error returned is proportional to the chi-squared value. If you omit this array, the fit is a least-squares error fit, and the error value returned is the sum of the squares of the errors in the y values.

Returns  The function returns 0. There is no other return value as all errors stop the script.

The number of data points is set by the smallest of the sizes of the `y[]`, `x[]` or `s[]` (if present) arrays. The number of data points must be at least the number of coefficients. If it is not, you will get a fatal error, so check this before calling the function.

### Set coefficient values and ranges

This variant of the function sets the initial value of each coefficient and optionally sets the range of allowed values. You can call this function at any time after the setup call and before the iterate call (below) has returned 0 indicating that the fit is completed.

In this type of fitting it is very important that you give reasonable starting values for the coefficients. In particular, when fitting multiple Gaussians, it is usual that the centre of each distribution is easy to determine. If you can set the centres and limit them so that they cannot overlap, the fit usually will proceed without any problems, even for multiple Gaussians. If you do not give starting values, the command will make a simplistic guess at the fitting values. As we expect that you know more about the "right" answer than the command does, we suggest that you set the values you want.

```
func FitGauss(coef%, val{, lo, hi});
```

`coef%`  The coefficient to set. The first coefficient is number 0, the last is `nCoef%`-1.

`val`  The initial value to assign to the coefficient. If you have set low and high limits, and the value is outside these limits, it is set to the nearer limit.

`lo,hi`  If present, these two values set the acceptable range of values for this coefficient. If omitted, or if both values are set to the same value, there is no limit. The value of the coefficient is tested against these limits after each iterative step, and if it exceeds a limit, it is set to that limit.

As long as you make a reasonable estimate of the centre points, there should be no problems fitting multiple Gaussians.

### Iterate to a solution

Once you have set up the problem and given initial values to your coefficients, you can start the iteration process that will move the coefficients from their starting values to new values that minimise the error (optionally scaled by the `s` argument).

```
func FitGauss(a[], &err{, maxI%{, &iTer%{, covar[][]}}});
```

`a[]`  An array of size at least `nCoef%` that is returned holding the current set of coefficient values. The first amplitude is in `a[0]`, the first centre in `a[1]`, the first sigma in `a[2]`, the second amplitude in `a[3]` and so on.

`err`  A real variable returned as the sum over the data points of (yx[i]-`y[i]`) $^2$/`s[i]` $^2$ if `s[]` is used or holding the sum of (yx[i]-`y[i]`) $^2$ if `s[]` is not used, where yx[i] is the value predicted from the coefficients at the x value `x[i]`.

`maxI%`  This is the maximum number of fitting iterations to do before returning from the function. If you omit this value, the function sets 1000. You can set any value from 1 to 10000. If you set more than 10000, the number is limited to 10000.

`iTer%`  An optional integer variable that is returned holding the number of iterations done before the function returned.

`covar`  An optional two dimensional array of size at least `[nCoef%][nCoef%]` that is returned holding the covariance matrix when the fit is complete. It is not changed unless the function return value is 0 or -1.

Returns  1 if the number of iterations are exhausted and the fit has not converged, 0 if the fit has converged, -1 if the fit is not improving but the result may be OK. Other negative numbers indicate failure.

Remember that even when a minimum is found, there is no guarantee that this is the minimum. It is the best minimum that this algorithm can find given the starting point.

### Select coefficients to fit

Normally the command fits all the coefficients, but you can use this variant to select the coefficients to fit. You can use this command at any time after you have set the problem until the iteration variant returns 0 or -1.

```
func FitGauss(const fit%[]);
```

fit%[] An array of at least nCoef% integers. If fit%[i] is 0, coefficient i is held constant, otherwise it is fitted. If all elements are 0, all arguments are fitted.

The effect of this command persists until either the iteration variant returns a value less than 1, or you set up a new problem, or you call this variant again.

### An example
This is a template for using these commands to fit all the coefficients:

```
const nData%:=50;          'set number of data elements
var x[nData%], y[nData%];   'space for our arrays
var s[nData%];             'space for sigma of each point
var coefs[4], err;         'coefficients and error squared
...                        'in here goes code to get the data
FitGauss(3, y[], x[], s[]); 'fit one gaussian
FitGauss(0, 1.0, 0.2, 4);   'set amplitude and limit range
FitGauss(1, 2, 1.5, 2.5);   'set centre of the gaussian and range
FitGauss(2, 0.5, 0.3, 1.9); 'set width and limit range
repeat
   DrawMyData(coefs[], x[], y[]);  'Some function to show progress
until FitGauss(coefs[], err, 1) < 1;
DrawMyData(coefs[], x[], y[]);     'Show the final state
```

### See also:
More about curve fitting, ChanFit(), FitExp(), FitLinear(), FitNLUser(), FitPoly(), FitSigmoid(), FitSin()

# FitLine()

This function calculates the best fit line to a set of data points using the least squares error method. This can be applied to any Waveform, Real marker or XY view channel. It fits the expression:

$y = \mathbf{m}\, x + \mathbf{c}$

through the data points $(x_i, y_i)$ so as to minimise the error given by:

$\text{Sum}_i (y_i - \mathbf{m}\, x_i - \mathbf{c})^2$

In this expression, $\mathbf{m}$ is the gradient of the line and $\mathbf{c}$ is the y axis intercept when x is 0.

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

chan%   A channel number (1 to n) holding waveform, real marker or XY view data.

start   The start position for processing. Start and finish are given in x axis units.

finish  The end position for processing. A data value at the finish position is included in the calculation.

grad    This is returned holding the gradient of the best fit line (**m**).

inter   This is returned holding the intercept of the line with the y axis (**c**).

corr    This is returned holding correlation coefficient indicating the "goodness of fit" of the line. Values close to 1 or -1 indicate a good fit; values close to 0 indicate a very poor fit. This parameter is often referred to as *r* in textbooks.

Returns  0 if all was OK, or -1 if there were not at least 2 data points.

The results are in user units, so in a view with a waveform measured in volts on an x axis of seconds, the units of the gradient would be volts per second and the units of the intercept would be volts.

### See also:
ChanFit(), ChanMeasure(), FitLinear(), FitPoly()

# FitLinear()

This command fits $y = \mathbf{a}_0 f_0(x) + \mathbf{a}_1 f_1(x) + \mathbf{a}_2 f_2(x)\ldots$ to a set of $(x,y)$ data points. If you can provide error estimates for each $y$ value, you can use the covariance output from this command to provide confidence limits on the calculated coefficients and you can use the returned chi-square value to test if the model is likely to fit the data. The command is:

```
func FitLinear(coef[], const y[], const x[][]{, const s[]|s{, covar[][]{,
r[]{, mR}}}});
```

`coef[]`　A real array which sets the number of coefficients to fit and which return the best fit set of coefficients. The array must be between 2 and 10 elements long. The coefficient a0 is returned in `coef[0]`, a1 in `coef[1]` and so on.

`y[]`　A real array of y values.

`x[][]`　This array specifies the values of the functions f(x) at each data point. If there are `nc` coefficients and `nd` data values, this array must be of size at least `[nd][nc]`. Viewed as a rectangular grid with the coefficients running from left to right and the data running from top to bottom, the values you must fill in are:

$$
\begin{array}{llllll}
f_0(x_0) & f_1(x_0) & f_2(x_0) & f_3(x_0) & \ldots & f_{nc-1}(x_0) \\
f_0(x_1) & f_1(x_1) & f_2(x_1) & f_3(x_1) & \ldots & f_{nc-1}(x_1) \\
f_0(x_2) & f_1(x_2) & f_2(x_2) & f_3(x_2) & \ldots & f_{nc-1}(x_2) \\
\ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\
f_0(x_{nd-1}) & f_1(x_{nd-1}) & f_2(x_{nd-1}) & f_3(x_{nd-1}) & \ldots & f_{nc-1}(x_{nd-1})
\end{array}
$$

`s`　This is either a real array holding the standard deviations of the `y[]` data points, or a real value holding the standard deviation of all data points. If `s` is omitted or zero, a least-squares error fit is performed, otherwise a chi-squared fit is done.

`covar`　An optional two dimensional array of size at least `[nc][nc]` (nc is the number of coefficients fitted) that is returned holding the covariance matrix.

`r[]`　An optional array of size at least `[nc]` (nc is the number of coefficients fitted) that is returned holding diagnostic information about the fit. The less relevant a fitting function $f_i(x)$ is to the fit, the smaller the value returned. The element of the array that corresponds to the most relevant function is returned as 1.0, smaller numbers indicate less relevance.

If your fitting functions are not independent of each other, several coefficients may have low `r` values. The solution is to remove one of the functions from the fit, or to set the `mR` argument to exclude one of the functions, then fit again. If the remaining coefficients become relevant, you have excluded a function that was a linear combination of the others. If the remaining coefficients still are not relevant, you have eliminated a function that did not contribute to the fit.

`mR`　You can use this optional variable to set the minimum relevance for a function. Functions that have less relevance than this are "edited" out of the fit and their coefficient is returned as 0. If you do not provide this value, the minimum is set to 10 -15, which will probably not exclude any values.

Returns　The function returns the chi-square value for the fit if `s[]` or `s` is given (and non-zero), or the sum of squares of the errors between the data points and the best fit line if `s` is omitted or is zero.

The smallest of the sizes of the `y[]` array (and `s[]` array, if provided) and the second dimension of `x[][]` sets the number of data points. It is a fatal error for the number of data points to be less than the number of coefficients.

## An example

This demonstrates how to fit data to the function `y = a*sin(x/10) + b*cos(x/20)`. The x values vary from 0 to 49 in steps of 1. The function `MakeFunc()` calculates a trial data set plus noise. We set `s` to 1.0, so `FitLinear()` returns the sum of squares of the errors between the fitted and input data. If you run this example, you will notice that the returned value is slightly less than the sum of squares of the added errors.

```
const NCOEF%:=2,NDATA%:=50;' coefficient and data sizes
var x[NDATA%][NCOEF%];      ' array of function information
const noise := 0.01;        ' controls how much noise we add
var data[NDATA%], err := 0;' space for our function and errors
' Generate raw data. Fit y = a*sin(x/10)+b*cos(x/20)
var coef[NCOEF%], i%, r;   ' coefficients, index, random noise
coef[0]:=1.0; coef[1]:=2;  ' set coefficients for generated data
MakeFunc(data[], coef[], x[][]); ' Generate the data, then...
for i%:=0 to NDATA%-1 do   ' ...add noise to it
   r := (rand()-0.5)*noise;' the noise to add
   data[i%] += r;          ' add noise to the data
   err := err + r*r;       ' accumulate sum of squared noise
   next;
var covar[NCOEF%][NCOEF%]; ' covariance array
var sig2, a[NCOEF%];       ' sigma, fitted coefficients
var rel[NCOEF%];           ' array for "relevance" values
sig2 := FitLinear(a[], data[], x[][], 1, covar[][], rel[]);
Message("sig^2=%g, err=%g\ncoefs=%g\nrel=%g",sig2,err,a[],rel[]);
halt;
'y is the output array (x values are 0, 1, 2...), a is the array
'of coefficients. Y = a*sin(x/10)+b*cos(x/20)
proc MakeFunc(y[], a[], x[][])
var nd%,v;                 ' coefficient index, work space
for nd% := 0 to NDATA%-1 do
   v := Sin(nd% / 10.0);   ' first funcion
   x[nd%][0] := v;         ' save the value;
   y[nd%] := a[0] * v;     ' start to build the result
   v := Cos(nd%/20.0);     ' second function
   x[nd%][1] := v;         ' save it
   y[nd%] += a[1]*v;       ' full result
next;
end;
```

**See also:**
More about curve fitting, `ChanFit()`, `FitExp()`, `FitGauss()`, `FitNLUser()`, `FitPoly()`, `FitSin()`

# FitNLUser()

This command uses a non-linear fitting algorithm to fit a user-defined function to a set of data points. The function to be fitted is of the form $y = f(x, \mathbf{a}_0, \mathbf{a}_1, \mathbf{a}_2…)$ where the $\mathbf{a}_i$ are coefficients to determine. You must be able to calculate the differential of the function $f$ with respect to each of the coefficients. You can optionally supply an array to weight each data point. The commands to implement this are:

**Set up the problem**
The first command sets the user-defined function, the number of coefficients you want to fit, the number of data points and optionally, you can set the weight to give each data point. You must call this function before you call any of the others.

```
func FitNLUser(User(ind%, a[], dyda[]), nCoef%, nData%{, const s[]|s});
```

`User()` A user-defined function which is called by the fitting routine. The function is passed the current values of the coefficients. It returns the error between the function and the data point identified by `ind%` and the differentials of the function with respect to each of the coefficients at that point. The return value should be the y data value at the index minus the calculated value of the function at the x value, using the coefficients passed in.

`ind%` The index into the data points at which to evaluate the error and differentials. If there are `n` data points, `ind%` runs from 0 to n-1. You can rely on the function being called with the same coefficients as `ind%` increments from 0 to n-1, which may be useful if you have complex functions of the coefficients to evaluate.

`a` An array of length `nCoef%` holding the current values of the coefficients. The coefficients are refreshed for each call to the user-defined function, so it is not an error to change them; however this is usually not done.

dyda    An array of length `nCoef%` which your function should fill in with the values of the partial differential of the function with respect to each of the coefficients. For example, if you were fitting $y = a_0 * \exp(-a_1 * x)$ then set `dyda[0]` $= \delta y/\delta a_0 = $ `exp(-a_1*x)` and `dyda[1]` $= \delta y/\delta a_1 = $ `-a_0*a_1*exp(-a_1*x)`.

nCoef%    The number of coefficients to fit in the range 1 to 10.

nData%    The number of data points you will be fitting. If `s[]` is provided as an array, the value of `nData%` used is the smaller of `nData%` and the length of the `s[]` array. It is a fatal error for the number of data points used to be less than `nCoef%`.

s    This argument is optional. It is either an array of weights to be given to each data point in the fit or a single weight to apply to all data points. If this value is the expected standard deviation of the y value of the data points, then the error value returned is the chi-squared value and the fit is a chi-squared fit. If this value is proportional to the expected error at the data point, then the fit is still a chi-squared fit, and the error returned is proportional to the chi-squared value. If you omit this argument, the fit is a least-squares error fit, and the error value returned is the sum of the squares of the errors in the y values.

Returns  The function returns 0. There is no other return value as all errors stop the script.

Unlike the other fitting routines, you will notice that the x and y data values are not passed into the command. Instead, the user-defined function is passed an index to the data values. It is assumed that the data is accessible by the user function.

Due to restrictions in the implementation of the script language, you cannot debug through the user-defined function. If you set a break point in it, or attempt to step into it you will get errors. We recommend that you check the returned values from the user-defined function by calling it from your own script code.

## Set coefficient values and ranges

This variant of the function sets the initial value of each coefficient and optionally sets the range of allowed values. You can call this function at any time after the setup call and before the iterate call (below) has returned 0 indicating that the fit is completed.

In this type of fitting it is very important that you give reasonable starting values for the coefficients. If you do not give starting values, the command will set them all to zero, which is unlikely to be correct.

```
func FitNLUser(coef%, val{, lo, hi});
```

coef%    The coefficient to set. The first coefficient is number 0, the last is `nCoef%`-1.

val    The initial value to assign to the coefficient. If you have set low and high limits, and the value is outside these limits, it is set to the nearer limit.

lo,hi    If present, these two values set the acceptable range of values for this coefficient. If omitted, or if both values are set to the same value, there is no limit. The value of the coefficient is tested against these limits after each iterative step, and if it exceeds a limit, it is set to that limit.

## Iterate to a solution

Once you have set up the problem and given initial values to your coefficients, you can start the iteration process that will move the coefficients from their starting values to new values that minimise the error (optionally scaled by the `s` argument).

```
func FitNLUser(a[], &err{, maxI%{, &iTer%{, covar[][]}}});
```

a[]    An array of size at least `nCoef%` that is returned holding the current set of coefficient values.

err    A real variable returned as the sum over the data points of (yx[i]-y[i]) `^2`/s[i] `^2` if `s[]` is used or holding the sum of (yx[i]-y[i]) `^2` if `s[]` is not used, where yx[i] is the value predicted from the coefficients at the x value `x[i]`.

maxI%    This is the maximum number of fitting iterations to do before returning from the function. If you omit this value, the function sets 1000. You can set any value from 1 to 10000. If you set more than 10000, the number is limited to 10000.

iTer%    An optional integer variable that is returned holding the number of iterations done before the function returned.

covar An optional two dimensional array of size at least `[nCoef%][nCoef%]` that is returned holding the covariance matrix when the fit is complete. It is not changed unless the function return value is 0 or -1.

Returns  1 if the number of iterations are exhausted and the fit has not converged, 0 if the fit has converged, -1 if the fit is not improving but the result may be OK. Other negative numbers indicate failure.

Remember that even when a minimum is found, there is no guarantee that it is the minimum. It is the best minimum that this algorithm can find given the starting point.

### Select coefficients to fit
Normally the command fits all the coefficients, but you can use this variant to select the coefficients to fit. You can use this command at any time after you have set the problem until the iteration variant returns 0 or -1.

```
func FitNLUser(const fit%[]);
```

fit%[] An array of at least `nCoef%` integers. If `fit%[i]` is 0, coefficient i is held constant, otherwise it is fitted. If all elements are 0, all arguments are fitted.

The effect of this command persists until either the iteration variant returns a value less than 1, or you set up a new problem, or you call this variant again.

### See also:
Simple example, More complex example, More about curve fitting, `FitExp()`, `FitGauss()`, `FitLinear()`, `FitPoly()`, `FitSin()`

# A simple example

The following is an example of using this set of commands to fit the user-defined function `y = a * exp(-b*x)`. In this example we generate some test data and add to it a random error. There are two coefficients to be fitted (a and b).  To use `FitNLUser()` we need to calculate the differential of the function with respect to a and b:

```
dy/da = exp(-b*x)
dy/db = -x * a * exp(-b*x)
```

These are standard results. Differential calculus is too large a subject to attempt a summary in this help. If you need to refresh your knowledge, this reference may help.

```
const NDATA%:=100, NCOEF% := 2;
var x[NDATA%],y[NDATA%],i%;
for i% := 0 to NDATA%-1 do  ' Generate data
   x[i%] := i%;              ' a:=1, b:=0.05 and add some noise
   y[i%] := exp(-0.05*i%)+(rand()-0.5)*0.01;
   next;

FitNLUser(UserFnc, NCOEF%, NDATA%); 'Link user function
FitNLUser(0, 0.5, 0.01, 2);   'Set range of amplitude
FitNLUser(1, 0.01, 0.001, 1); 'Set range of exponent

var coefs[NCOEF%], err, iter%;
i% := FitNLUser(coefs[], err, 100, iter%);
Message("fit=%d, Err=%g, iter=%d, coefs=%g",i%,err,iter%,coefs[]);
halt;

' The user-defined function: y = a * exp(-b*x);
' dy/da = exp(-b*x), dy/db = -x * a * exp(-b*x)
func UserFnc(ind%, a[], dyda[])
var xi,yi,r;
xi := x[ind%];            ' local copy of x value
yi := y[ind%];            ' local copy of y value
dyda[0] := exp(-a[1]*xi); ' differential of y with respect to a
r := dyda[0] * a[0];      ' intermediate value
dyda[1] := -xi * r;       ' differential of y with respect to b
return yi-r;
end
```

The script is in 4 sections:

1.  The header declares the number of data points and the number of coefficients plus space to store a set of x,y data points. It then fills in the data points with the function y = exp(-0.05*x) and adds some random numbers in the range -0.005 up to 0.005. This will be the test data for the fitting.

2.  The second section sets up the problem by telling `FitNLUser()` the name of the user function, the number of data points and the number of coefficients. Then for each coefficient, it sets a starting guess value and a range of acceptable values. The better your initial guess is, the more likely you are to get a useful result. Sometimes you will know a likely range of values for each coefficient, on other occasions, you may be able to make some reasonable guess from the initial data. Conversely, if you make a perversely bad initial guess, you may get results that are totally useless. It is not a good idea to set limits that are too close to the guess; sometimes a function has to chase a minimum along a tortuous route and setting very tight limits may prevent it doing this.

3.  The third section declares space for the coefficients, tells the function to iterate a maximum of 100 times and prints the results.

4.  The fourth section holds the user function. This is called once for each data point per iteration, so try to minimise the calculations required. Within each iteration, it is called with the index argument having the values 0 to the number of data points minus 1. For each data point we must fill in the differential of the fitting function with respect to each coefficient and we must return the error between the y value and the fitting function.

## Fitting a Gabor function

This demonstrates the use of the `FitNLUser()` script function. This is used to fit to a one dimensional Gabor function (the product of a Gaussian and a cosine wave) defined as:

```
y(x) = b + a*exp(-g*x*x*0.5)*Sin(l*x+p)
```

where `b` is an offset, `a` is the amplitude, `g` is twice the variance of the Gaussian, `l` is 2*pi/wavelength of the sinusoid and p is the phase of the sinusoid. There are 5 parameters to fit, being: `b`, `a`, `g`, `l`, `p`. The script starts with a header that defines the number of data points and number of coefficients, plus how much noise to add (to make this a more realistic example). It also sets the values of the coefficients that we are to attempt to fit.

```
'$FitGABOR|Demonstration of using the NL fitting to fit a Gabor function
'Author: Greg Smith, Cambridge Electronic Design Limited
const NDATA%:=100, NCOEF% := 5;
var x[NDATA%],y[NDATA%],i%, xv;
' coef#       0         1        2        3        4
const b:=0.002, a:=1.00, g:=0.01, l:=0.1, p:=0.0;
const noise := 0.04;          ' maximum noise amplitude to add

' Phase 1. Generate some sample data at equal-spaced x values. If you want
'          other than x increments of 1, set xv to i%*width where width is
'          the desired x spacing. Note that random noise is added to the
'          data values.
for i% := 0 to NDATA%-1 do  ' Generate data
   xv := i%;
   x[i%] := i%;               ' a:=1, b:=0.05 and add some noise
   y[i%] := b + a*exp(-g*xv*xv*0.5)*Sin(l*xv+p)+(rand()-0.5)*noise;
next;

' Phase 2. Set up the problem. This tells the script language the name of
'          the function to use to get the data values, then sets a range of
'          acceptable values for each argument and a starting guess for each
'          value. It is MOST important that the starting guess is reasonable.
'          If it is not, the fitting may yield stupid results.
FitNLUser(UserFnc, NCOEF%, NDATA%); 'Link user function
FitNLUser(0, 0, -0.1, 0.1);   'Set range of offset b
FitNLUser(1, 0.3, 0.5, 2);    'Set range of amplitude a
FitNLUser(2, 0.04, 0.01, 0.1);'Set range of g (1/sigma squared)
FitNLUser(3, 0.03, 0.01, 0.3);'Set range of omega (frequency) l
FitNLUser(4, 1,    -3, 3);    'Set range of the phase p

' Phase 3. Solve the problem. This tells the system to improve the initial
'          guess. Each iteration will call the User function NDATA% times.
'          The result is 0 if it worked, err is the sum of squares of the
'          difference between the fitted curve and the raw data, iterations
'          is the number of times round the function went and coefs are the
'          fitted parameters.
var coefs[NCOEF%], err, iter%, covar[NCOEF%][NCOEF%];
i% := FitNLUser(coefs[], err, 100, iter%, covar[][]);

Message("fit=%d, Err=%.4f, iterations=%d, coefs=.4f", i%, err ,iter% ,coefs[]);

'
' If you need to know the likely errors in the fitted parameters, this is
' available from the covar array (co-variance). The diagonal of the array
' holds the expected variance (sigma squared) of each parameter, on the
' assumption that the errors in the original data have a normal distribution.
'
' See the on-line documentation for the FitNLUser() script function and read the
' linked section "More about curve fitting" for additional information.
halt;

' The next section is the user-defined function. Differentials of functions with
' several factors will have common expressions that need only be calculated once.
' Although the functions is quite complicated, the calculations turn out to be
' relatively simple.
'
' The user-defined function: y = b + a*exp(-g*x*x*0.5)*Sin(l*x+p);
' dy/db = 1;
' dy/da = exp(-g*x*x*0.5)*Sin(l*x+p)
' dy/dg = -x*x*0.5*a*exp(-g*x*x*0.5)*Sin(l*x+p)
' dy/dl = a*x*exp(-g*x*x*0.5)*Cos(l*x+p)
' dy/dp = a*exp(-g*x*x*0.5)*Cos(l*x+p)

func UserFnc(ind%, a[], dyda[])
var yv,xi,yi,r, xsq2, gauss, angle;
xi := x[ind%];                     ' local copy of x value
yi := y[ind%];                     ' local copy of y value
xsq2 := xi*xi*0.5;                 ' calculate once as used twice
gauss := Exp(-a[2]*xsq2);          ' calculate once as used twice
angle := a[3]*xi+a[4];             ' used by Sin() and Cos()

dyda[0] := 1;                      ' dy/db is 1
dyda[1] := gauss*Sin(angle);       ' exp(-g*x*x*0.5)*Sin(l*x+p)
```

```
dyda[2] := -xsq2*a[1]*dyda[1];  ' -x*x*0.5*a*exp(-g*x*x*0.5)*Sin(l*x+p)
dyda[4] := a[1]*gauss*Cos(angle); ' a*exp(-g*x*x*0.5)*Cos(l*x+p)
dyda[3] := dyda[4]*xi;           ' a*x*exp(-g*x*x*0.5)*Cos(l*x+p)
yv := a[0] + a[1]*dyda[1];
return yi-yv;
end
```

# FitPoly()

This command fits $y = a_0 + a_1x + a_2x^2 + a_3x^3$ … to a set of (x,y) data points. If you can provide error estimates for each y value, you can use the covariance output from this command to provide confidence limits on the calculated coefficients and you can use the returned chi-squared value to test if the model is likely to fit the data. The command is:

```
func FitPoly(coef[], const y[], const x[]{, const s[]|s{, covar[][]}});
```

coef[]  A real array that sets the number of coefficients and returns the coefficient values. The array must be between 2 and 10 elements long. The coefficient a0 is returned in `coef[0]`, a1 in `coef[1]` and so on.

y[]  A real array of y values. The smaller of the sizes of the `x[]` and `y[]` arrays (and `s[]` array, if provided), sets the number of data points. It is a fatal error for the number of data points to be less than the number of coefficients.

x[]  A real array of x values.

s  This is an optional argument. It is either a real array holding the standard deviations of each of the `y[]` data points, or it is a real value holding the standard deviation of all of the data points. If the argument is omitted or set to zero, a least-squares error fit is performed, otherwise a chi-squared fit is done.

covar  An optional two dimensional array of size at least `[nc][nc]` (nc is the number of coefficients fitted) that is returned holding the covariance matrix.

Returns  The function returns chi-squared if `s[]` or `s` is given (and non-zero), otherwise it returns the sum of squares of the errors between the raw and fitted data.

**An example**
This example generates test data, adds random noise, then fits a polynomial to the data.

```
const NCOEF% := 5;          ' number of coefficients
const NDATA%:=50;           ' number of data points
var y[NDATA%], x[NDATA%]; ' space for x and y values for fit
const noise := 1;           ' noise to add
var err := 0.0;             ' will be sum of squares of added noise
var cf[NCOEF%], i%, r;
cf[0]:=1.0; cf[1]:=-80; cf[2]:=-2.0; cf[3]:=0.5; cf[4]:=-0.009;
MakePoly(cf[],x[],y[]);    ' generate ideal data as polynomial
for i%:=0 to NDATA%-1 do  ' now add some noise to it
   r := (rand()-0.5)*noise;
   y[i%] += r;              ' add noise to the data
   err += r*r;              ' sum of squares of added noise
   next;
var sig2, a[NCOEF%];        ' a[] will be the fitted coefficients
sig2 := FitPoly(a[], y[], x[]);
Message("sig2=%g, noise=%g\nfitted=%8.4f\nideal =%8.4f",
                                   sig2, err, a[], cf[]);
halt;
'a[]  input array of coefficients
'x[]  output x co-ordinates, y[] output data values
proc MakePoly(a[], x[], y[])
var i%,j%,xv,s;
for i% := 0 to Len(y[])-1 do
   s := 0.0;
   xv := 1;
   for j% := 0 to NCOEF%-1 do
      s += a[j%]*xv;
      xv *= i%;
      next;
   y[i%] := s;
       x[i%] := i%;
   next;
end;
```

**See also:**
More about curve fitting, ChanFit(), FitExp(), FitGauss(), FitLinear(), FitNLUser(), FitSigmoid(), FitSin()

# FitSigmoid()

This command fits a single Sigmoid function to x,y data points with an optional weight for each point. Fitting is by an iterative method. The input data is fitted to the equation:

$$y = \mathbf{a}_0 + (\mathbf{a}_1 - \mathbf{a}_0) / (1 + \exp((\mathbf{a}_2 - \mathbf{x}) / \mathbf{a}_3))$$

The fitted parameters (coefficients) are the $\mathbf{a}_i$. You can only fit 1 Sigmoid. The commands to implement this are:

**Set up the problem**
The first command sets the number of Sigmoids to fit and the data set to be fitted. You must call this function before you call any of the others.

```
func FitSigmoid(nCoef%, const y[], const x[]{, const s[]|s});
```

nCoef% The number of coefficients to fit. The only legal value is 4 for one Sigmoid.

y[]     An array of y data values. The length of the array must be at least nCoef%.

x[]     An array of x data values. The length of the array must be at least nCoef%.

s       An optional argument which is either an array with one value for each y data point or a single value for all data points. If the value is the standard deviation, then the error value returned when you iterate to find the best fit is the chi-squared value and the fit is a chi-squared fit.

        If this value is proportional to the error in the y values, then the fit is still a chi-squared fit, and the error returned is proportional to the chi-squared value. If you omit this array, the fit is a least-squares error fit, and the error value returned is the sum of the squares of the errors in the y values.

Returns  The function returns 0. There is no other return value as all errors stop the script.

The number of data points is set by the smallest of the sizes of the `y[]`, `x[]` or `s[]` (if present) arrays. The number of data points must be at least the number of coefficients. If it is not, you will get a fatal error, so check this before calling the function.

### Set coefficient values and ranges

This variant of the function sets the initial value of each coefficient and optionally sets the range of allowed values. You can call this function at any time after the setup call and before the iterate call (below) has returned 0 indicating that the fit is completed.

In this type of fitting it is very important that you give reasonable starting values for the coefficients. In particular, when fitting a Sigmoid, it is usual that the two levels (a0 and a1) are easy to determine. If you can set the levels and limit them so that they cannot overlap, the fit usually will proceed without any problems. If you do not give starting values, the command will make a simplistic guess at the fitting values. As we expect that you know more about the "right" answer than the command does, we suggest that you set the values you want.

```
func FitSigmoid(coef%, val{, lo, hi});
```

coef%    The coefficient to set. The first coefficient is number 0, the last is `nCoef%`-1.

val      The initial value to assign to the coefficient. If you have set low and high limits, and the value is outside these limits, it is set to the nearer limit.

lo,hi    If present, these two values set the acceptable range of values for this coefficient. If omitted, or if both values are set to the same value, there is no limit. The value of the coefficient is tested against these limits after each iterative step, and if it exceeds a limit, it is set to that limit.

As long as you make a reasonable estimate of the two levels, there should be no problems fitting a Sigmoid.

### Iterate to a solution

Once you have set up the problem and given initial values to your coefficients, you can start the iteration process that will move the coefficients from their starting values to new values that minimise the error (optionally scaled by the `s` argument).

```
func FitSigmoid(a[], &err{, maxI%{, &iTer%{, covar[][]}}});
```

a[]      An array of size at least `nCoef%` that is returned holding the current set of coefficient values. The first amplitude is in `a[0]`, the first centre in `a[1]`, the first sigma in `a[2]`, the second amplitude in `a[3]` and so on.

err      A real variable returned as the sum over the data points of (yx[i]-`y[i]`) `^2`/`s[i]` `^2` if `s[]` is used or holding the sum of (yx[i]-`y[i]`) `^2` if `s[]` is not used, where yx[i] is the value predicted from the coefficients at the x value `x[i]`.

maxI%    This is the maximum number of fitting iterations to do before returning from the function. If you omit this value, the function sets 1000. You can set any value from 1 to 10000. If you set more than 10000, the number is limited to 10000.

iTer%    An optional integer variable that is returned holding the number of iterations done before the function returned.

covar    An optional two dimensional array of size at least `[nCoef%][nCoef%]` that is returned holding the covariance matrix when the fit is complete. It is not changed unless the function return value is 0 or -1.

Returns  1 if the number of iterations are exhausted and the fit has not converged, 0 if the fit has converged, -1 if the fit is not improving but the result may be OK. Other negative numbers indicate failure.

Remember that even when a minimum is found, there is no guarantee that this is the minimum. It is the best minimum that this algorithm can find given the starting point.

### Select coefficients to fit

Normally the command fits all the coefficients, but you can use this variant to select the coefficients to fit. You can use this command at any time after you have set the problem until the iteration variant returns 0 or -1.

```
func FitSigmoid(const fit%[]);
```

fit%[] An array of at least `nCoef%` integers. If `fit%[i]` is 0, coefficient `i` is held constant, otherwise it is fitted. If all elements are 0, all arguments are fitted.

The effect of this command persists until either the iteration variant returns a value less than 1, or you set up a new problem, or you call this variant again.

### An example

This is a template for using these commands to fit all the coefficients:

```
const nData%:=50;                'set number of data elements
var x[nData%], y[nData%];         'space for our arrays
var s[nData%];                   'space for sigma of each point
var coefs[4], err;               'coefficients and error squared
...                              'in here goes code to get the data
FitSigmoid(4, y[], x[], s[]);    'fit one gaussian
FitSigmoid(0, 1.0, 0.2, 4);      'set base level and limit range
FitSigmoid(1, 20, 15, 25);       'set end level and range
FitSigmoid(2, 8);                'initial 50% point in X units
FitSigmoid(3, 0.5);              'initial slope
repeat
   DrawMyData(coefs[], x[], y[]);  'Some function to show progress
until FitSigmoid(coefs[], err, 1) < 1;
DrawMyData(coefs[], x[], y[]);   'Show the final state
```

### See also:

More about curve fitting, `ChanFit()`, `FitExp()`, `FitLinear()`, `FitGauss()`, `FitNLUser()`, `FitPoly()`, `FitSin()`

# FitSin()

This command fits multiple sinusoids to arrays of x,y data points, with an optional weight for each point. Fitting is by an iterative method. The input data is fitted to the equation:

$$y = \mathbf{a}_0 \sin(\mathbf{a}_1 x + \mathbf{a}_2) + \mathbf{a}_3 \sin(\mathbf{a}_4 x + \mathbf{a}_5) + \dots \{+ \mathbf{a}_{3n}\}$$

The fitted coefficients are the $\mathbf{a}_i$. Angles are evaluated in radians. You can fit up to 3 sinusoids and an optional offset. Although the function is given in terms of sine functions, you can easily convert to cosines by subtracting $\pi/2$ from the phase angle ($\mathbf{a}_2$, $\mathbf{a}_5$, $\mathbf{a}_8$) after the fit. The commands to implement this are:

### Set up the problem

The first command sets the number of sinusoids to fit and the data set to be fitted. You must call this function before you call any of the others.

```
func FitSin(nCoef%, const y[], const x[]{, const s[]|s});
```

nCoef% The number of coefficients to fit. The only legal values are 3, 6 and 9 for one, two and three sinusoids or 4, 7 and 10 to include an offset as the last coefficient.

y[] An array of y data values. The length of the array must be at least `nCoef%`.

x[] An array of x data values. The length of the array must be at least `nCoef%`.

s An optional argument which is either an array with one value for each y data point or a single value for all data points. If the value is the standard deviation, then the error value returned when you iterate to find the best fit is the chi-squared value and the fit is a chi-squared fit.

If this value is proportional to the error in the y values, then the fit is still a chi-squared fit, and the error returned is proportional to the chi-squared value. If you omit this array, the fit is a least-squares error fit, and the error value returned is the sum of the squares of the errors in the y values.

Returns  The function returns 0. There is no other return value as all errors stop the script.

The number of data points is set by the smallest of the sizes of the `y[]`, `x[]` or `s[]` (if present) arrays. The number of data points must be at least the number of coefficients. If it is not, you will get a fatal error, so check this before calling the function.

## Set coefficient values and ranges

This variant of the function sets the initial value of each coefficient and optionally sets the range of allowed values. You can call this function at any time after the setup call and before the iterate call (below) has returned 0, indicating that the fit is complete.

In this type of fitting it is very important that you give reasonable starting values for the coefficients. In particular, when fitting multiple sinusoids you will usually either know, or have a good idea of the frequencies. You should limit the range of each frequency so that they cannot overlap. If you can do this, the fit will proceed quickly. If you do not give starting values, the command will make a simplistic guess at the fitting values. As we expect that you know more about the "right" answer than the command does, we suggest that you set the values you want.

```
func FitSin(coef%, val{, lo, hi});
```

coef%   The coefficient to set. The first coefficient is number 0, the last is `nCoef%`-1.

val     The initial value to assign to the coefficient. If you have set low and high limits, and the value is outside these limits, it is set to the nearer limit.

lo,hi   If present, these two values set the acceptable range of values for this coefficient. If omitted, or if both values are set to the same value, there is no limit. The value of the coefficient is tested against these limits after each iterative step, and if it exceeds a limit, it is set to that limit.

## Iterate to a solution

Once you have set up the problem and given initial values to your coefficients, you can start the iteration process that will move the coefficients from their starting values to new values that minimise the error (optionally scaled by the `s` argument).

```
func FitSin(a[], &err{, maxI%{, &iTer%{, covar[][]}}});
```

a[]     An array of size at least `nCoef%` that is returned holding the current set of coefficient values. The first amplitude is in `a[0]`, the first frequency in `a[1]`, the first phase angle in `a[2]`, the second amplitude in `a[3]` and so on.

err     A real variable returned as the sum over the data points of $(yx[i]-y[i])$ $^2/s[i]$ $^2$ if $s[]$ is used or holding the sum of $(yx[i]-y[i])$ $^2$ if $s[]$ is not used, where yx[i] is the value predicted from the coefficients at the x value $x[i]$.

maxI%   This is the maximum number of fitting iterations to do before returning from the function. If you omit this value, the function sets 1000. You can set any value from 1 to 10000. If you set more than 10000, the number is limited to 10000.

iTer%   An optional integer variable that is returned holding the number of iterations done before the function returned.

covar   An optional two dimensional array of size at least `[nCoef%][nCoef%]` that is returned holding the covariance matrix when the fit is complete. It is not changed unless the function return value is 0 or -1.

Returns  1 if the number of iterations are exhausted and the fit has not converged, 0 if the fit has converged, -1 if the fit is not improving but the result may be OK. Other negative numbers indicate failure.

Even when a minimum is found, there is no guarantee that this is the minimum, only that it is the best minimum that this algorithm can find given the starting point.

## Select coefficients to fit

Sometimes you may wish to hold some coefficients fixed while you fit others. Normally the command will fit all the coefficients, but you can use this command variant to select the coefficients to fit. You can use this command at any time after you have set the problem until the iteration variant returns 0 or -1.

```
func FitSin(const fit%[]);
```

fit%[]  An array of at least `nCoef%` integers. If `fit%[i]` is 0, coefficient `i` is held constant, otherwise it is fitted. If all elements are 0, all arguments are fitted.

The effect of this command persists until either the iteration variant returns a value less than 1, or you set up a new problem, or you call this variant again. For a sinusoidal fit it is likely that you will know the frequency to fit, so you may well hold this constant.

### An example

The following is a template for using this command (assuming you don't want to fit the frequency, which we assume you know).

```
const nData%:=50;          'set number of data elements
var x[nData%], y[nData%]; 'space for our arrays
var s[nData%];             'space for sigma of each point
var fit%[3];               'we want to hold the frequency
var coefs[4];              'space for coefficients
var err;                   'will hold error squared
...                        'in here goes code to get the data
FitSin(3, y[], x[], s[]); 'fit one sinusoid
'Note that we let the phase take any value
FitSin(0, 1.0, 0.2, 4);   'set amplitude and limit range
FitSin(1, .02, .01, .03); 'set frequency
FitSin(2, 0., 0.3, 1.9);  'set width and limit range
'Now we say that we don't want to fit the frequency
ArrConst(fit%[],1);        'set all elements to 1
fit%[1] := 0;              'but not element 1 (=frequency)
FitSin(fit%[]);            'so the frequency is fixed
repeat
   DrawMyData(coefs[], x[], y[]);  'Some function to show progress
until FitSin(coefs[], err, 1) < 1;
DrawMyData(coefs[], x[], y[]);      'Show the final state
```

**See also:**
More about curve fitting, `FitExp()`, `FitGauss()`, `FitLinear()`, `FitNLUser()`, `FitSigmoid()`, `FitPoly()`

## FitValue()

This function returns the value at a particular x position of the fitted function set by the last `FitData()` command.

`Func FitValue(x{, &valid%});`

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

valid% If present, this integer variable is set to 1 if the returned value is valid, 0 if not.

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

**See also:**
`FitCoef()`, `FitData()`, `FitExp()`, `ChanFitValue()`

## Floor()

Returns the next lower integral number of the real number or array. `Floor(4.7)` is `4.0`, `Floor(4)` is `4`. `Floor(-4.7)` is `-5`.

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

x     A real number or a real array.

Returns 0 or a negative error code for an array or the next lower integral value.

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

## FocusHandle()

This function returns the view handle of the script-controllable window with the input focus (the active window). Unlike `FrontView()`, it can return any type of window, for example the toolbar.

`Func FocusHandle();`

Returns  The handle of a window that the script can manipulate, or 0 if the focus is not in such a window.

**See also:**
FrontView()

# FontGet()

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

```
Func FontGet(&name$, &size, &flags%{, style%{, &fore%{, &back%}}});
```

name$  This string variable is returned holding the name of the font.

size  The real number variable is returned holding the point size of the font.

flags%  Returned holding the sum of the style values: 1=Italic, 2=Bold, 4=Underline, 8=Force upper case, 16=force lower case. Only one of 8 or 16 will be returned.

style%  Text-based views all have a default style (style%=32) that is used as the basis of all other styles, a style for line numbers (style%=33), plus a number of additional styles that are used to highlight keywords and the like (in output sequencer and script windows). The style% values for these additional styles starts at 0 and runs upwards. For simple text views the only other style number is 0 for normal text, you can find a list of styles for Script views here and for Sequencer views here, or you can see the list of styles for each view type in the Display page of the Preferences dialog in the Text view settings. If you omit style% the settings for the default style (32) are returned.

fore%  This value returns the foreground colour of the style. Bits 0-7 hold the red intensity, bits 8-15 hold the green and bits 16-23 hold the blue.

back%  This value returns the background colour of the style.

Returns  The function returns 0 if all was well or a negative error code. If an error occurs, the variables are not changed.

The arguments from style% onwards and the flag% values 4, 8 and 16 only apply to text-based views and are new in version 4.

**See also:**
FontSet(), TabSettings()

# FontSet()

This function sets the font for the current view. Text-based views (text, sequencer and script) normally avoid proportionally spaced fonts as they did not display correctly before Signal version 4. Arguments from style% onwards are for text-based views.

```
Func FontSet(name$|code%, size, flags%{, style%{, fore%{, back%}}});
```

name$  A string holding the font name to use or you can specify a font code:

code%  This is an alternative method of specifying a font. We recognise these codes:
  0  The standard system font, whatever that might be
  1  A non-proportionally spaced font, usually Courier-like
  2  A proportionally spaced non-serifed font, such as Helvetica or Arial
  3  A proportionally spaced serifed font, such as Times Roman
  4  A symbol font
  5  A decorative font, such as Zapf-Dingbats or TrueType Wingdings

size  The point size required. Your system may limit the allowed range.

flags%  The sum of the style values to set: 1=Italic, 2=Bold, 4=Underline, 8=Force upper case, 16=force lower case. If both 8 and 16 are set, 16 is ignored. Values from 4 upwards are only supported by text-based views.

style%  Text-based views all have a default style (style%=32) that is used as the basis of all other styles, a style for line numbers (style%=33), plus a number of additional styles that are used to highlight keywords and the like (in output sequencer and script windows). The style% values for these additional styles starts at

0 and runs upwards. For simple text views the only other style number is 0 for normal text, you can find a list of styles for Script views here and for Sequencer views here, or you can see the list of styles for each view type in the Display page of the Preferences dialog in the Text view settings. If you omit `style%`, the default style is changed. Set the value -1 to set all styles to the values you give.

`fore%`   This sets the style foreground colour or is set to -1 to make no change. Bits 0-7 hold the red intensity, bits 8-15 hold the green and bits 16-23 hold the blue. Bits 24-31 are 0. It is convenient to code this as a hexadecimal number, for example:

```
const red%:=0x0000ff, green%:=0x00ff00, blue%:=0xff0000;
const gray%:=0x808080, yellow% :=0x00ffff;
```

`back%`   This sets the background colour of the style in the same format as `fore%`.

Returns   The function returns 0 if the font change succeeded, or a negative error code.

**See also:**
Edit setup dialog, `FontGet(), TabSettings()`

# Frac()

Returns the fractional part of a real number or replaces an array of reals with its fractional parts.

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

`x`   A real number or an array of reals.

Returns   For arrays, it returns 0 or a negative error code. If `x` is not an array it returns a real number equal to `x-Trunc(x)`. E.g. `Frac(4.7)` is 0.7, `Frac(-4.7)` is -0.7.

**See also:**
`Trunc(), Round()`

# Frame...()

# Frame()

This function gets or sets the current frame in a data view.

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

`frame%`   If this is present and in range, the current frame changes to the new number.

Returns   The frame number for the view at the time of the call.

**See also:**
`FrameComment$(), FrameCount(), FrameFlag(), FrameTag(), FrameUserVar()`

# FrameAbsStart()

This function obtains the absolute start time for the current frame in a data view, it can also set the start time for frames in a memory view. The absolute start time for a frame is the time for time zero in a frame relative to the time at which sampling was started.

```
Func FrameAbsStart({new});
```

`new`   If this is present and the current view is a memory view not yet saved to disk, it sets the new absolute start time for the current frame.

Returns   The absolute start time, in seconds, of the current frame in the current data view.

**See also:**
`Frame(), SampleStart()`

# FrameComment$()

This function gets or sets the comment for the current frame. This is a string of up to 72 characters that is stored with each frame.

```
Func FrameComment$({c$});
```

c$     If this is present it provides a new comment to store with the frame.

Returns  The frame comment at the time of the call.

**See also:**
```
FileComment$(), FrameState(), FrameTag(), FrameUserVar()
```

# FrameCount()

This function returns the number of frames in a data document.

```
Func FrameCount();
```

Returns  Number of frames (sweeps) in the file or memory view.

**See also:**
```
Frame(), Sweeps()
```

# FrameFlag()

This command turns a frame flag on or off or retrieves the current setting of a flag from the specified frame. This function will fail with a run-time error if used to change a flag with read-only data, you can check for read-only data using the `Modified()` function.

```
Func FrameFlag(frm%|frm$|frm%[], flag%{, set%});
```

frm%    Frame number or a negative code as follows:
  -1   All frames in the file.
  -2   The current frame.
  -3   Only tagged frames.
  -6   Only untagged frames.

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

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

flag%   The flag number (1..32). For CFS files not created by Signal, only flags 1 to 4 and flag 16 are present, though the other flags can be used while the frame is held in memory.

set%    If non-zero, this sets flag number `flag%` on in the current frame. If zero, it clears the flag. In a file not created by Signal only flags 1,2,3,4 or 16 can be permanently changed.

Returns  1 if the flag number `flag%` is set in the last frame specified when the function was called or zero if not. Returns -1 if no frames are found to match the specification.

**See also:**
```
FrameComment$(), FrameState(), FrameTag(), FrameUserVar()
```

# FrameGapFree()

This function returns a flag indicating if the current data view file was collected using sampling in gap-free mode, it can also set the gap-free flag for a memory view.

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

new%    If this is present and the current view is a memory view not yet saved to disk, it sets the new gap free flag for the current frame. A value of zero turns the gap free flag off, a value of 1 turns it on.

Returns The gap free flag in the current data view file at the time of the function call.

**See also:**
```
SampleStart()
```

# FrameGetIntVar()

This function reads a CFS frame variable of integer type from the current frame.

```
Func FrameGetIntVar(name${, &nVar%{, &units${, nType%}}});
```

name$ The name of the variable to look for. This string is not case sensitive but every character including spaces must match exactly.

nVar% If present this returns the variable number, -1 if not found, or a negative error code.

units$ If present this returns the units for the variable.

nType% If present this returns a code for the CFS type of an integer variable:

    0: INT1, 1: WRD1, 2: INT2, 3: WRD2, 4: INT4:

Returns The function returns the value of the variable if the operation was a success, otherwise zero.

**See also:**
```
FileGetIntVar(), FileGetRealVar(), FileGetStrVar$(), FileVarCount(), FileVarInfo(),
FrameGetRealVar(), FrameGetStrVar$(), FrameVarCount(), FrameVarInfo(), FrameUserVar()
```

# FrameGetRealVar()

This function reads a CFS frame variable of real type from the current frame.

```
Func FrameGetRealVar(name${, &nVar%{, &units$}});
```

name$ The name of the variable to look for. This string is not case sensitive but every character including spaces must match exactly.

nVar% If present this returns the variable number, -1 if not found, or a negative error code.

units$ If present this returns the units for the variable.

Returns The function returns the value of the variable if the operation was a success, otherwise zero.

**See also:**
```
FileGetIntVar(), FileGetRealVar(), FileGetStrVar$(), FileVarCount(), FileVarInfo(),
FrameGetIntVar(), FrameGetStrVar$(), FrameVarCount(), FrameVarInfo(), FrameUserVar()
```

# FrameGetStrVar$()

This function reads a CFS frame variable of string type from the current frame.

```
Func FrameGetStrVar$(name${, &nVar%{, &units$}});
```

name$ The name of the variable to look for. This string is not case sensitive but every character including spaces must match exactly.

nVar% If present this returns the variable number, -1 if not found, or a negative error code.

units$ If present this returns the units for the variable.

Returns The function returns the string contents of the variable if the operation was a success, otherwise an empty string.

**See also:**
```
FileGetIntVar(), FileGetRealVar(), FileGetStrVar$(), FileVarCount(), FileVarInfo(),
FrameGetIntVar(), FrameGetRealVar(), FrameVarCount(), FrameVarInfo(), FrameUserVar()
```

# FrameList()

This function generates an array of specified frame numbers from the current view.

```
Func FrameList(list%[], sFrm%{, eFrm%{, mode%}});
Func FrameList(list%[], frm$|frm%[]{, mode%});
```

list%    An integer array to fill with frame numbers. The first element of the array, `list%[0]`, is set to the number of frames returned, and the remaining elements in the array are frame numbers. If the array is too short, enough frames are returned to fill the array.

sFrm%    First frame to include. This option returns a range of frames. `sFrm%` can also be a negative code as follows:

   -1   All frames in the file are included.

   -2   The current frame.

   -3   Frames must be tagged.

   -6   Frames must be untagged.

eFrm%    Last frame to include. If this is -1 the last frame number in the data file is used. This argument is ignored if `sFrm%` is a negative code.

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

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

mode%    If `mode%` is present it is used to supply an additional criterion for including each frame in the range, list or specification. If `mode%` is absent all frames are included. The modes are:

   0-n  Frames must have a state matching the value of `mode%`.

   -1   All frames in the range list are included.

   -2   Only the current frame in the view is included.

   -3   Frames must be tagged.

   -6   Frames must be untagged.

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

**See also:**
Frame lists, FrameState(), FrameTag(), FrameUserVar(), ExportFrameList(), ProcessFrames()

# FrameMean()

This command turns the frame mean flag on or off or gets the setting of the mean flag for the specified frame or frame type.

```
Func FrameMean(frm%|frm$|frm%[]{, on%});
```

frm%     One frame number or a negative code as follows:

   -1   All frames in the file.

   -2   The current frame.

   -3   All tagged frames.

   -6   All untagged frames.

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

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

on%      A value of 1 marks the frame as a mean, zero marks it as a total.

Returns  1 if the last frame specified was a mean when the function was called or zero if not. Returns -1 if no frames are found to match the specification.

**See also:**
`FrameComment$(), FrameFlag(), FrameState(), FrameUserVar(), Sweeps()`

# FrameSave()

This command saves changed frame data in a file view back into the file, bypassing the usual interactive process controlled by the Data page in the Preferences dialog. It can also be used to discard changes to ensure that the user is not promped to save them. This command can only be used on frames already present on disk; appended frames and memory view frames will be saved as part of `FileSave` or `FileClose`. This function will fail with a run-time error if used to save changes with read-only data, you can check for read-only data using the `Modified()` function.

```
Func FrameSave({no%});
```

`no%`   If present and non-zero, this causes changed data to be discarded by marking the data as unchanged. If the parameter is not present or set to zero the function causes the changed data to be written back to the disk file.

Returns  One if changed frame data was written to disk, zero if nothing or only frame variables were written, or a negative error code.

**See also:**
`FileExportAs(), FileSave(), FileClose(), Frame(), Modified()`

# FrameState()

This command sets or gets the state code value for the specified frame or frames. This function will fail with a run-time error if used to change the state with read-only data, you can check for read-only data using the `Modified()` function.

```
Func FrameState(frm%|frm$|frm%[]{, new%});
```

`frm%`   One frame number or a negative code as follows:

-1   All frames in the file

-2   The current frame

-3   All tagged frames

-6   All untagged frames

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

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

`new%`   If present this sets the state stored with each frame specified. For values above 9 this is only effective in a file created by Signal.

Returns  The state of the last frame specified when the function was called. Returns -1 if no frames are found to match the specification.

**See also:**
`FrameComment$(), FrameFlag(), FrameTag(), FrameUserVar()`

# FrameTag()

This command turns the frame tag on or off, or gets the setting of the tag, for the specified frame or frame type. This function will fail with a run-time error if used to change the tag with read-only data, you can check for read-only data using the `Modified()` function.

```
Func FrameTag(frm%|frm$|frm%[]{, on%});
```

`frm%`   One frame number or a negative code as follows:

-1   All frames in the file.

-2   The current frame

-3   All tagged frames

-6    All untagged frames

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

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

on%    A value of 1 tags the frame, zero untags it.

Returns   1 if the last frame specified was tagged when the function was called or zero if not. Returns -1 if no frames are found to match the specification.

**See also:**
```
FrameComment$(), FrameFlag(), FrameState(), FrameUserVar()
```

# FrameUserVar()

This command gets or sets one of the user frame variable values stored with the current frame of the file associated with the current file or memory view, a second form can be used to read back or change the name and units of a variable - which applies to all frames. This function will fail with a run-time error if used to change a user variable with read-only data, you can check for read-only data using the `Modified()` function.

Frame user variables are a subset of the frame variables in a file that can be both read and changed by the script language. There are sixteen of them, all are floating point values, and they are initially named User1 to User16 with blank units. Signal makes use of frame variables for various purposes, for example to store information about the setup of an auxiliary states device, but the user frame variables are specifically available for general use. It is necessary to be very cautious about changing non-user frame variables, but you can make use of the user variables freely and safely by using this function and can rename them and alter the units to reflect the information stored within them.

```
Func FrameUserVar(n%{, val});
Func FrameUserVar(n%, name$, units${, set%});
```

n%    The user variable number, between 1 and 16.

val    If present this is the new value for user variable number n.

name$   This string sets or is updated with the name of user frame variable n%. When used to set the name this should not be more than 14 characters long.

units$   This string sets or is updated with the units of user frame variable n%. When used to set the units this should not be more than 8 characters long.

set%    If present and set to 1, the user variable name and units will be set using name$ and units$, otherwise the current name and units will be returned.

Returns   The value of frame user variable number n before the call.

**See also:**
```
FrameComment$(), FrameFlag(), FrameState(), FrameTag(), FrameGetIntVar(),
FrameGetStrVar$(), FrameVarCount(), FrameVarInfo()
```

# FrameVarCount()

This function counts CFS frame variables in the data file. Frame variables are extra values attached to each frame in a CFS data file. These are used by Signal for various purposes, for example to hold the frame state and absolute start time. Some of the frame variable are user variables and can be both read and written-to. A Signal script can read the values of the other frame variables but is not allowed to change them.

```
Func FrameVarCount();
```

Returns   The number of frame variables in the data file for file views, the number of frame user variables for memory views, or zero for other types of view.

**See also:**

```
FileGetIntVar(), FileGetRealVar(), FileGetStrVar$(), FileVarCount(), FileVarInfo(),
FrameGetIntVar(), FrameGetRealVar(), FrameGetStrVar$(), FrameVarInfo(),
FrameUserVar()
```

# FrameVarInfo()

This function reads the name, type and optionally the units of a CFS frame variable. Frame variables are extra values attached to each frame in a CFS data file. These are used by Signal for various purposes, for example to hold the frame state and absolute start time. Some of the frame variable are user variables and can be both read and written-to. A Signal script can read the values of the other frame variables but is not allowed to change them.

`Func FrameVarInfo(nVar%, &name${, &units$});`

nVar%  This is the variable number for which information is required. The first frame variable is number zero, the first user variable is number five for file views and zero for memory views.

name$  This is returned holding the name of the variable, which can be used in the commands for reading the frame variable values.

units$  If provided, this is returned holding the units for the variable.

Returns  The function returns the type of the variable or -1 if the variable was not found or is of unknown type. The variable type codes are as follows:

    0    An integer variable which can be read using `FrameGetIntVar()`.

    1    A floating point variable which can be read using `FrameGetRealVar()`.

    2    A string variable which can be read using `FrameGetStrVar$()`.

**See also:**
```
FileGetIntVar(), FileGetRealVar(), FileGetStrVar$(), FileVarCount(), FileVarInfo(),
FrameGetIntVar(), FrameGetRealVar(), FrameGetStrVar$(), FrameVarCount(),
FrameUserVar()
```

# FrontView()

The front view is the Data, XY, Grid or text-based view that decides which menu appears in the main window. You can use this command to get the active view (view with the input focus), or to set it, or to move any other script-controlled view to the front (to make it visible). When a view becomes the front view, it is moved on top of other views in Signal and made the current view. If an invisible or iconized view is made the front view, it is made visible automatically, (equivalent to `WindowVisible(1)`).

You can also use this command to move other windows (for example the cursor windows) to the front and make them the current view, but the view handle of such windows are not returned by this command. Use `FocusHandle()` to return these windows.

`Func FrontView({vh%});`

vh%  Either 0 or omitted to return the front view handle, the handle of the view to be set, or n, meaning the nth duplicate of the data view associated with the current view. If vh% is the view handle of a data, XY, grid or text-based view, this makes it the Active view. If vh% is the handle of any other view, that view is moved to the front.

Returns  0 if there are no visible views, -1 if the view handle passed is not a valid view handle, otherwise it returns the view handle of the view that was at the front.

**See also:**
```
View(), Window(), WindowVisible(), FocusHandle()
```

# G

# GammaP() and GammaQ()

These functions return the incomplete gamma function *P(a, x)*. It is defined as:

the integral between 0 and x of `exp(-t)*pow(t,a-1)` with respect to `t` divided by Gamma(a)

This is named `GammaP()` in the script. We also define the complement of this function, `GammaQ(a,x)`, which is `1.0-GammaP(a,x)`. From them are obtained the error function, the cumulative Poisson probability function and the Chi-squared probability function.

The error function *erf(x)* = `GammaP(0.5, x*x)`

The cumulative Poisson probability function relates to a Poisson process of random events and is the probability that, given an expected number of events `r` in a given time period, the actual number was greater than or equal to `n`. This turns out to be `GammaP(n,r)`. Also, the probability that there are less than `n` events is `GammaQ(n,r)`.

The Chi-squared probability function is useful where we are fitting a model to data. Given a fitting function that fits the data with n degrees of freedom (if you have `nData` data points and `nCoef` coefficients you usually have `nData-nCoef` degrees of freedom), and given that the errors in the data points are normally distributed, the probability of a Chi-squared value less than `chisq` is `GammaP(n/2, chisq/2)`. Similarly, the probability of a `chisq` value at least as large as `chisq` is `GammaQ(n/2, chisq/2)`. So, if you know the chi-squared value from a fitting exercise, you can ask "What is the probability of getting this value (or a greater one) given that my model fits the data?" If the probability is very small, it is likely that your model does not fit the data, or your fit has not converged to the correct solution.

```
Func GammaP(a, x);
Func GammaQ(a, x);
```

a       This must be positive, it is a fatal error if it is not.

x       This must be positive, it is a fatal error if it is not.

Returns  These functions return the incomplete Gamma function and the complement of the incomplete Gamma function.

**See also:**
`LnGamma()`

# GammaQ()

The complement of `GammaP()`; `GammaQ(a,x)` is `1.0-GammaP(a,x)`.

```
Func GammaQ(a, x);
```

a       This must be positive, it is a fatal error if it is not.

x       This must be positive, it is a fatal error if it is not.

Returns  The complement of the incomplete Gamma function.

**See also:**
`GammaP()`

# Grd...()

# GrdAlign()

This command sets the alignment of text in grid view columns.

```
Func GrdAlign(align%, col%)
```

align% Set -1 to make no change, 0 for left aligned, 1 for centred and 2 for right aligned columns.

col%    The column number to align or -1 for all columns that intersect the currently selected cells in the grid or -2 for all columns.

Returns   The alignment of the lowest numbered column defined by `col%` as 0 for left, 1 for centred and 2 for right aligned before any change made by this command is applied.

**See also:**
`GrdGet(), GrdSet(), GrdColWidth(), GrdShow(), GrdSize()`

# GrdColWidth()

This command reads or sets the width of a nominated column, or all columns that intersect with the current selection. There are two variants:

```
Func GrdColWidth(col%{, width%);
Func GrdColWidth(col%, text$);
```

`col%`   The column to use or -1 to use all columns that intersect the currently selected cells in the grid or -2 for all columns.

`width%` Optional. If present, and positive it sets the width of the columns nominated by `col%` in pixels. You are allowed to set a 0 width, which hides the column(s). You can also set

-1   Set the default column width.

-2   Optimise the column width based on the contents.

`text$`   Set the column wide enough to display this text, based on the current font set for the body of the grid (not the headings).

Returns   The width of the first column set by `col%`, in pixels or -1 if `col%` is too large to refer to a column.

**See also:**
`GrdGet(), GrdSet(), GrdAlign(), GrdShow(), GrdSize()`

# GrdGet()

Collect data from the current grid view. You can read back data from a single cell, or from a column or a row, or from a square region of the grid.

```
Func GrdGet(&text$|text$[]{[]}, col%, row%);
```

`text$`   This is either a simple string variable or a string vector or a string matrix. With a simple variable, you read back one cell from `row%`, `col%`. With a vector, you read back a column of data starting at `row%`, `col%`. With a matrix, you read back a rectangular grid of items starting at `row%`, `col%`. To read back a row of data into a vector, use the transpose operator on the vector (`trans(text$[])` or `text$`).

`col%`   This sets the start column. The first data column is 0. The row headings are always the row numbers so these cannot be accessed.

`row%`   This sets the start row. You can read back the column headers by setting this to -1. The first data column is 0.

Returns   The number of elements of `text$` that were set.

If either `row%` or `col%` exceed the size of the grid, nothing is read and 0 is returned.

**See also:**
`GrdSet(), GrdColWidth(), GrdAlign(), GrdShow(), GrdSize()`

# GrdSet()

Set data in the current grid view. You can write data to a single cell, or to a column or a row or to a square region of the grid.

```
Func GrdSet(text$|text$[]{[]}, col%, row%);
```

`text$`   This is either a simple string or a string vector or a string matrix. With a simple variable, you set one cell at `row%`, `col%`. With a vector, you set a column of data starting at `row%`, `col%`. With a matrix, you set a rectangular grid of items starting at `row%`, `col%`. To set a row of data from a vector `text$[]`, use the transpose operator on the vector (`trans(text$)` or `text$`).

`col%`     This sets the start column. The first data column is 0. The row headings are always the row numbers so these cannot be accessed.

`row%`     This sets the start row. You can set the column headers by setting this to -1. The first data column is 0. If you set a blank column header, the displayed heading reverts to the default.

Returns  The number of cells of the grid that were set.

If either `row%` or `col%` exceed the size of the grid, nothing is set and 0 is returned.

**See also:**
```
GrdGet(), GrdColWidth(), GrdAlign(), GrdShow(), GrdSize()
```

# GrdShow()

Show, hide and report visibility of column and row headers.

```
Func GrdShow({cHead%, rHead%});
```

`cHead%` Omit or set -1 for no change, 0 to hide and 1 to show the column header.

`rHead%` Omit or set -1 for no change, 0 to hide and 1 to show the row header.

Return   The original state as the sum of 1 (for column headers visible) and 2 (for row headers visible), so possible return values are 0, 1, 2 or 3.

**See also:**
```
GrdGet(), GrdSet(), GrdColWidth(), GrdAlign(), GrdSize()
```

# GrdSize()

This function resizes the grid or reports the grid size. If you change the size, existing cell contents in surviving cells are preserved. There are two command variants.

**Change grid size**
```
Func GrdSize(cols%, rows%);
```

`cols%`    The number of columns to set in the grid or -1 for no change.

`rows%`    The number of rows to set in the grid or -1 for no change.

Returns  0 if done or a negative error code (for example, too many cells).

**Get grid size**
The second variant returns the number of rows or columns:

```
Func GrdSize({get%});
```

`get%`     Set 0 or omit to return the number of columns. Set -1 to return the number of rows.

Returns  The information requested by `get%`.

**See also:**
```
GrdGet(), GrdSet(), GrdColWidth(), GrdAlign(), GrdShow()
```

# Grid()

This function turns the background grid on and off for the current data or XY view and returns the state of the grid. The grid is a mesh of lines that follow the big and small ticks on the x and y axes that is drawn on top of the data view background and under the data. Separate control of x and y grid lines was added at version 4.07.

```
Func Grid({on%});
```

`on%`      Optional, sets the grid state. Omit or set -1 for no change, 0 = no grid, 1 = both x and y grid,  2=x grid only, 3=y grid only.

Returns  The state of the grid at the time of the call as 0-3 or a negative error code. Changes 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.

# H

# HCursor...()

# HCursor()

This function returns the y axis position of a horizontal cursor, and optionally sets a new position. You can get and set positions of cursors attached to invisible channels or channels that have no y axis.

```
Func HCursor(num%{, where{, chan%}});
```

num%  The cursor to use, from 1 to n. It is an error to attempt this operation on an unknown cursor.

where  If this parameter is given it sets the new y axis position of the cursor.

chan%  If this parameter is given, it sets the channel number (1 to n).

Returns  The function returns the y axis position of the cursor at the time of the call, or zero for a non-existent cursor number.

**See also:**
```
Cursor(), HCursorChan(), HCursorDelete(), HCursorLabel(), HCursorLabelPos(),
HCursorNew(), HCursorRenumber()
```

# HCursorActive()

This function sets and retrieves the parameters used by an active horizontal cursor in taking a measurement from the channel on which the cursor is placed and moving to the measured level. There are two forms of the function, the first one sets the active cursor mode and parameters, the second is used to read the mode and parameters back.

```
Func HCursorActive(num%{, mode% {,start|start$|expr$, end|end$ {,factor
{,seq%}}}});
```

This form of the function is used to set active horizontal cursor modes and parameters. Note that the use of some of the parameters varies according to the cursor mode that is set. The function parameters are:

num%  The horizontal cursor number for which we are setting the active mode parameters, from 1 to 10. If only this argument is provided the function returns the current cursor mode and makes no changes.

mode%  The active cursor mode (see the main active horizontal cursor documentation for a complete description of the modes). The possible values of mode% are:

| | | | | | |
|---|---|---|---|---|---|
| 0 | Static | 3 | Mean level | 6 | Extreme |
| 1 | Value at point | 4 | Maximum | 7 | Mean level + (SD * factor) |
| 2 | Expression | 5 | Minimum | | |

start  The start time for the measurement in seconds, or the time for the Value at point measurement.

start$ The start time for the measurement as a string. Time expressions such as `"XLow()+0.2"` can be used for the start time, numbers used in the string are assumed to be in seconds unless a specific modifier is used thus: `"XLow()+0.2ms"`.

expr$ For Expression mode only this is a string expression relating to the channel on which the cursor is placed such as `"HCursor(1)"` or `"Mean(0.1, 0.2)"`.

end The end time for the measurement, in seconds.

end$ The end time for the measurement as a string, again time expressions can be used.

factor This is the scale factor used in mode 7.

seq% This is a number specifying when the measurement should be evaluated and the cursor repositioned relative to the sequence of active vertical cursor repositioning that occurs when the current frame changes or vertical cursor 0 is iterated. Set seq% to 0 for automatic sequencing, 1 for evaluation before all the vertical cursors and 2 for after all the vertical cursors.

Returns The active cursor mode set at the time that the function was called.

**Func HCursorActive(num%{, item% {, &val$}});**

This form of the function is used to read back the active cursor mode and parameters. Note that the use of some of the parameters varies according to the cursor mode that is set. The function parameters are:

num% The horizontal cursor number for which to retrieve information, from 1 to 10.

item% This specifies the parameter for which to retrieve the current value:

| -1 | mode% | -3 | end$ | -5 | factor |
|---|---|---|---|---|---|
| -2 | start$ | -4 | expr$ | -6 | seq% |

If item% is omitted it is treated as -1.

val$ This optional string variable will be updated with the parameter value as a string when item% is -2, -3 or -4.

Returns The numerical value of the selected value. For items -2, -3 and -4 this value is generated by parsing the associated string so it will reflect the position of any cursors or data values used, if the string parsing fails zero is returned.

**See also:**
ChanMeasure(), ChanValue(), HCursor(), HCursorDelete(), HCursorNew(), HCursorValid(), CursorActive()

# HCursorChan()

This function returns the channel number that a particular horizontal cursor is currently attached to.

**Func HCursorChan(num%);**

num% The horizontal cursor number, from 1 to n.

Returns It returns the channel number that the cursor is attached to, or 0 if this cursor is not attached to any channel or if the channel number is out of the allowed range.

**See also:**
HCursor(), HCursorDelete(), HCursorLabel(), HCursorLabelPos(), HCursorNew(), HCursorRenumber()

# HCursorDelete()

This deletes the designated horizontal cursor. It is not an error to delete an unknown cursor, it just has no effect.

**Func HCursorDelete({num%});**

num% The number of the cursor to delete, or -1 to delete all horizontal cursors. If this is omitted, the highest numbered cursor is deleted.

Returns The number of the deleted cursor or 0 if no cursor was deleted.

**See also:**
```
CursorDelete(), HCursor(), HCursorChan(), HCursorLabel(), HCursorLabelPos(),
HCursorNew(), HCursorRenumber()
```

# HCursorExists()

This function tests if a given horizontal cursor exists at the time of the call.

```
Func HCursorExists(num%)
```

num%   The cursor number to be tested.

Returns 1 if the cursor exists, 0 if it does not.

**See also:**
```
HCursor(), HCursorDelete(), CursorExists()
```

# HCursorLabel()

This gets and optionally sets the horizontal cursor label style for the view or for a cursor. You can label cursors with a position and/or the cursor number, or with user-defined text. There are two variants: the first sets styles and labels; the second reads back user-defined labels.

```
Func HCursorLabel({style%{, num%{, form$}}})
Func HCursorLabel(&form$, num%);
```

style%  The cursor style. Cursors can be annotated with a position or the cursor number or a user-defined style. The styles are: 0=Neither, 1=Position, 2=Number, 3=Both, 4=User-defined. Unknown styles cause no change.

num%   1-9 to select a single cursor or to return the label string. Omit num% or set it less than 1 (use -1 in case we allow a horizontal cursor 0 in the future) for all cursors and to get and set the style for new cursors.

form$   The user-defined label string for style 4. The string has replaceable parameters %p, and %n for position and number. We also allow %w.dp where w and d are numbers that set the field width and decimal places. You cannot read back a label format string.

Returns A cursor style before any change as 0-4 if a single cursor is selected, or the view cursor style as 0-3. If style% is omitted or not 0-3, the current view cursor style is not changed.

**See also:**
```
CursorLabel(), HCursor(), HCursorChan(), HCursorDelete(), HCursorLabelPos(),
HCursorNew(), HCursorRenumber()
```

# HCursorLabelPos()

This lets you set and read the position of the horizontal cursor label.

```
Func HCursorLabelPos(num%{, pos});
```

num%   The cursor number to use, from 1 to n. If the cursor does not exist the function does nothing and returns -1.

pos    If present, the command sets the position. The position is a percentage of the distance from the left of the cursor at which to position the value. Out of range values are set to the appropriate limit.

Returns The cursor position before any change was made, or -1 if the cursor does not exist.

**See also:**
```
CursorLabelPos(), HCursor(), HCursorChan(), HCursorDelete(), HCursorLabel(),
HCursorNew(), HCursorRenumber()
```

# HCursorNew()

This function creates a new horizontal cursor and assigns it to a channel. You can create up to 10 horizontal cursors, which can subsequently be moved to any channel.

```
Func HCursorNew(chan%{, where});
```

chan%   A channel number (1 to n) for the new cursor. If the channel is hidden the cursor is hidden.

where   An optional argument setting the cursor position. If this is omitted, the cursor is placed in the middle of the y axis or at zero if there is no y axis.

Returns  It returns the horizontal cursor number or 0 if all cursors are in use.

**See also:**
```
CursorNew(), HCursor(), HCursorChan(), HCursorDelete(), HCursorLabel(),
HCursorLabelPos(), HCursorRenumber()
```

# HCursorRenumber()

This command renumbers the cursors starting with channel 1. Cursors in each channel are numbered from bottom to top. There are no arguments.

```
Func HCursorRenumber();
```

Returns  The number of cursors found in the view.

**See also:**
```
CursorRenumber(), HCursor(), HCursorChan(), HCursorDelete(), HCursorLabel(),
HCursorLabelPos(), HCursorNew()
```

# HCursorValid()

Use this function to test if the last measurement for a level of an active horizontal cursor succeeded. Horizontal cursor positions are valid if a measurement succeeds or if the cursor is positioned manually or by a script command. The position of a newly created cursor is valid.

```
Func HCursorValid(num%);
```

num%   The horizontal cursor number to test for a valid measurement result.

Returns  The result is 1 if the position of the nominated horizontal cursor is valid or 0 if it is invalid or the horizontal cursor does not exist.

**See also:**
```
HCursorActive(), HCursorNew(), CursorSearch(), ChanMeasure()
```

# HCursorVisible()

Horizontal cursors can be hidden without deleting them by using this script function. This command was added at Signal version 7.05.

```
Func HCursorVisible(num%{, show%});
```

num%    The horizontal cursor number or -1 for all horizontal cursors.

show%   If present set this to 0 to hide the cursor and non-zero to show it, if not present no change is made.

Returns  The state of the horizontal cursor at the time of the call (0=hidden, 1=visible, 0 if the cursor does not exist). If num% is -1, the result is the number of horizontal cursors.

**See also:**
```
CursorVisible(), HCursor(), HCursorChan(), HCursorDelete(), HCursorLabel(),
HCursorLabelPos(), HCursorNew()
```

# HCursorX()

This function returns the position of a horizontal cursor in a Time, Result or XY view before the move that took it to the current position. It is particularly useful with Active horizontal cursors. The same mechanism is available as a dialog expression. This command was added at Signal version 6.05.

```
Func HCursorX(num%);
```

num%    The cursor to use. It is an error to attempt this operation on an unknown cursor.

Returns The function returns the position of the cursor before the move (however caused) to the current position. Use `HCursor(num%)` to get the current position.

### See also:

HCursor(), HCursorChan(), HCursorDelete(), HCursorLabel(), HCursorLabelPos(), HCursorNew(), HCursorRenumber()

# Help()

This function displays a help file page. Signal uses the standard Windows HTML help system.

```
Func Help(topic%|topic${, file$});
```

topic%  A numeric code for the help topic. These codes are assigned by the help system author. Code 0 changes the default help file to `file$`.

topic$  A string holding a help topic keyword or phrase to look-up.

file$   If this is omitted, or the string is empty, the standard Signal help file is used. If this holds a filename, this filename is used as the help file.

Returns  1 if the help topic was found, 0 if it was not, -1 if the help file was not found.

The Windows SDK has some help-authoring tools, and third-party tools are available.

# I

# IIR commands

The `IIRxxxx()` script commands make it easy for you to generate and apply IIR (Infinite Impulse Response) filters to data held in arrays of real numbers. The data values are assumed to be a sampled sequence, spaced at equal intervals. You can create digital filters that are modelled on Butterworth, Bessel, Chebyshev type 1 and Chebyshev type 2 highpass, lowpass, bandstop and bandpass filters. You can also create digital resonators and notch filters. The commands are:

| | | | | | |
|---|---|---|---|---|---|
| IIRBp() | Bandpass filter | IIRHp() | Highpass filter | IIRNotch() | Notch filter |
| IIRBs() | Bandstop filter | IIRLp() | Lowpass filter | IIRReson() | Resonator |

The algorithms used to create the filters are based on the `mkfilter` program, written by Tony Fisher of York University. The basic idea is to position the s-plane poles and zeros for a normalised low-pass filter of the desired characteristic and order, then to transform the filter to the desired type.

The theory of IIR filters is beyond the scope of this manual; a classic reference work is *Theory and Application of Digital Signal Processing* by Rabiner and Gold, published in 1975. The IIR filters generated by these commands can be modelled by:

$$y[n] = \text{Sum}(A_i * x[n-i]/G) + \text{Sum}(B_i * y[n-i])$$
$$i=0,N \qquad\qquad i=1,M$$

where the x[n] are the sequence of input data values, the y[n] are the sequence of output values, the $A_i$ and the $B_i$ are the filter coefficients (some of which may be zero) and *G* is the filter gain. Although *G* could be incorporated into the $A_i$, for computational reasons we keep it separate. In the filters designed by the `IIRxxxx()` commands, N=M and is the order of the filter for lowpass and highpass designs, twice the order for bandpass and bandstop

designs and is 2 for resonators and notch filters. The order of the low pass, high pass, band pass and band stop filters determines the sharpness of the filter cut-off: the higher the order, the sharper the cut-off.

**IIR and FIR filters**

When compared to FIR filters, IIR filters have advantages:

- They can generate much steeper edges and narrower notches for the same computational effort.
- The filters are causal, which means that the filter output is only affected by current and previous data. If you run a step change through FIR filters you typically see ringing before the step as well as after it.

However, they also have disadvantages:

- FIR filters are unconditionally stable. IIR filters are prone to stability problems if very narrow features (<0.0001 of the sample rate) are used. Problems increase at high filter orders. Filters report if they are likely to be unstable.
- They impose a group delay on the data that varies with frequency. This means that they do not preserve the shape of a waveform, in particular, the positions of peaks and troughs will change.

You can remedy the group delay problem by running a filter forwards, then backwards, through the data. However, this makes the filter non-causal, removing one of the advantages of using an IIR filter. The commands allow you to check the impulse, step, frequency and phase response of the filters, and we recommend that you do so before using a generated filter for a critical purpose.

The lowpass, highpass, bandpass and bandstop filters generate digital filters modelled on four types of analogue filter: Butterworth, Bessel, Chebyshev type 1 and Chebyshev type 2. The resulting digital filters are not identical to the analogue filters as the mapping from the analogue to the digital domain distorts the frequency scale. In many cases, this improves the performance of the digital filter over the analogue counterpart.

**Filter types**

You can generate notch and resonator filters plus lowpass, highpass, bandpass and bandstop filters modelled on Butterworth, Bessel and Chebyshev analogue filters.

**Butterworth**

These have a maximally flat pass band, but pay for this by not having the steepest possible transition between the pass band and the stop band.

**Bessel**

An analogue Bessel filter has the property that the group delay is maximally flat, which means that it tends to preserve the shape of a signal passed through it. This leads to filters with a gentle cut-off. When digitised, the constant group delay property is compromised; the higher the filter order, the worse the group delay.

**Chebyshev type 1**

Filters of this type are based on Chebyshev polynomials and have the fastest transition between the pass band and the stop band for a given ripple in the pass band and no ripples in the stop band.

**Chebyshev type 2**

Filters of this type are defined by the start of the stop band and the stop band ripple. The filter has the fastest transition between the pass and stop bands given the stop band ripple and no ripple in the pass band.

**Notch**

Notch filters are defined by a centre frequency and a q factor. q is the width of the stop band at the –3 dB point divided by the centre frequency: the higher the q, the narrower the notch. Notch filters are sometimes used to remove mains hum, but if you do this you will likely need to set notches at the first few odd harmonics of the mains frequency. If you have a fairly constant coupling of mains noise into your signal, there are other ways to remove it that may cause much less signal degradation than notch filtering.

### Resonator

A resonator is the inverse of a notch. It is defined in terms of a centre frequency and a q factor. q is the width of the pass band at the –3 dB point divided by the centre frequency: the higher the q, the narrower the resonance. Resonators are sometimes used as alternatives to a narrow bandpass filter.

### Non-filter bank commands

The `IIRBp()`, `IIRBs()`, `IIRLp()`, `IIRHp()`, `IIRReson()` and `IIRNotch()` commands are all independent and they each remember the last filter you created and the filter state after the last filtering operation. They have common operations to read back data and apply the filter to an array.

### Filter bank commands

The `IIRApply()`, `IIRComment$()`, `IIRCreate()`, `IIRInfo()` and `IIRName$()` commands use the IIR filter bank to generate and apply digital filters to channels of data.

### See also:

`FIRMake(), IIRApply(), IIRBp(), IIRBs(), IIRComment$(), IIRCreate(), IIRHp(),`
`IIRInfo(), IIRLp(), IIRName$(), IIRNotch(), IIRReson()`

# Get IIR filter information

For the `IIRBp()`, `IIRBs()`, `IIRLp()`, `IIRHp()`, `IIRReson()` and `IIRNotch()` commands, you use a variant of the following form to return filter information:

```
Func IIRxxxx(get%{, arr[]{[]}});
```

get%   The form of the command using this argument is used to read back information about the last filter created with this command. The argument can be:

    1  `arr[]` is set to the impulse response of the filter. This is the response to an input of 1 followed by an infinite number of zeros assuming that all previous inputs were also zero. It is up to you to set a suitable length of the array. Each array data points corresponds with one sample interval. The return value is the magnitude of the largest value in `arr[]`. For example, if the impulse response ranged in values from –0.5 to 0.3, 0.5 would be returned.

    2  `arr[]` is set to the step response of the filter. The input to the filter is assumed to be an infinite number of zeros, followed by an infinite number of ones, and the response is taken from the moment where the input changes to a 1. It is up to you to set a suitable array length. The return value is the magnitude of the largest value in `arr[]`.

    3  `arr[][]` is a matrix with r rows and 2 columns. The frequency response is returned as complex numbers in the columns; `arr[][0]` holds the real part and `arr[][1]` the imaginary part. The first row corresponds to a frequency of 0; the final row corresponds to a frequency of half the sampling rate. The frequencies are spaced as `0.5/(r-1)`. The more rows you set, the finer the frequency response. The return value is the maximum magnitude of the returned frequency response.

    4  The same as 3 except that the results are returned as the amplitude response in column 0 and the phase response in column 1. If the real and imaginary parts of the response are r and i, `arr[][0]` holds `sqrt(r*r+i*i)` and `arr[][1]` holds `atan(i, r)`. The return value is the maximum returned amplitude response.

    5  Returns the number of filter coefficients ($N+1$) to apply to the filter input values and fills in `arr[]` with these values. These correspond to the $a_i$ in the filter expression. However, we return the values in reverse order as this makes them easier to use as a dot product with old values. $N$ is the filter order for low pass and high pass, twice the order for band pass and band stop, and 2 for resonators and notch filters.

    6  Returns the number of filter coefficients ($N+1$) to apply to the filter output values and fills in `arr[]` with these values. These correspond to the $b_i$ in the filter expression. However, we return the values in reverse order to match the $a_i$. The final value is always –1.0 (corresponding to $b_0$, which is not used when implementing the filter). $N$ is the filter order for low pass and high pass, twice the order for band pass and band stop, and 2 for resonators and notch filters.

    7  Returns the filter gain $G$ as defined in the filter expression.

    8  `arr[][]` is a matrix with $N$ rows and 2 columns. $N$ is the filter order for low pass and high pass, twice the order for band pass and band stop, and 2 for resonators and notch filters. The return value is the

number of poles in the s-plane and the matrix is filled in with the poles as complex numbers with the real part in column 0 and the imaginary part in column 1.

9   The same as 8, but returning the s-plane zeros.

10  The same as 8, but returning the z-plane poles.

11  The same as 8, but returning the z-plane zeros.

12  Returns a measure of filter stability, being the distance of the nearest pole to the unit circle. It is our experience that values greater than 1e-12 generate plausible filters. As the filter depends on a function of this distance, which is of the form (pole position-1), as the pole position approaches 1, the numerical accuracy of the result become significantly compromised (floating point numbers have around 15 significant digits of accuracy, all else being equal). You can improve the stability by reducing the order of the filter. Getting the stability measure was added at Signal version 5.05.

To read back information about a filter in the filter bank, use the `IIRInfo()` command.

**See also:**
`FIRMake(), IIRApply(), IIRBp(), IIRBs(), IIRHp(), IIRInfo(), IIRLp(), IIRNotch(), IIRReson()`

## Apply IIR filter to an array

This command applies the current filter set by one of the `IIRBp(), IIRBs(), IIRLp(), IIRHp(), IIRReson()` and `IIRNotch()` commands to an array of equally spaced data

```
Func IIRBp(data[]{, flags%{, save[]}});
Func IIRBs(data[]{, flags%{, save[]}});
Func IIRLp(data[]{, flags%{, save[]}});
Func IIRHp(data[]{, flags%{, save[]}});
Func IIRReson(data[]{, flags%{, save[]}});
Func IIRNotch(data[]{, flags%{, save[]}});
```

data    An array of data to filter.

flags%  Optional, taken as 0 if omitted. Add:

1   To apply the filter backwards through the array. Applying a filter introduces a phase shift; running a filter forwards then backwards cancels the phase shift at the expense of a non-causal filter.

2   To treat `data[]` as a continuation of the last filtering operation (only do this if it really is a continuation, otherwise the results are nonsense). The filter state is saved separately for each filter command, but if you want to interleave use of the the same command between multiple data streams, you must use the `save` argument. If you apply the filter backwards, you must present the data blocks backwards.

save[]  Optional. This is a real array that preserves the state of the IIR filter so that you can interleave continuous filtering using the same filter between multiple data streams, for example different channels. You will need a separate `save[]` array for each data stream/channel, the IIR filtering command will restore its state from the array before filtering if the 2 value in `flags%` is set and will save the filter state in the array after finishing the filtering - thus preserving the internal state separately for each data stream. The minimum size of the array depends on the filter type and order. It is 2*order+2 for `IIRLp()` and `IIRHp()`, 6 for `IIRReson()` and `IIRNotch()` and 4*order+2 for `IIRBp()` and `IIRBs()`.

If `save` is present and 2 is added to `flags%`, the filter state is loaded from it before filtering; it is always updated after filtering. You can simplify code by always having the continuous data flag set if you make sure that `save[]` is initialised to zeros before it is used for the first time.

**See also:**
`FIRMake(), IIRApply(), IIRBp(), IIRBs(), IIRHp(), IIRLp(), IIRNotch(), IIRReson()`

## IIRApply()

Applies a filter in the IIR filter bank to a waveform channel in the current data view.

```
Func IIRApply(index%, cSpc, frm%|frm%[]|frm$);
```

index%  Index of the filter in the filter bank to apply in the range -1 to 11

cSpc    A channel specifier for the channels to filter.

frm%     Frame number or a negative code as follows:

-1   All frames in the file

-2   The current frame

-3   Only tagged frames

-6   Only untagged frames

frm$     A frame specification string. This option specifies a list of frames using a string such as "1..32,40,50".

frm%[]   An array of frame numbers. This option provides a list of frame numbers. The first element holds the number of frames in the list.

Returns  Zero or a negative error code. A negative error code is also returned if the user clicks Cancel from the progress bar that may appear during a long filtering operation.

**See also:**

The filter bank, `IIRComment$()`, `IIRCreate()`, `IIRBp()`, `IIRInfo()`, `IIRName$()`

# IIRBp()

This function creates and applies IIR (Infinite Impulse Response) band pass filters to arrays of data. You can run the filter forwards or backwards through the data.

```
Func IIRBp(data[]|0, lo, hi, order%{, type%{, ripple}});
Func IIRBp(data[]{, flags%{, save[]}});
Func IIRBp(get%{, arr[]{[]}});
```

data     An array of data to filter. If there are only 1 or 2 arguments, the last created band pass filter is used. Otherwise, the filter defined by the remaining arguments is used forwards. Replace `data` with 0 to create a filter without applying it.

flags%   Optional, taken as 0 if omitted. Add 1 to run the filter backwards, 2 to treat `data[]` as a continuation of the last filtering operation.

save[]   Optional. If present it must have a size of at least `4*order%+2`. It is used to save the filter state to allow use with the continued data flag with multiple data streams.

lo       The low corner frequency of the band stop filter. This is expressed as a fraction of the sample rate and is limited to the range 0.000001 to 0.499998. For Chebyshev type 2 filters, this is the point at which the attenuation reaches the `ripple` value, for all other filters this sets the –3 dB point.

hi       The high corner frequency of the band pass filter. This is expressed as a fraction of the sampling rate and is limited to the range `lo`+0.000001 to 0.499999. For Chebyshev type 2 filters, this is the point at which the attenuation reaches the `ripple` value, for all other filters this sets the –3 dB point.

order%   The order of the lowpass filter used as the basis of the design, in the range 1 to 10. The order of the filter implemented is `order%*2`. High orders (`order%` > 7) and narrow bands may cause inaccuracy in the filter. Narrow means that `(hi-lo)/sqrt(lo*hi)` is less than 0.2, for example.

type%    Set 0 for Butterworth, 1 for Bessel and 2 for Chebyshev type 1, 3 for Chebyshev type 2. The default value is 0 for a Butterworth filter.

ripple   The desired pass band ripple in dB for Chebyshev type 1 filters (default 3 dB) or the desired minimum cut in the stop bands for Chebyshev type 2 filters (default 40 dB). The ripple value must be in the range 0.01 to 1000 dB.

get%     The command variant with this argument returns information about the last filter you created with this command. See the discussion of IIR commands for details.

arr      An optional vector or matrix used to return information about the last filter you created with this command. See the discussion of IIR commands for details.

Returns  All forms of the command return negative numbers for errors. The forms that apply or create filters return 0 for success or 1 if a created filter has stability less than 1e-12. The other command forms have their return values included in the description of the `get%` argument or return 0.

**See also:**

More about IIR filters and commands, Get filter information, `FIRMake()`, `IIRBs()`, `IIRHp()`, `IIRLp()`, `IIRNotch()`, `IIRReson()`

# IIRBs()

This function creates and applies IIR (Infinite Impulse Response) band stop filters to arrays of data. You can run the filter forwards or backwards through the data.

```
Func IIRBs(data[]|0, lo, hi, order%{, type%{, ripple}});
Func IIRBs(data[]{, flags%{, save[]}});
Func IIRBs(get%{, arr[]{[]}});
```

data    An array of data to filter. If there are only 1 or 2 arguments, the last created band stop filter is used. Otherwise, the filter defined by the remaining arguments is applied forwards. Replace `data` with 0 to create a filter without applying it.

flags%   Optional, taken as 0 if omitted. Add 1 to run the filter backwards, 2 to treat `data[]` as a continuation of the last filtering operation.

save[]   Optional. If present it must have a size of at least `4*order%+2`. It is used to save the filter state to allow use with the continued data flag with multiple data streams.

lo    The low corner frequency of the band pass filter. This is expressed as a fraction of the sample rate and is limited to the range 0.000001 to 0.499998. For Chebyshev filters, this is the point at which the attenuation reaches the `ripple` value, for other filters this sets the –3 dB point.

hi    The high corner frequency of the band pass filter. This is expressed as a fraction of the sampling rate and is limited to the range `lo+0.000001` to 0.499999. For Chebyshev type 2 filters, this is the point at which the attenuation reaches the `ripple` value, for all other filters this sets the –3 dB point.

order%   The order of the lowpass filter used as the basis of the design, in the range 1 to 10. The order of the filter implemented is `order%*2`. High orders and narrow pass bands may lose numerical accuracy in the filter output.

type%    Set 0 for Butterworth, 1 for Bessel and 2 for Chebyshev type 1, 3 for Chebyshev type 2. The default value is 0 for a Butterworth filter.

ripple   The desired pass band ripple in dB for Chebyshev type 1 filters (default 3 dB) or the desired minimum cut in the stop band for Chebyshev type 2 filters (default 40 dB). The ripple value must be in the range 0.01 to 1000 dB.

get%    The command variant with this argument returns information about the last filter you created with this command. See the discussion of IIR commands for details.

arr    An optional vector or matrix used to return information about the last filter you created with this command. See the discussion of IIR commands for details.

Returns   All forms of the command return negative numbers for errors. The forms that apply or create filters return 0 for success or 1 if a created filter has stability less than 1e-12. The other command forms have their return values included in the description of the `get%` argument or return 0.

**See also:**
More about IIR filters and commands, Get filter information, `FIRMake()`, `IIRBp()`, `IIRHp()`, `IIRLp()`, `IIRNotch()`, `IIRReson()`

# IIRComment$()

This function gets and sets the comment associated with an IIR filter in the filter bank.

```
Func IIRComment$(index%{, new$});
```

index%   Index of the filter in the filter bank to use in the range -1 to 11.

new$    If present, sets the new comment.

Returns   The previous comment for the filter at the index.

**See also:**

Filter banks, `IIRApply()`, `IIRInfo()`, `IIRName$()`

# IIRCreate()

This creates an IIR filter description and adds it to the filter bank.

**`Func IIRCreate(index%, type%, model%, order%, fr1{, fr2{, extra}});`**

`index%` Index of the filter in the filter bank in the range -1 to 11.

`type%` Sets the filter type as: 0=Low pass, 1=High pass, 2=Band pass, 3=Band stop.

`model%` Sets the filter model: 0=Butterworth, 1=Bessel, 2=Chebyshev type 1, 3=Chebyshev type 2, 4=Resonator.

`order%` Sets the filter order in the range 1-10. Resonators always set an order of 2.

`fr1` Sets the corner frequency for low pass, high pass filters, the centre frequency for resonators, and the low corner frequency for band pass and band stop filters.

`fr2` Sets the upper corner frequency for band pass/stop filters, otherwise ignored.

`extra` Sets the ripple for Chebyshev filters in the range 0.01 to 1000 and the Q factor for resonators in the range 1 to 10000.

Returns 0 if OK or a negative error code if the operation failed.

**See also:**
The filter bank, `IIRApply()`, `IIRComment$()`, `IIRCreate()`, `IIRName$()`

# IIRHp()

This function creates and applies IIR (Infinite Impulse Response) high pass filters to arrays of data. You can run the filter forwards or backwards through the data.

**`Func IIRHp(data[]|0, edge, order%{, type%{, ripple}});`**
**`Func IIRHp(data[]{, flags%{, save[]}});`**
**`Func IIRHp(get%{, arr[]{[]}});`**

`data` An array of data to filter. If there are only 1 or 2 arguments, the last created high pass filter is used. Otherwise, the filter defined by the remaining arguments is applied forwards. Replace `data` with 0 to create a filter without applying it.

`flags%` Optional, taken as 0 if omitted. Add 1 to run the filter backwards, 2 to treat `data[]` as a continuation of the last filtering operation.

`save[]` Optional. If present it must have a size of at least `2*order%+2`. It is used to save the filter state to allow use with the continued data flag with multiple data streams.

`edge` The corner frequency of the high pass filter. This is expressed as a fraction of the sample rate and is limited to the range 0.000001 to 0.499999. For Chebyshev filters, this is the point at which the attenuation reaches the `ripple` value, for other filters this sets the –3 dB point.

`order%` The order of the filter in the range 1 to 10.

`type%` Set 0 for Butterworth, 1 for Bessel and 2 for Chebyshev type 1, 3 for Chebyshev type 2. The default value is 0 for a Butterworth filter.

`ripple` The desired pass band ripple in dB for Chebyshev type 1 filters (default 3 dB) or the desired minimum cut in the stop bands for Chebyshev type 2 filters (default 40 dB). The ripple value must be in the range 0.01 to 1000 dB.

`get%` The command variant with this argument returns information about the last filter you created with this command. See the discussion of IIR commands for details.

`arr` An optional vector or matrix used to return information about the last filter you created with this command. See the discussion of IIR commands for details.

Returns  All forms of the command return negative numbers for errors. The forms that apply or create filters return 0 for success or 1 if a created filter has stability less than 1e-12. The other command forms have their return values included in the description of the `get%` argument or return 0.

**See also:**
More about IIR filters and commands, Get filter information, `FIRMake()`, `IIRBp()`, `IIRBs()`, `IIRLp()`, `IIRNotch()`, `IIRReson()`

# IIRInfo()

Retrieves information about an IIR filter in the bank.

```
Func IIRInfo(index%, &model%, &order%, &fr1{, &fr2{, &extra}});
```

`index%`  Index of the filter in the filter bank in the range -1 to 11.

`model%`  Returned as the filter model: 0=Butterworth, 1=Bessel, 2=Chebyshev type 1, 3=Chebyshev type 2, 4=Resonator.

`order%`  Returned as the filter order in the range 1-10. Resonators always return 2.

`fr1`  Returned as the corner frequency for low and high pass filters, as the low corner for band pass and band stop filters and as the centre frequency for resonators.

`fr2`  Returned as the upper corner frequency for band pass and band stop filters, otherwise set the same as `fr1`.

`extra`  Returned as the ripple for Chebyshev filters and as the Q factor for resonators.

Returns  The type of the filter as 0=Low pass, 1=High pass, 2=Band pass, 3=Band stop.

**See also:**
The filter bank, `IIRApply()`, `IIRComment$()`, `IIRCreate()`, `IIRName$()`

# IIRLp()

This function creates and applies IIR (Infinite Impulse Response) low pass filters to arrays of data. You can run the filter forwards or backwards through the data.

```
Func IIRLp(data[]|0, edge, order%{, type%{, ripple}});
Func IIRLp(data[]{, flags%{, save[]}});
Func IIRLp(get%{, arr[]{[]}});
```

`data`  An array of data to filter. If there are only 1 or 2 arguments, the last created low pass filter is used. Otherwise, the filter defined by the remaining arguments is applied forwards. Replace `data` with 0 to create a filter without applying it.

`flags%`  Optional, taken as 0 if omitted. Add 1 to run the filter backwards, 2 to treat `data[]` as a continuation of the last filtering operation.

`save[]`  Optional. If present it must have a size of at least `2*order%+2`. It is used to save the filter state to allow use with the continued data flag with multiple data streams.

`edge`  The corner frequency of the low pass filter. This is expressed as a fraction of the sample rate and is limited to the range 0.000001 to 0.499999. For Chebyshev filters, this is the point at which the attenuation reaches the `ripple` value, for other filters this sets the –3 dB point.

`order%`  The order of the filter in the range 1 to 10.

`type%`  Set 0 for Butterworth, 1 for Bessel and 2 for Chebyshev type 1, 3 for Chebyshev type 2. The default value is 0 for a Butterworth filter.

`ripple`  The desired pass band ripple in dB for Chebyshev type 1 filters (default 3 dB) or the desired minimum cut in the stop bands for Chebyshev type 2 filters (default 40 dB). The ripple value must be in the range 0.01 to 1000 dB.

`get%`  The command variant with this argument returns information about the last filter you created with this command. See the discussion of IIR commands for details.

arr    An optional vector or matrix used to return information about the last filter you created with this command. See the discussion of IIR commands for details.

Returns  All forms of the command return negative numbers for errors. The forms that apply or create filters return 0 for success or 1 if a created filter has stability less than 1e-12. The other command forms have their return values included in the description of the `get%` argument or return 0.

### See also:

More about IIR filters and commands, Get filter information, `FIRMake()`, `IIRBp()`, `IIRBs()`, `IIRHp()`, `IIRNotch()`, `IIRReson()`

## IIRName$()

This function gets and/or sets the name of an IIR filter in the filter bank.

```
Func IIRName$(index%{, new$});
```

index% Index of the filter in the filter bank to use in the range -1 to 11.

new$    If present, sets the new name.

Returns  The previous name of the filter at that index.

### See also:

Filter banks, `IIRApply()`, `IIRComment$()`, `IIRInfo()`

## IIRNotch()

This function creates and applies IIR (Infinite Impulse Response) notch filters to arrays of data. You can run the filter forwards or backwards through the data. The gain of the notch filter is zero at the notch frequency and 1 at low and high frequencies.

```
Func IIRNotch(data[]|0, fr, q);
Func IIRNotch(data[]{, flags%{, save[]}});
Func IIRNotch(get%{, arr[]{[]}});
```

data    An array of data to filter. If there are only 1 or 2 arguments, the last created notch filter is used. Otherwise, the filter defined by the remaining arguments is applied forwards. Replace `data` with 0 to create a filter without applying it.

flags% Optional, taken as 0 if omitted. Add 1 to run the filter backwards, 2 to treat `data[]` as a continuation of the last filtering operation.

save[] Optional. If present it must have a size of at least 6. It is used to save the filter state to allow use with the continued data flag with multiple data streams.

fr      The frequency of the notch. This is expressed as a fraction of the sample rate and is limited to the range 0.000001 to 0.499999.

q      The `q` factor for the notch in the range 1 to 10000; the higher the q, the narrower the notch. If `Flo` and `Fhi` are the frequencies of the –3 dB points either side of the notch, q is `fr/(Fhi-Flo)`. Try 100 as a starting point.

get%   The command variant with this argument returns information about the last filter you created with this command. See the discussion of IIR commands for details.

arr    An optional vector or matrix used to return information about the last filter you created with this command. See the discussion of IIR commands for details.

Returns  All forms of the command return negative numbers for errors. The forms that apply or create filters return 0 for success or 1 if a created filter has stability less than 1e-12. The other command forms have their return values included in the description of the `get%` argument or return 0.

### See also:

More about IIR filters and commands, Get filter information, `FIRMake()`, `IIRBp()`, `IIRBs()`, `IIRHp()`, `IIRLp()`, `IIRReson()`

# IIRReson()

This function creates and applies IIR (Infinite Impulse Response) resonator filters to arrays of data. You can run the filter forwards or backwards through the data. The gain of the filter is 1 at the resonator frequency and zero at low and high frequencies.

```
Func IIRReson(data[]|0, fr, q);
Func IIRReson(data[]{, flags%{, save[]}});
Func IIRReson(get%{, arr[]{[]}});
```

data     An array of data to filter. If there are only 1 or 2 arguments, the last created resonator filter is used. Otherwise, the filter defined by the remaining arguments is applied forwards. Replace `data` with 0 to create a filter without applying it.

flags%   Optional, taken as 0 if omitted. Add 1 to run the filter backwards, 2 to treat `data[]` as a continuation of the last filtering operation.

save[]   Optional. If present it must have a size of at least 6. It is used to save the filter state to allow use with the continued data flag with multiple data streams.

fr       The centre frequency of the resonator. This is expressed as a fraction of the sample rate and is limited to the range 0.000001 to 0.499999.

q        The `q` factor for the resonator in the range 1 to 10000; the higher the q, the narrower the resonance. If `Flo` and `Fhi` are the frequencies of the –3 dB points either side of the resonance, `q` is `fr/(Fhi-Flo)`. Try 100 as a starting point.

get%     The command variant with this argument returns information about the last filter you created with this command. See the discussion of IIR commands for details.

arr      An optional vector or matrix used to return information about the last filter you created with this command. See the discussion of IIR commands for details.

Returns  All forms of the command return negative numbers for errors. The forms that apply or create filters return 0 for success or 1 if a created filter has stability less than 1e-12. The other command forms have their return values included in the description of the `get%` argument or return 0.

**See also:**
More about IIR filters and commands, Get filter information, `FIRMake()`, `IIRBp()`, `IIRBs()`, `IIRHp()`, `IIRLp()`, `IIRNotch()`

# Info...()

# InfoCloseAll()

This function closes all the information windows attached to the current view.

```
Proc InfoCloseAll();
```

**See also:**
`InfoNew()`, `InfoSettings()`, `InfoText$()`, `InfoImage()`, `InfoReset()`, `InfoStart()`, `InfoStop()`, `FileClose()`

# InfoImage()

This function sets and retrieves information about the image that is displayed in the current information window.

```
Func InfoImage(path$);
Func InfoImage(hide%, opa{, left{, top{, width{, height}}}});
```

These forms of the command set the image file to be displayed or other image parameters.

path$    A string holding the complete path and filename of the image file to be displayed in the information window.

hide%    This sets the new image hidden state: 0 for visible, 1 for hidden, -1 for no change.

opa       This sets the image opacity from 0 for fully transparent to 1 for completely opaque, or negative for unchanged.

left      This sets the X position of the top left corner of the image area as a percentage of the info window width, from 0 to 100. A value of zero is used if this is omitted.

top       This sets the Y position of the top left corner of the image area as a percentage of the info window height, from 0 to 100. A value of zero is used if this is omitted.

width     This sets the width of the image area as a percentage of the info window width, from 0 to 100. A value of 100 is used if this is omitted.

height    This sets the height of the image area as a percentage of the info window height, from 0 to 100. A value of 100 is used if this is omitted.

Returns   The function result is not used and will be zero.

### Func InfoImage(opt%{, path$});

This form of the command returns information about the displayed image.

opt%      A value controlling the information that is returned:

| opt% | Function result |
|------|-----------------|
| -1 | The image hidden state: 1 if the image is hidden, 0 if it is shown |
| -2 | The image opacity from 0 for fully transparent to 1 for completely opaque |
| -3 | The X position of the top left corner of the image area |
| -4 | The Y position of the top left corner of the image area |
| -5 | The width of the image area |
| -6 | The height of the image area |

path$     A string variable that will be updated with the complete path and filename of the image file that is displayed.

Returns   The function result is the image information value as selected by flag%.

**See also:**
InfoNew(), InfoSettings(), InfoText$(), InfoCloseAll(), InfoReset(), InfoStart(), InfoStop()

# InfoNew()

This function creates a new information window with the current data, memory or XY view as its parent.

### Func InfoNew({text${, flags%}});

text$     This sets the text that will be displayed by the new information window, with any special fields appropriately interpreted.

flags%    This parameter controls various information window options, it is the sum of the following values (taken as zero if omitted):

| | |
|---|---|
| 1 | Add this value for a timer that counts down, otherwise it counts upwards |
| 2 | Add this value for a timer that stops when the stop value is reached, otherwise stop is ignored |
| 4 | Add this value for a timer which automatically starts when it is reset |
| 8 | Add this value to hide the buttons at the top of the information window |
| 16 | Add this value to lock the text font size. |
| 32 | Add this value to hide the window title bar. |

Returns   The function result is the view handle of the new information window or a negative error code.

**See also:**
InfoImage(), InfoSettings(), InfoText$(), InfoCloseAll(), InfoReset(), InfoStart(), InfoStop()

# InfoReset()

This function resets the timer in the current information window or all associated information windows.

```
Func InfoReset({which%});
```

which% A value controlling which timers are reset. If zero or omitted then the current view must be the information window that is reset. If which% is -1 then all information windows attached to the current data or XY view (or all information windows attached to the data or XY view which is the owner of the current information window) are reset.

Returns  The function result is not used and will be zero.

**See also:**
```
InfoNew(), InfoSettings(), InfoText$(), InfoCloseAll(), InfoImage(), InfoStart(),
InfoStop()
```

# InfoSettings()

This function changes the settings of the current information window, and also supports a special mode that reads the individual settings back from the window.

```
Func InfoSettings(text${, flags%{, reset{, stop{, step%{, show}}}}});
```

text$  This sets the text that will be displayed by the new information window, with any special fields appropriately interpreted.

flags% This parameter controls various information window options, it is the sum of the following values (taken as zero if omitted):

   1   Add this value for a timer that counts down, otherwise it counts upwards
   2   Add this value for a timer that stops when the stop value is reached, otherwise stop is ignored
   4   Add this value for a timer which automatically starts when it is reset
   8   Add this value to hide the buttons at the top of the information window
  16   Add this value to lock the text font size.
  32   Add this value to hide the window title bar.

reset  This sets the time in seconds to which the timer is set when it is reset.

stop   This sets the time in seconds at which the timer will automatically stop if the flags% value has enabled this option. For timers configured to count upwards this value (if used) must be greater than reset, for timers counting down it must be less than reset.

step%  This value selects how often the displayed time is updated and the smallest unit of time that is shown; 0 for 0.1 seconds, 1 for 1 second, 2 for 1 minute and 3 for 1 hour.

show%  This value selects how the time is shown; 0 for a count of 100 millisecond intervals, 1 for a count of seconds, 2 for MM:SS and 3 for HH:MM:SS.

```
Func InfoSettings(opt%);
```

opt%   To read back the settings, opt% must be a negative value that selects what value is returned.

Returns  The function result is a setting value as selected by opt%:

   opt%  Function result
   -1   The reset time, in seconds
   -2   1 if the timer is counting down, 0 if it counts upwards
   -3   1 if the timer will stop when it reaches the stop time, 0 if not
   -4   1 if the timer is automatically started on a reset, 0 if not
   -5   1 if the info window buttons are hidden, 0 if not
   -6   The time at which the timer stops (if enabled), in seconds

-7    How often the displayed time is updated and the smallest unit of time that is shown; 0 for 0.1 seconds, 1 for 1 second, 2 for 1 minute and 3 for 1 hour

-8    How the time is shown; 0 for a count of 100 millisecond intervals, 1 for a count of seconds, 2 for MM:SS and 3 for HH:MM:SS

-9    1 if the window timer is running, 0 if it is stopped

**See also:**
```
InfoImage(), InfoNew(), InfoText$(), InfoCloseAll(), InfoReset(), InfoStart(),
InfoStop()
```

# InfoStart()

This function starts the timer in the current information window or all associated information windows. You can read back the current running/stopped state without changing it using `InfoSettings(-9)`.

```
Func InfoStart({which%});
```

`which%` A value controlling which information windows are affected. If zero or omitted then the current view must be an information window whose timer is started. If `which%` is -1 then the timers in all information windows attached to the current data or XY view (or all information windows attached to the data or XY view which is the owner of the current information window) are started.

Returns  The function result when `which%` is zero or omitted is the timer run state at the time of the function call; 1 if the timer was running and zero if not. If `which%` is -1 then the function result is 0.

**See also:**
```
InfoNew(), InfoSettings(), InfoText$(), InfoCloseAll(), InfoImage(), InfoStop(),
InfoReset()
```

# InfoStop()

This function stops the timer in the current information window or all associated information windows. You can read back the current running/stopped state without changing it using `InfoSettings(-9)`.

```
Func InfoStop({which%});
```

`which%` A value controlling which information windows are affected. If zero or omitted then the current view must be an information window whose timer is stopped. If `which%` is -1 then the timers in all information windows attached to the current data or XY view (or all information windows attached to the data or XY view which is the owner of the current information window) are stopped.

Returns  The function result when `which%` is zero or omitted is the timer run state at the time of the function call; 1 if the timer was running and zero if not. If `which%` is -1 then the function result is 0.

**See also:**
```
InfoNew(), InfoSettings(), InfoText$(), InfoCloseAll(), InfoImage(), InfoStart(),
InfoReset()
```

# InfoText$()

This function returns the text used by the current information window.

```
Func InfoText$();
```

Returns  The information window text which is used at the time of the function call.

**See also:**
```
InfoNew(), InfoSettings(), InfoStart(), InfoStop(), InfoCloseAll(), InfoImage(),
InfoReset()
```

# Input()

This function reads a number from the user. It opens a window with a message, and displays the initial value of a variable. You can limit the range of the result.

```
Func Input(text$, val {,low {, high{, pre%}}});
```

text$   A string holding a prompt for the user. If the string includes a vertical bar, the text before the vertical bar is used as the window title.

val     The initial value to be displayed for editing. If limits are given, and the initial value is outside the limits, it is set to the nearer limit.

low     An optional low limit for the result. If low>=high, the limits are ignored.

high    An optional high limit for the result.

pre%    If present this sets the number of significant figures to use to represent the number, in the range 6 (the default) to 15.

Returns The value typed in. The function always returns a value. If an out-of-range value is entered, the function warns the user and a correct value must be given. When parsing the input, leading white space is ignored and the number interpretation stops at the first non-numeric character or the end of the string.

**See also:**
DlgReal(), DlgInteger(), Input$()

# 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()

# 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(), InStrRE()`

# InStrRE()

This function searched for a string within another string using regular expressions. This code is implemented though the regular expression engine that is part of the C++ standard. There are a lot of flag values and error codes that are defined by name in the standard, but not by value. Because of this, we have given them all symbolic names, and the associated values are defined in the script file `regex.sgs` in the system include folder. Should you need to refer to any of these symbolic names, `#include <regex.sgs>` at the start of your script. There are 3 command variants:

## Set the regular expression
Call this function first to set up for matching. The regular expression and syntax you set here persist until you change it.

`Func InStrRE(sytax%, re$);`

`syntx%` Sets the syntax to use when parsing the regular expression (both the grammar and options). You should only set one of the flags that set the regular expression grammar, or set none for ECMAScript. The call with two arguments sets the default values to use if `match%` and `syntx%` are not supplied in the other command variants.

| | |
|---|---|
| `resECMA Script` | Use the script grammar as described for ECMAScript. This is the default and is set if you do not specify a grammar. |
| `resBasi c` | Use the basic POSIX script grammar. |
| `resExte nded` | Use the extended POSIX script grammar. |
| `resAwk` | Use the awk grammar. |
| `resGrep` | Use the POSIX grep grammar. |
| `resEgre p` | Use the POSIX egrep grammar. |
| `resIcas e` | Ignore case when matching. |
| `resNosu bs` | Do not return information about any sub-matches. |
| `resOpti mise` | Spend more time optimising the search. Maybe useful when searching a long string or if you will apply the same regular expression to multiple strings. This is not guaranteed to have any effect. |
| `resColl ate` | Make character ranges [a-d] sensitive to the locale. This can be important if your search text is not in English. |

`re$`  The regular expression text to use for the search.

Returns  0 if the regular expression was compiled without error or a negative error code (see below).

## Search text for a match
`Func InStrRE(text${, index%{, match%{, ms%}}});`

`text$`  The text to search. You can move the start point with index%.

`index%` If present, the start character index into `text$` for the search. The first character is index 1. That is, characters before this index are not searched (or even visible to the search engine).

`match%` A set of flags that can be used to control the matching process. Although these flags ought to have stable values, we cannot guarantee that they will not be changed, so you must use the constants defined in `regex.sgs`.

| `remDefault` | The default state. You should use this unless you really know you need something else. |
|---|---|
| `remNotBol` | Do not take the first character as matching beginning of line (`^`). |
| `remNotEol` | Do not take the end of the string as matching end of line (`$`). |
| `remNotBow` | Do not take the start of the string as matching beginning of word (`\b`). |
| `remNotEow` | Do not take the end of the string as matching the end of a word (`\b`). |
| `remAny` | If more than 1 match is possible, then any match is acceptable. |
| `remNotNull` | If `text$` is empty, then no match is possible (prevents "`x*`" matching an empty string, for example). |
| `remContinuous` | The search expression must match with the first character to succeed. |

ms%  An optional time out in milliseconds. If you omit this it defaults to 1000 milliseconds. If you are certain that your regular expression is well-formed and will terminate quickly you can set this to 0, which removes the time out and the search will run faster. Beware: it is possible to write expressions that will search forever. If you set `ms%` to 0 and set such a search, you will not be able to stop it and Signal will hang.

Returns 0 if no match was found. If a match is found, the return value is 1 plus the number of capture groups in the regular expression. The return code is negative for an error (see below). If you are timed out, the return code is `rerrComplexity`.

### Get information about the matches and captured groups
You can get the matched text as positions and sizes in the original string. Positions and sizes are very useful if you want to replace the found text.

**Func InStrRE(pos%[]{, sz%[]});**

pos%[]  This array is filled in with the start indices into text$ of the match, followed by the start indices of any captured groups. If a capture group is part of an alternative and is not matched, for example "(dog)|(cat)", the non-matching capture group has a start index of 0 and a size of 0. It is not an error for the array to be too small or too large.

sz%[]  Optional. If there is a match, this is filled in with the size of the complete match, followed by the sizes of each captured group. It is not an error for the array to be too small or too large.

got$[]  An array of strings to be filled in with the matched text, followed by the captured groups.

Returns The number of available matches.

### All in one call
This call sets up a search, performs it and reports the start index of any match in a single call. You can follow it with the previous variant to get more detailed match information.

**Func InStrRE(text$, re${, index%{, match%{, syntax%{, ms%}}}});**

text$  The text to search. You can move the start point with index%.

index%  If present, the start character index into text$ for the search.

match%  An optional set of flags that can be used to control the matching process as described above.

syntx%  Sets the syntax to use when parsing the regular expression (both the grammar and options) as described above. If omitted, the default syntax is set.

ms%  An optional time out in milliseconds as described above.

Returns The position of the start of the match or 0 if no match or a negative error code. If you need the positions and sizes of matches you can follow this with the `InStrRE(pos%[]{, sz%[]})` call.

### Error codes

All of the function variants return a positive code for normal operation and a negative code for an error. The `regex.sgs` file holds definitions of the error code constants described below and also `Func regexError$(err %)` that converts a negative error code into a text string. Currently defined error codes are:

| | |
|---|---|
| `rerrCollate` | Use of an invalid collating element name. |
| `rerrCtype` | Use of invalid character class name :`lower`: is an example of a valid name. |
| `rerrEscape` | Use of an invalid escaped character, or a trailing escape. |
| `rerrBackref` | Invalid back reference, for example \3 when no such parenthesized group. |
| `rerrBrack` | Mismatched square brackets [ and ]. |
| `rerrParen` | Mismatched parentheses ( and ). |
| `rerrBrace` | Mismatched braces { and }. |
| `rerrBadbrace` | The contents of the braces { and } was invalid. |
| `rerrRange` | There was an invalid character range. |
| `rerrSpace` | Ran out of memory when converting the regular expression to a state machine. |
| `rerrBadrepeat` | One of the repeat specifiers (+*? or {...}) did not have a valid expression in front of it. |
| `rerrComplexity` | The regular expression is too complicated or the search was timed out. |
| `rerrStack` | Ran out of space when matching the expression. |
| `rerrParse` | Some other error when parsing the regular expression. |
| `rerrSyntax` | A syntax error when parsing that is not covered above. |

### Example

The following very simple example locates a numeric data with the format `dd/mm/yy` in text. We allow the day and month to be specified by 1 or 2 decimal digits and the year by 4 decimal digits. We capture the day, month and year as separate strings:

```
var text$ := "There is a date 1/12/14 in here";
var find$ := "(\\d{1,2})/(\\d{1,2})/(\\d{2,2})";
InStrRE(0, find$);            'Setup the search
var n% := InStrRE(text$);  'Search for a match
PrintLog("%d\n", n%);        'Print the matches
var pos%[4],sz%[4], i%;
InStrRE(pos%, sz%);          'get positions and sizes
for i% := 0 to n%-1 do
    PrintLog("%s\n", Mid$(text$, pos%[i%], sz%[i%]))
next;
```

Note that you have to double up the \ characters when they are supplied as part of a literal string (as they are used as escapes). The output from this script is:

```
4
1/12/14
1
12
14
```

### See also:

`Chr$()`, `DelStr$()`, `LCase$()`, `Left$()`, `Len()`, `Mid$()`, `Print$()`, `ReadStr()`, `Right$()`, `Str$()`, `UCase$()`, `Val()`, `InStr()`

# Grammars

The C++ specified regular expression engine can cope with several different grammars. We suggest that you stick with the ECMAScript grammar as this is pretty much a superset of the available features. However, you may be more familiar with one of the other grammars and prefer to use it. Note that `(...)` is used for capture groups (so you must use `\(` and `\)` to match parentheses) except for the basic and grep grammars. The available grammars are (together with links to web-based descriptions that worked in 2014):

| | |
|---|---|
| `ECMAScript` | Use the script grammar as described for ECMAScript (JavaScript). This has the most supported grammar features of all the supported grammars. |
| `Basic` | Use the basic POSIX script grammar. This uses `\(...\)` to capture a group. |
| `Extended` | Use the extended POSIX script grammar. |
| `Awk` | Use the awk grammar. |
| `Grep` | Use the POSIX grep grammar. This is the same as `Basic`, but accepts `\n` as well as `|` for alternation. |
| `Egrep` | Use the POSIX egrep grammar. This is the same as `Extended`, but accepts `\n` as well as `|` for alternation. |

We are using the Microsoft supplied regular expression library and you can find their description of the differences between supported grammars at http://msdn.microsoft.com/en-us/library/bb982727.aspx. This reference also has the Microsoft description of regular expressions.

The grammar used by the editor Find dialog is a cut down version of the Basic syntax.

# Regular expressions

Using regular expressions is a huge topic, well beyond the reach of the Signal online help. This section attempts to get you started with the basics of the ECMAScript regular expression grammar. If you need more than this you can look it up online; there are many sites that will tell you more than you ever wanted to know, for example this one. If you are really serious about regular expressions, there are programs that will analyse and debug your regular expressions for you (for example, RegexBuddy). The other supported regular expression grammars are basically subsets of this one with some syntax differences.

### What does a regular expression do?
Basically, it works through target text and either matches some part (or maybe all) of the text, or it does not. If it matches the text, you have the option of saving sub-matches that were found along the way. You are most likely to use regular expressions for verifying that text matches some particular specification, or when transforming input text. Both the regular expression and the target text are processed from left to right, but there are circumstances where the search will backtrack (which can cause problems). A lot of the skill in writing a good regular expression is to avoid backtracking as much as possible.

### Special characters
The characters `^ $ \ . * + ? ( ) [ ] {` and `}` are special. All other characters just stand for themselves, so a regular expression of `"rat"` will match the same letters in `"scratch"`.

### Single character matches
Any character except one of the special characters listed above, matches itself. A dot matches any single character except an end of line character.

### Escapes
If you want to match any of these special characters you must put a `\` in front of them. Beware: if the regular expression is being provided by a script literal, you will need to double the `\` to `\\` to get it past the Script compiler. For example if you want to match the parenthesised text in `"cat (tiger) and"` you would need to use a regular expression of `"\\(tiger\\)"`, which becomes `\(tiger\)` after being processed by the script compiler:

You can also use escapes for the file format escape characters: form feed `\f`, new line `\n`, carriage return `\r`, tab `\t` and vertical tab `\v`. Don't forget to double the `\` in literal strings: tab is `"\\t"`, for example. There is also an escape `\0` (`"\\0"` in literal strings) for the character with code 0, which you should not need in Signal.

### Ranges

You can state that instead of matching a single, defined character, you can match a whole range of characters. This is done using square braces holding the list of matching characters. You can also provide a list of non-matching characters:

| | |
|---|---|
| `[0-9a-zABC-]` | Matches the digits 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, the lower case letters z through z, the upper case letters A, B and C and a hyphen. A minus sign (-) at the start or end of the character list stands for itself. Otherwise, it stands for an inclusive character range. The range a-c is the characters a, b, c. The range c-a is an error because c comes after a in the collation order. |
| `[^-1357A-Z]` | Matches any character that is not in the list. So this will not match the minus sign, the odd digits, or any upper-case letter (unless ignore case is set, when it will not match a-z and A-Z). |

You could match both "scare" and "score" with "sc[ao]re". You do not need to escape the special characters except ] and \ inside the range brackets as there is no possibility of using them for other purposes.

### Character classes

You can refer to classes of characters by class names. These mostly use up more space than typing the range equivalent, but are useful in the Unicode context as they cover more than the ASCII characters. A class is specified by [: followed by a name followed by :]. You can use these constructs within a range. The names recognised are:

| | |
|---|---|
| `alnum` | Lower case and upper case letters and decimal digits. Underline is considered a letter. |
| `alpha` | Lower case and upper case letters |
| `blank` | Space or tab |
| `cntrl` | The file format escape characters |
| `digits` | The decimal digits |
| `d` | Same as `digits` |
| `graph` | Lower case and upper case letters, decimal digits and punctuation |
| `lower` | Lower case letters |
| `print` | Lower case and upper case letters, decimal digits, punctuation and space or tab |
| `punct` | Punctuation |
| `space` | Space characters (not tab) |
| `s` | Same as `space` |
| `upper` | Upper case letters |
| `w` | Same as `alnum` |
| `xdigit` | Hexadecimal digits `[0-9a-fA-f]` |

To use these classes, include them within a range or a not-range. For example to include hexadecimal characters and space and the letter z: `[[:xdigit:][:space:]z]`. To exclude all digits you could use: `[^[:d:]]`. If you forget the range brackets `[:d:]` will match the letter d and a colon.

### dsw character escapes

These provide a useful shorthand for ranges (but note that you will have to double the `\` in a string literal). You can use these both in a range, where they stand for the equivalent character class, and outside a range, where they are equivalent to a range of the character class.

| Escape | Same as | Use |
|---|---|---|
| `\d` | `[[:d:]]` | Matches decimal digit. |
| `\D` | `[^[:d:]]` | Not decimal digit. |
| `\s` | `[[:s:]]` | Matches space. |

| | | |
|---|---|---|
| `\S` | `[^[:s` `]]` | Not space. |
| `\w` | `[[:w` `]` | Lower and upper case letters and decimal digits. |
| `\W` | `[^[:w` `]]` | Not lower and upper case letters and decimal digits. |

## Capturing and non-capturing groups and back references

`(stuff)` This groups the `stuff` within the parentheses and saves it as capture group n, where n is the count of the left parentheses from the start of the search. The first group is numbered 1. The group then behaves as a single element. You can then use `\1`, `\2` and so on (`"\\1"` in a literal string) to refer back to a previously matched group. So the literal string `"(cat)\\1"` will match "catcat". Note that the `Basic` and `Grep` grammars treat ( and ) as normal characters and you must escape them as `\(` and `\)` for use as syntax elements.

`(?` `:stuff)` This groups the `stuff` within the parentheses for lexical purposes, but does not capture it

It is possible to have captured text within a repeated group. In this case, the captured text is the last capture.

## Alternation

The vertical bar separates alternatives. Thus `"dog|cat"` will match either domestic pet. You can use groups to limit the scope of the alternation. `"gr(a|e)y"` will match `"grey"` or `"gray"` and capture the character, `"gr(?:a|e)y"` will match both and not capture the character.

## Quantifiers (repeats) and greedy/non-greedy matching

A quantifier refers to

| | |
|---|---|
| `{n}` | Matches the previous entity is repeated exactly n times (n is composed of decimal digits). For example `"x{10}"` matches `"xxxxxxxxxx"`. |
| `{min,max}` | Matches the previous entry repeated between `min` and `max` times (`min` and `max` are composed of decimal digits). `min` can be 0, `max` must be the same as or more than `min`. |
| `{min,}` | Matches the previous entry repeated at least `min` times. |
| `*` | Matches the previous entry 0 or more times. Equivalent to `{0,}`. |
| `+` | Matches the previous entry 1 or more times. Equivalent to `{1,}`. |
| `?` | Matches the previous entry 0 or 1 times. Equivalent to `{0,1}`. |

You can follow any qualifier with a `?` to convert it from a greedy match to a non-greedy match. Normally, a match will consume as much as possible of the text while still matching the data. For example, the expression `".*end"` will match all of `"it depends on what you spend"`. However, `".*?end"` will match `"it depend"`. You should think very carefully about using unlimited greedy matches as these cause backtracking if a subsequent part of the match fails. Backtracking is where a greedy match has taken as much text as it can such that a subsequent portion of the regular expression fails. The search then tries to match a bit less in an attempt to match the next part of the expression and then a bit less again. In a bad case, this can cause the search to take so long that it never ends, or it can use up so much memory that the system crashes. By default, the regular expression search will time out after 1 second.

## Anchors and boundaries

These items are a little different from the matches we have seen so far as they do not consume characters in the target text, but they state a condition that must be true for a match to continue. You can think of these as asserting that a condition must be true.

| | |
|---|---|
| `^` | Assert that the current position is the start of the text or start of a line. `"^T"` will only match text/lines that begin with a capital T. You can use `remNotBol` to state that the start of the text is not at the start of a line. |
| `$` | Assert that the current position is the end of the text or a line. `"end.$"` will match the final `"end."`, not the one in `"spend."` in `"Do not spend. That is the end."` You can use `remNotEol` to state that the end of the text is not the end of a line. |

| | |
|---|---|
| \b | Asserts that the current position is a word boundary. That is, the previous character is not part of the alnum class and the current one is part of the alnum class or vice versa. The start of the text is also considered a word boundary unless you use the remNotBow flag when it is not a word boundary. The end of the text is also considered a word boundary unless you use the remNotEow flag. |
| \B | Asserts that the current position is not a word boundary, that is both the previous and current characters are part of the alnum class, or they are both not part of the class. |
| (?=expr) | Assert that the following text matches the regular expression expr but does not move the search position past expr. |
| (?!expr) | Assert that the following text does not match expr. |

# Interact()

This function provides a quick and easy way to interact with a user. Cursors can always be dragged as we assume that they are one of the main ways of interacting with the data. You can restrict the user to a single view and limit the menu commands that can be used.

```
Func Interact(msg$, allow%{, help{, lb1${, lb2${, lb3$...}}}});
```

msg$    The prompt to display in the tool bar during the operation. If there is not enough space to display the message and buttons, the message is truncated.

allow%  A code that specifies the actions that the user can and cannot take while interacting with Signal. The code is the sum of possible activities:

| | | |
|---:|---|---|
| 1 | 0x0001 | User may swap to other applications |
| 2 | 0x0002 | User may change the current window |
| 4 | 0x0004 | User may move and resize windows |
| 8 | 0x0008 | User may use File menu |
| 16 | 0x0010 | User may use Edit menu |
| 32 | 0x0020 | User may use View menu |
| 64 | 0x0040 | User may use Analysis menu |
| 128 | 0x0080 | User may use Cursor menu and add cursors |
| 256 | 0x0100 | User may use Window menu |
| 512 | 0x0200 | User may use Sample menu |
| 1024 | 0x0400 | User may not double click y axis |
| 2048 | 0x0800 | User may not double click the x axis or scroll it |
| 4096 | 0x1000 | User may not change channel of horizontal cursors |
| 8192 | 0x2000 | User may not change to another frame |

A value of 0 would restrict the user to inspecting data and positioning cursors in a single, unmoveable window, but being able to switch frames. A value of 8192 is the same but without changing frames.

help    This can be either a number or a string. If it is a number, it is the number of a help item (if help is supported). If it is a string, it is a help context string. This is used to set the help information that is presented when the user requests help in the manner supported on the host machine. Set 0 to accept the default help.

lb1$    These label strings create buttons, from right to left, in the tool bar. If no labels are given, one label is displayed with the text "OK". The maximum number of buttons is 17. Buttons can be linked to the keyboard using & and by adding a vertical bar followed by a key code to the end of the label. You can also set a tooltip. The format is "Label|code|tip". See the documentation for label$ in the ToolbarSet() command for details.

Returns The number of the button that was pressed. Buttons are numbered in order, so lb1$ is button 1, lb2$ is button 2 and so on.

With `allow%` set to 0, all the user could do would be to press a button on the tool bar. The tool bar would be displayed (if it was not present) when `Interact()` was called. When the user presses a button to exit, the tool bar is returned to the state it was in before Interact was used.

**See also:**
`Toolbar()`

# L

# LastTime()

This function finds the first item on a channel before a particular x axis position.

```
Func LastTime(chan%, &pos{, &val|code%|code%[]{, &rval|rval[]}});
```

chan%   The channel number (1 to n) to use for the search.

pos     The x axis value to search before. Items at the start position are ignored. To start a backward search that guarantees to iterate through all items, start at `Maxtime(chan%)+1`.

        pos is updated to contain the x axis position of the previous item. It is left unchanged if no more items are found or there is an error.

val     This optional parameter returns the waveform value for waveform channels.

code%   This optional parameter is only used if the channel is a marker type, it can be an array that is filled in with the marker code values or a single integer variable that is updated with the first code value. Each marker has four code values, the data returned in `code%[]` is the lesser of 4 and the array size.

rval    This optional parameter is only used if the channel is real marker type, it is either an array that is filled in with the real marker float values or a single real variable that is updated with the first real marker float value. Real markers can have one or more float values, the data returned in `rval[]` is the lesser of the data available and the array size.

Returns The function returns 1 if a data item is found, 0 if there are no more items to be found or a negative error code.

**See also:**
`Maxtime(), Mintime(), NextTime()`

# LCase$()

This function converts a string into lower case.

```
Func LCase$(text$);
```

text$   The string to convert.

Returns  A lower cased version of the original string.

**See also:**
`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 or the size of the n$^{th}$ dimension of an array.

```
Func Len(text$);
Func Len(const arr[]);
Func Len(const nd[]{[]...}, n%)   added at version 7.00
```

text$   The text string.

arr[]   A one dimensional array of any kind. It is an error to pass in a two dimensional array.

nd[]    A one or more dimensional array.

n%      The index of the dimension to return the size of.

Returns  The length of the string or the array, as an integer.

You can find out the size of each dimension of a two dimensional array as follows:

```
proc something(arr[][])        'function passed a 2-d array
var n%; n% := Len(arr[][0]);   'get size of first dimension
var m%; m% := Len(arr[0][]);   'get size of second dimension
```

The largest multi-dimensional array we support is 5-d, so here is the code for finding the sizes of all the dimensions of a 5-d array.

```
proc arrayMax(a[][][][][])
var d1% := Len(a[][0][0][0][0]);
var d2% := Len(a[0][][0][0][0]);
var d3% := Len(a[0][0][][0][0]);
var d4% := Len(a[0][0][0][][0]);
var d5% := Len(a[0][0][0][0][]);
```

From version 5.09 onwards you can pass an array with a 0 length to Len(), and the result is 0 rather than a fatal script error.

From version 7.00 you can use the third variant, so the previous example becomes:

```
proc arrayMax(a[][][][][])
var d1% := Len(a, 0);
var d2% := Len(a, 1);
var d3% := Len(a, 2);
var d4% := Len(a, 3);
var d5% := Len(a, 4);
```

This also allows you to detect a zero length dimension as you are not allowed to create a sub-array of a zero-sized array.

```
var a[2][0];
if Len(a[0][]) == 0 then 'Syntax error as a has zero size
```

but

```
if Len(a, 1) == 0 then 'OK as not creating a sub-array
```

**Unicode**

Versions of Signal from 6.03 onwards are Unicode enabled. The underlying storage used for strings is a vector of 16-bit values. Most characters fit in one 16-bit value, but some require two. The Len() function returns the number of 16-bit values required to hold the string, not the number of characters.

If you are running the Unicode version of Signal, you may wish to know the length of a string if it were written as UTF-8 (for example for use by BWriteSize()). The following function will return the length of the UTF-8 equivalent in bytes.

```
func LenUTF8%(str$)
var i% := 1, bytes%, c%; ' bytes% initialised to 0
var n% := Len(str$);
while i% <= n% do
    c% := Asc(Mid$(str$, i%));  'get code of next char or surrogate pair
```

```
    docase
    case c% < 0x80 then bytes% += 1;      'ASCII code
    case c% < 0x800 then bytes% += 2;    'needs 2 bytes
    case c% < 0x10000 then bytes% += 3;  'needs 3 bytes
    else bytes% += 4; i% +=1;             'Surrogate pair
    endcase;
    i% += 1;
wend;
return bytes%;
end;
```

**See also:**
Asc(), Chr$(), DelStr$(), InStr(), LCase$(), Left$(), Mid$(), Print$(), Right$(),
Str$(), UCase$(), Val()

# LinPred()

Linear prediction can be used to predict future (or past) data values based on a sequence of data values on the assumption that the data is statistically stationary. It can also be used to estimate power spectra using the Maximum Entropy or All Poles method. The command generates a set of coefficients that when applied to the previous m points, generate the next predicted point. Some of the explanation for this command relies on technical knowledge; see the references for more information. The command has the following variants:

```
Func LinPred(const data[], mMax%{, limit{, out[]{, dir%{, stab%}}}});
Func LinPred(0, out[]{, data[]});      'Predict forward
Func LinPred(1, out[]{, data[]});      'Predict backwards
Func LinPred(2, stab%);                'Check stability
Func LinPred(3, coef[]);               'Get the coefficients
Func LinPred(4, refl[]);               'Get reflection coefficients
Func LinPred(5, power[], frLo, frHi);  'Get estimated power spectrum
Func LinPred(6, poles[][]{, fr[]});    'get poles and frequencies
```

data An array holding the data to be used to form the linear prediction coefficients or to be used to initialise a prediction based on existing coefficients.

mMax% The maximum order of the prediction, which is the number of previous points to use to predict each future point. The actual number may be less than this, depending on the value of the limit argument. You will generally want to use the smallest value of m that you can; values in the range 5-50 are common, but this does depend on the data. The value of m must be less than the number of data points and is generally much less. We have set an upper bound of 1024 on mMax% (this is higher than you are likely to need).

limit The algorithm to calculate the poles is an iterative procedure. The command holds an array of residual values that is initialised to the raw data. Each iteration subtracts data from the residual based on a normalised autocorrelation of the residual (in the range -1 to 1) at increasing lags, aiming to reduce the residual array to a list of zeros. We track a number that models the significance of the remaining data. This number is 1.0 at the start, and is multiplied by $(1-ac^2)$ at each iteration, where ac is the autocorrelation, so the value decreases at each iteration unless ac is 0. If this value becomes less than limit, the iteration will stop. So set limit to 0 for no early stopping, setting a small value above 1.0e-31 may stop it early. The actual value of m used is returned by the first command variant.

out An array of output values predicted by the command. When predicting backwards, the data values are written into the out array so that the first data item in the array is the last predicted point (the oldest). When predicting forwards, the first item in the out array is the first predicted point.

dir% Optional argument to the setup command that sets the prediction direction. Set 0 or omit for a forward prediction, 1 for reverse.

stab% The linear prediction coefficients that are generated form a characteristic polynomial, and the roots of this polynomial are the positions of the poles in the z-plane. For the resulting filter to be stable (have an output that does not increase exponentially), the poles must lie inside (corresponding to decaying sinusoids) or on (corresponding to constant amplitude sinusoids) the unit circle. This algorithm should not produce poles outside the unit circle unless there are numerical accuracy problems (usually when mMax% is large). You can use this argument to check the stability of the result and to adjust the pole positions and recompute the coefficients by setting the values:

0    Do nothing to the poles.

1  If a pole lies outside the unit circle, reflect it across the unit circle so that a growing sinusoid becomes a decaying sinusoid.

2  If a pole lies outside the unit circle, move it onto the unit circle, corresponding to a constant sinusoid.

3  Move all poles onto the unit circle. This produces an output that neither grows nor decays with time. However, this is very prone to numerical stability problems and is likely to be interesting only when `mMax%` is small.

`coef`   An array to be filled in with coefficients. The array can be any size, but only the points corresponding with coefficients will be set.

`refl`   An array to be filled with reflection coefficients. These values represent the proportion of the residual that was removed at each iteration. These are the values that are modified by the Levinson recursion to form the coefficients.

`power`  An array to be filled in with estimated power spectrum components.

`frLo`   A value in the range -0.5 to 0.5, being the fraction of the sampling frequency that the first bin of `power` will hold the estimated power spectrum for.

`frHi`   A value in the range -0.5 to 0.5, being the fraction of the sampling frequency that the last bin of `power` will hold the estimated power spectrum for.

`poles`  A matrix with the second dimension of size at least 2. `poles[n][0]` is returned holding the real component of the n$^{th}$ pole, and `poles[n][1]` holds the imaginary component.

`fr`     An array to be filled in with the frequencies that correspond to the pole positions, in the range -0.5 to 0.5 (fraction of the sampling rate).

### Setup
```
Func LinPred(data[], mMax%{, limit{, out[]{, dir%{, stab%}}}});
```

This command must be used before any other as it calculates the initial set of coefficients. The command returns the number of coefficients that have been generated (this will be the same as `mMax%` if `limit` is 0). The command can be used to generate predicted data, or to set up the system for further `LinPred()` commands. The following example takes 1000 data points from near the start of a waveform channel, predicts the next 100 points and writes them to a memory channel.

```
const chan% := 1;                         ' a waveform channel
var data[1000], out[100];
var f%;
f% := FileOpen("*.cfs", 0);               ' Open a file containing test data
arrconst(data[], view(f%, chan%).[0:1000]); ' copy data into array
var m% := LinPred(data, 20, 0, out, 0, 0); ' predict forwards
```

The predicted data will not be the same as the actual data that follows the first 1000 points unless the first 1100 points are composed of the sum of constant amplitude sinusoids. The command forms a mathematical model of the data held in the `data[]` array based on the assumption that the spectral components are not changing with time (the signal is stationary) and that the data is modelled by a set of resonances. Unless you set `stab%` to 3, the result will usually decay with time.

### Predict forwards and backwards
```
Func LinPred(0, out[]{, data[]});          'Predict forward
Func LinPred(1, out[]{, data[]});          'Predict backwards
```

These two command versions fill the `out[]` array with data predicted by the coefficients established by the Setup version of the command. The setup command `dir%` argument will have set up the command to generate output that joins up with the original data array, or with any output if the setup command generated output. You can choose to continue generating output in the same direction, in which case you must NOT supply the `data[]` array, or you can supply a `data[]` array of at least the size of the number of coefficients to reset the prediction and you can then run forwards or backwards. By supplying a `data[]` array, you are not recalculating the coefficients, these remain unchanged; you are reloading the data points that are used with the coefficients to predict values. If you have m coefficients, when going forwards, the last m data points of `data[]` are used; when going backwards, the first m data points. For example, if we wanted to extend the previous example to predict the 100 data points that might have led up to the original 1000 points we could add the lines:

```
var back[100];
LinPred(1, back, data);                    ' predict back from start
```

If we had just used `LinPred(1, back);` this would have caused an error as the previous use of the command was to go forwards. These command variants return 0.

## Check stability

```
Func LinPred(2, stab%);                    'Check stability
```

This command variant is used to check that the poles of the characteristic polynomial lie within the unit circle. The command returns the distance of the pole furthest from the origin of the z-plane. This should be less than or equal to 1.0 for a stable set of coefficients. Stable, in this context, means that the predicted data does not grow exponentially. The algorithms we use should generate stable solutions, but poles can be generated outside the unit circle due to loss of numerical precision in the calculations. The `stab%` argument can be used to adjust the pole positions, as described above. If you are using this command to replace a short stretch of damaged data (like fixing a scratch in a record), you may want to predict both forwards and backwards across the damaged data, then mix the two predictions together. If the data used to generate the coefficients is not stationary, it will decay across the gap, in which case using `stab%` set to 3 will generate a result that maintains its amplitude, which may be what you require.

## Get coefficients

```
Func LinPred(3, coef[]);                    'Get the coefficients
```

This function returns the number of coefficients and returns them in the `coef[]` array. Note that the first coefficient is the one that is multiplied with the most recent data point (when going forward), that is the coefficients run backwards compared to the data. Given an array `x[]` of data and coefficient `coef[]`, both of length `m%` points, the next forward and backward predictions are given by:

```
var i%, fwd:=0, rev:=0;
for i% := 0 to m%-1 do
   fwd += x[m%-i%-1] * coef[i%];
   rev += x[i%] * coef[i%];
next;
```

It is usually much simpler to use the forward and backward prediction versions of the command to do this. If you cannot do this, for example when you need to process several waveforms simultaneously, save the coefficients for each waveform (and reverse the order if predicting forwards), then you can use `ArrDot()` to predict each new point.

## Get reflection coefficients

```
Func LinPred(4, refl[]);                    'Get reflection coefficients
```

This function returns the number of reflection values that are available (the same value as the number of coefficients) and sets `refl` to the results of the auto-correlations done at each lag as the original data was analysed. The results are normalised such that the results lie in the range -1 to 1. The first value is the autocorrelation at a lag of 1 point, the next is for a lag of 2 points, and so on. The term reflection comes from the use of this technique in seismology.

## Power spectrum

```
Func LinPred(5, power[], frLo, frHi); 'Get estimated power spectrum
```

The variant generates an estimate of the power spectrum based on the representation of the original as the sum of a set of resonances. You should leave the `stab%` argument at 0 when using the command for this purpose. This method is particularly effective if you know that the data contains a small number of constant frequency sinusoids and can allow you to separate peaks that would merge into a single peak using the FFT. However, if your spectrum is not representable as the sum of a set of resonances, the result may be misleading unless `mMax%` is large enough so that the spectrum can be approximated. Unlike the FFT, where the resolution of the result depends on the number of points in the transform, here the resolution of a particular peak is determined by the number of bins and the frequency range that you set. Make sure you set the bin width small enough so you do not miss a very narrow resonance. The frequencies are defined in terms of a fraction of the sampling rates
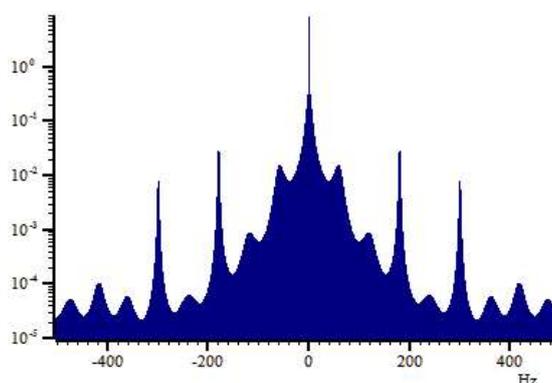


*Typical Power spectrum*

from -0.5 to 0.5 (but note that -0.5 is the same frequency as 0.5); you will find that the power at frequency *f* is equal to the power at frequency *-f*.

The result is scaled so that the integral of the power from -0.5 up to (but not including) 0.5 is equal to the mean square of the values in the original `data[]` array used in the setup call. For example, we could extend the previous examples to display the power with:

```
const bins% := 10001;                    ' Bins in spectrum
var bsz := BinSize(chan%);                ' source data rate
var power[bins%];                         ' Array to hold the power
LinPred(5, power, -0.5, 0.5);             ' NB: -0.5 is same as 0.5
var rv% := SetMemory(1, bins%, 1/(bsz*(bins%-1)), -0.5/bsz, 0, 0, 0, "Power", "Hz");
ArrConst(view(rv%, 1).[], power);Optimise(-1);WindowVisible(1); ' display the result
var MeanSq := ArrDot(data, data)/Len(data); ' Mean square of data
var MemSum := ArrSum(power[:bins%-1])/(bins%-1); 'Integral of power
Message("Mean sumSq = %g, SumPower = %g", MeanSq, MemSum);
```

The example image shows the power spectrum obtained by data with 60 Hz mains interference, leading to peaks at odd multiples of 60. If you try this with data constructed from pure sinusoids you will find that `MemSum < MeanSq`. The reason is that the purer the sinusoid, the higher and narrower the peak in the power spectrum; simple schemes for adding up equal width bins will not give an accurate result in this case. In the limiting case where the roots of the characteristic polynomial lie on the unit circle, the resonance has infinite amplitude, but zero width.

You might be puzzled about how this method can get much better frequency resolution than the FFT. The basic reason is that this method makes the assumption that the data continues in *a sensible way* outside the range of input points so as to preserve the auto-correlation between the data values. The FFT assumes that the data repeats exactly outside the initial range of data, leading to a limit on the frequency resolution.

### Get Poles and frequencies
`Func LinPred(6, poles[][]{, fr[]});      'get poles and frequencies`

In the z-plane, the coefficients are represented by a set of poles. You can use this variant to read back the positions of the poles as complex numbers. The poles are sorted in order by their real components, so the pole pairs for each imaginary root should be adjacent. You can also read back the frequencies (as a fraction of the sample rate in the range -0.5 to 0.5) at which the poles are located.

### Predicting data across a gap
If you are predicting data across a gap (a very common use of this function), you will find that your choice of `mMax %` is critical. If your data is periodic, for example a blood pressure signal and there are around *p* points in the period, then you will find that setting `mMax%` to *p* (the exact value is not critical) will usually work. If your data is the sum of a small number of sinusoids, you will find that a value of around twice the number of sinusoids may work. In other cases, if you are predicting across a gap of *g* points, then setting `mMax%` to *g* (again the exact value is not critical) may work.

It is a good idea to predict forwards and backwards across the gap, then merge the two predictions using some sort of weighting function that starts with all forwards prediction at the start and all backwards prediction at the end.

Remember that the mathematics makes the assumption that the data used for the prediction is stationary; that is that the statistical properties of the data (power spectra) remain constant with time. In many cases this will not be true; this leads to predicted data that decays away to the mean value.

The stability parameter should usually be 0. If you find that your predicted values are growing exponentially, then you can try setting the `stab%` argument to 1 or 2, which should stop this. However, the more normal problem is for the data to decay away. Setting `stab%` to 3 is a rather drastic act as it moves all the sinusoidal resonances that are used to model the power spectrum of the data onto the unit circle (converts them all into resonances with infinite Q and zero width). This may be useful if you have a small value of `mMax%` and you know that the data consists of constant sinusoids.

### References
Claerbout, Jon F. (1976). "Chapter 7 - Waveform Applications of Least-Squares." Fundamentals of Geophysical Data Processing. Palo Alto: Blackwell Scientific Publications. This has an explanation of the John P Burg algorithm that we implement.

Levinson recursion is a method for inverting a Toeplitz matrix in $O(n^2)$ time, taking advantage of the symmetry of the matrix, and improving on the $O(n^3)$ time of a general inversion (*n* is the order of the prediction). Symmetrical Toeplitz matrices come about as a natural consequence of the linear prediction equations. However, implementing

the obvious equations often results in unsatisfactory solutions, and the Burg algorithm is a better approach that incorporates the iterative idea of the Levinson recursion, but calculates the autocorrelation in a different way that leads to stable solutions.

There is a good overview of linear prediction and the Maximum Entropy (All Poles) method of power spectrum estimation in Numerical Recipes, The Art of Scientific Computing, by Press, Flannery, Teukolsky and Vettering.

# Ln()

This function calculates the natural logarithm (inverse of `Exp()`) of an expression, or replaces the elements of an array with their natural logarithms.

```
Func Ln(x|x[]{[]...});
```

x      A real number or a real array. Zero or negative numbers cause the script to halt with an error unless the argument is an array, when an error code is returned.

Returns  When used with an array, it returns 0 if all was well, or a negative error code. When used with an expression, it returns the natural logarithm of the argument.

**See also:**
Abs(), ATan(), Cos(), Exp(), Frac(), Log(), Max(), Min(), Pow(), Rand(), Round(),
Sin(), Sqrt(), Tan(), Trunc()

# LnGamma()

This function returns the natural logarithm of the gamma function *Gamma(x)* for real values of $x > 0.0$. *Gamma(x)* has the useful property that *Gamma(n+1)* is the same as *n*! (*n* factorial) for integral values of *n*. However, it increases very rapidly with *x*, reaching floating-point infinity when *x* is 172.62. To avoid this problem, the script returns the natural logarithm of the gamma function. The definition of the gamma function is:

*Gamma(a)* is the integral between 0 and infinity of `exp(-t)*pow(t,a-1)` with respect to `t`.

```
Func LnGamma(a);
```

a      A positive value. The script stops with a fatal error if this is negative.

Returns  The natural logarithm of the Gamma function of `a`.

**See also:**
GammaP()

# 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 function returns the view handle of the log window. The log window, also called the log view, is a text view created by the application and is the destination for `PrintLog()`. You need this if you are to size or hide the log window, or make it the current or front window, or use the editing commands to clear it.

```
Func LogHandle();
```

Returns  The view handle of the log window.

**See also:**
```
EditClear(), EditSelectAll(), View(), FrontView(), Window(), WindowGetPos(),
WindowSize(), WindowVisible()
```

# M

# MarkCode()

This returns the data stored in a marker at a particular x axis position.

```
Func MarkCode(chan%, pos{, code%|code%[]{, &rval|rval[]}});
```

chan%   The marker channel to read, this can be a simple marker or real marker channel.

pos     The marker time. This must match to within ± half the time interval returned by `BinSize()` for the channel.

code%   This optional parameter can be an array that is filled in with the marker code values or a single integer variable that is updated with the first code value. Each marker has four code values, the number of values returned in `code%[]` is the lesser of 4 and the array size.

rval    This optional parameter is only used if the channel is real marker type, it is either an array that is filled in with the real marker float values or a single real variable that is updated with the first real marker float value. Real markers can have one or more float values, the number of values returned in `rval[]` is the lesser of the count of values available and the array size.

Returns  The first code if a marker was found, or -1 if no marker exists at time `pos`.

**See also:**
```
BinSize(), MarkEdit(), MarkTime()
```

# MarkEdit()

This changes the data stored in a marker at a particular x axis position.

```
Func MarkEdit(chan%, pos, code%|code%[]{, &rval|rval[]});
```

chan%   The marker channel to edit, this can be a simple marker or real marker channel.

pos     The marker time. This must match to within ± half the time interval returned by `BinSize()` for the channel.

code%   This is either a single value from 0 to 255 to replace the first code for the marker or an array of up to 4 values (from 0 to 255) to replace codes for the marker. If the array size is smaller than 4 the other codes are left untouched.

rval    This optional parameter is only used for real marker channels, it is either a single real value that will replace first float value of the real marker or an array of reals to replace float values for the marker. If the array size is smaller than the number of floats in the marker the other values are left untouched.

Returns  0 if a marker was edited, or -1 if no marker exists at time `pos`.

**See also:**
```
BinSize(), MarkCode(), MarkTime()
```

# MarkInfo()

This function returns the number of real values stored with each item of a real marker channel, it was added in version 6.03 of Signal to match Spike2 and as a tidier way of getting this information than by using `ChanIndex()`.

```
Func MarkInfo(chan%);
```

chan%   The marker channel to test, this should be a real marker channel.

Returns  The number of real values per item for the real marker channel, or 0 if the channel was not real marker type.

**See also:**
```
ChanIndex(), MemChan()
```

# MarkShow()

This sets which of the four marker codes to display for one or more marker channels. Setting the default draw mode with DrawMode(chan%,0) resets the displayed code to 0. This is equivalent to the Use marker code field in the channel Draw Mode dialog.

```
Func MarkShow(cSpc{, code%});
```

cSpc    A channel specifier or -1 for all, -2 for visible or -3 for selected channels.

code%   This optional argument sets which of the 4 marker codes to display in the range 0 to 3, omitted or -1 for no change.

Returns The original marker code display selection (0-3).

**See also:**
```
DrawMode(), LastTime(), MarkEdit(), MemGetItem(), NextTime()
```

# MarkTime()

This reads and changes the time for a marker.

```
Func MarkTime(chan%, pos{, new});
```

chan%   The marker channel holding data to move, this can be a simple marker or real marker channel.

pos     The marker time. This must match to within ± half the time interval returned by BinSize() for the marker channel.

new     If supplied, the new time (x axis value) for the marker. Note that marker times must be in order, so this time will be truncated to prevent the marker time reaching or going past adjacent markers.

Returns  The exact marker time before any changes or 0 if no marker exists at time pos.

**See also:**
```
BinSize(), MarkCode(), MarkEdit()
```

# MATDet()

This calculates the determinant of a matrix (a two dimensional array).

```
Func MATDet(const mat[][]);
```

mat     A two dimensional array with the same number of rows and columns.

Returns  The determinant of mat or 0.0 if the matrix is singular.

**See also:**
```
ArrAdd(), MATInv(), MATMul(), MATTrans()
```

# MATInv()

This inverts a matrix (a two dimensional array) and optionally returns the determinant.

```
Func MATInv(inv[][]{, const src[][]{, &det}});
```

inv     A two dimensional array to hold the result. If src is omitted, inv is replaced by its own inverse. The number of rows and columns of inv must be the same.

src     If present, the matrix to invert. The numbers of rows and columns of this two dimensional array must be at least as large as inv.

det    If present, returned holding the determinant of the inverted matrix.

Returns  0 if all was OK, -1 if the matrix was singular or very close to singular.

**See also:**
ArrAdd(), MATMul()

# MatLab Script Support

If you have a copy of MATLAB installed on your computer, the `MatLabXxx()` family of script commands let you start a MATLAB process for the purpose of using MATLAB as a computational engine. This process can have a visible window allowing some user interaction, but it is not a full MATLAB workspace and will be separate from any normally opened MATLAB workspaces that you use. You can transfer script variables and arrays (but not arrays of strings) to the MATLAB workspace, command MATLAB to process your data, then move results back into the script language. You must select the MATLAB script support option when installing Signal, otherwise these commands will not be installed. These functions were added to Signal in version 5.02.

**See also:**
MatLabOpen(), MatLabClose(), MatLabPut(), MatLabGet(), MatLabEval(), MatLabShow()

# MatLabClose()

This function closes a connection to MATLAB that was created using `MatLabOpen()`.

**Proc MatLabClose();**

**See also:**
MatLabOpen(), MatLabPut(), MatLabGet(), MatLabEval(), MatLabShow()

# MatLabEval()

This function requests the MATLAB command window to execute an arbitrary command and gets its result as a string. Typical use of this would be to set up data with `MatLabPut()`, process it with `MatLabEval()`, then retrieve the result with `MatLabGet()`.

**Func MatLabEval(cmd$ {,&resp$});**

cmd$   This is the command string that is sent to MATLAB.

resp$  Optional. If present, it is set to the response from MATLAB. For example, it is set to an error message if the command fails. The maximum length of the response is 511 characters.

Returns  0 if cmd$ was successfully passed to MATLAB and -1 if it was not.

**See also:**
MatLabOpen(), MatLabClose(), MatLabGet(), MatLabPut(), MatLabShow()

# MatLabGet()

This function copies a variable from the MATLAB workspace into a script variable.

**Func MatLabGet(name$, &v%|&v|&v$|v%[]{[]...]|v[]{[]...});**

name$  the name of the MATLAB variable to be copied. If the variable does not exist the function will return -1 and script execution continues. MATLAB variables that are structures, cell arrays, functions, logical (BOOLEAN) data or 64-bit integer data are not supported. Of the remaining types of data (basically character and numeric values), the allowed types vary with the type of script language variable supplied.

v$     For a string variable, the MATLAB variable must be a simple string (a 1-dimensional array of characters). String arrays are not supported.

v% v   For non-array integer and real variables, the MATLAB variable must be a numeric value. Integer and floating-point data are automatically converted, floating point values put into an integer variable are truncated.

v[]     For real and integer script arrays, the MATLAB variable must be a vector with compatible numbers and sizes of dimensions: a 1-dimensional script array of length n matches an n x 1 vector (or a 1 x n vector), a 2 dimensional script array `arr[m][n]` matches an m x n vector and so forth. The size of the last dimension of the script array is allowed to be larger than or equal to the size of the last dimension of the MATLAB vector so an m x n vector can be transferred into `arr[m][n]`, `arr[m][n+1] ... arr[m][n+x]`. MATLAB floating-point types are converted to script reals and MATLAB integer types are converted to script integers. Converting between integer and floating-point types is not supported.

Returns -1 if the variable was not found in MATLAB or a type match error, otherwise it returns 0 for a simple variable or the size of the last dimension of the script array that was filled.

**See also:**
`MatLabOpen()`, `MatLabClose()`, `MatLabPut()`, `MatLabEval()`, `MatLabShow()`

# MatLabOpen()

This function opens a connection to an invisible MATLAB command window. All of the other `MatLabXxx()` script commands fail if there is no connection. Making a connection takes a detectable time; it is inadvisable to open and close the connection many times.

```
Func MatLabOpen({mode%});
```

mode%    If `mode%` is absent or set to zero, the MATLAB connection is shared, allowing other applications to use the same command window. Set `mode%` to 1 for an unshared connection, for exclusive use by the script.

Returns 0 if the MATLAB connection was opened successfully or a negative error code if the function failed. If this function fails it will generally be because Signal cannot locate the relevant DLLs that were installed with MatLab, which is most likely caused by the Windows system path being incorrect. To help with debugging connection problems, Signal will generate error messages in the log window.

**Troubleshooting the MATLAB connection**
The Signal-MATLAB connection can be difficult to set up. We have found that there are two things that are the cause of most connection problems:

*Get the correct MATLAB directory into the Windows path*
This is by far the most common issue, as the more modern MATLAB installers do not set the path up in the way that we need. The Windows path is a list of directories which are searched to find the MATLAB DLLs (a DLL is a loadable library file) that are needed for the connection. If the directory holding the required DLLs is not in the path then they will not be found and opening the connection will fail, normally with error code 126. To find the directory where your copy of the MATLAB DLLs are stored, search your hard disk for the libeng.dll file, it will normally be somewhere like C:\Program Files\MATLAB\R2016a\bin\win64. Ignore any copies of this file in the directories where Signal is installed! It is this directory which must be in the path, often you will find that only the directory one level up (C:\Program Files\MATLAB\R2016a\bin) is there.

There is a second, rather more rare, problem that can be caused by the Windows path. MATLAB uses a very large number of libraries in DLL files, quite a lot of which are third-party libraries from other sources. If another application is installed which uses the same DLLs but installs different versions from MATLAB, <u>and</u> the path to these other DLLs appears in the Windows path above the paths to the MATLAB directories, then these other DLLs will be loaded instead of the MATLAB versions, which can cause MATLAB to fail. The solution to this problem is to move both MATLAB directories up to the top of the Windows path, which will ensure that the MATLAB DLLs are found first and loaded. However this situation is quite rare, so I would recommend only trying this fix after all other attempts have failed.

**WARNING! Incorrect changes to the Windows path can damage your system and, at worst, stop it working altogether.**

**DO NOT make any changes unless you are sure that you understand these instructions and can avoid mistakes. If you are not sure, get an experienced user or administrator to help you.**

To examine and change the Windows path (this is based on a Win10 system, others will be similar):

- Open the system properties by right-clicking on This PC, either as a desktop shortcut or where it appears on the left-hand section of File Explorer. Select Properties in the menu that appears. Very probably there is a way to get this via the standard settings, but I cannot find it! It is the System section of the old fashioned control panel.

- Select Advanced system settings from the options on the left.

- In the tabbed dialog that now appears, at the bottom of the Advanced tab, click on the Environment Variables button.

- Find Path in the lower System variables section, select it, and click on the Edit button to get the Edit environment variable window.

- That will give you a list of the directories in the path. In the list, can you see the C:\Program Files\MATLAB\R2016a\bin\win64 (or wherever) directory where the MATLAB DLLs are? If not, it needs to be added.

- Press on the New button, and enter the complete path to the MATLAB DLLs, press return to complete the entry.

- Use the Move Up button to move the new entry up to immediately after the existing entry for MATLAB. Press OK to accept the changes.

- On older systems, the Win10 dialog that shows the individual path entries is not there, instead you get a much simpler edit variable window with the entire path shown as one line of text, with semicolons separating the individual entries. Editing the path with this dialog is a bit more complicated, but the idea is much the same - search through the text for the main MATLAB directory and, if the directory we want is not there, add it after the main MATLAB directory (not forgetting the required semicolon!).

- **If something goes wrong while you are adding the new entry, do not panic! Pressing Cancel will exit from the Edit environment variable window without any changes being made.**

### *Register the MATLAB server*

In order to make the connection to MATLAB, MATLAB must be registered as a suitable server. This is normally done as part of the MATLAB installation process but it can fail, or the registration can be lost, causing the connection to fail with error code 0 or 3. To fix this we need to re-register MATLAB:

- Open a command prompt window (Start menu button -> Windows system (or Accessories) -> Command prompt).

- Type the command matlab /regserver and press Return.

- That's it...

### Possible effects on MatLab file export

Various users have reported that exporting data to MatLab files works normally but fails after the `MatLabOpen()` script command has been used. This appears to be caused by DLL version incompatibilities between the MatLab DLLs installed with Signal for the purposes of file export (which should be available even if MatLab is not installed) and the DLLs installed with MatLab (which are loaded by `MatLabOpen()`). It appears that the problem can be fixed, at least in some cases, by hiding the MatLab DLLs installed with Signal so that file export uses the DLLs installed with MatLab.

The DLLs in question are all in the export directory inside the Signal export directory, they are: boost_date_time-vc100-mt-1_49.dll, boost_filesystem-vc100-mt-1_49.dll, boost_log-vc100-mt-1_49.dll, boost_regex-vc100-mt-1_49.dll, boost_signals-vc100-mt-1_49.dll, boost_system-vc100-mt-1_49.dll, boost_thread-vc100-mt-1_49.dll, hdf5dll.dll, icudt49.dll, icuin49.dll, icuio49.dll, icuuc49.dll, libeng.dll, libexpat.dll, libmat.dll, libmwfl.dll, libmwi18n.dll, libmwMATLAB_res.dll, libmwresource_core.dll, libmx.dll, libut.dll, msvcp100.dll, msvcr100.dll, tbb.dll, tbbmalloc.dll and zlib1.dll.

The best way to hide these DLLs is to put them, but not the matexp.sxl, msvcp100.dll and msvcr100.dll files that are also in the export directory, into a separate sub-directory inside the export directory. That way you will be able to get the files back easily if necessary.

### See also:
MatLabClose(), MatLabPut(), MatLabGet(), MatLabEval(), MatLabShow()

# MatLabPut()

This function takes a script variable and puts it into the MATLAB workspace. A connection with MATLAB must have been set up for this function to be used.

```
Func MatLabPut(name$, const v%|v|v$|v%[]{[]...}|v[]{[]...}{, as%});
```

name$    the name of the MATLAB variable that will be created in the workspace. If a variable of this name already exists, it is overwritten. MATLAB variable names can only use alphanumeric characters plus the underscore character '_' and must start with an alphabetic character.

v An expression or variable holding the data to move to MATLAB. The variable type created in the MATLAB workspace is set by the type of `v` and the `as%` argument. All of the script language variable types are supported.

v[] A script array variable holding the data to move to MATLAB. The type or the vector created in the MATLAB workspace is set by the type of `v` and the `as%` argument, the vector dimensions and sizes are set by the dimensions and sizes of `v[]`. All of the script language array types except strings are supported.

as% Optional. Sets the type of the MATLAB workspace variable. If `v` is a real value or array, `as%` can be 4 or 8 to specify single- or double-precision real data. If `v` is an integer value or array, `as%` can be 1, 2 or 4 for 1, 2 and 4-byte signed integer data and -1, -2 and -4 for the corresponding unsigned integer data. For 1 and 2-byte data only the bottom 1 and 2 bytes of the script variable are used and the upper bytes discarded. If `as%` is omitted, it is taken as 8 for a real value and 4 for an integer value.

For string data, a string variable is created in the workspace and `as%` is ignored.

For integer and real data, either a single variable or a vector is created in the MATLAB workspace. If a Signal script array is used then the dimensions of the vector created in the workspace match the script array: a 1-dimensional script array length n creates an n x 1 vector, a 2 dimensional script array `arr[m]` `[n]` creates an m x n vector and so forth.

Returns 0 if the data was successfully transferred and -1 if it was not.

**See also:**
`MatLabOpen(), MatLabClose(), MatLabGet(), MatLabEval(), MatLabShow()`

# MatLabShow()

This command retrieves the visible state of a MATLAB command window opened by `MatLabOpen()` and optionally changes it.

```
Func MatLabShow({show%});
```

show% Optional. If provided, sets the new visible state. Set 1 to show the window and 0 to hide it.

Returns the command window visibility at the time the call was made.

**See also:**
`MatLabOpen(), MatLabClose(), MatLabGet(), MatLabPut(), MatLabEval()`

# MATMul()

This function multiplies matrices and/or vectors. In matrix terms, this evaluates A = BC where A is an `m` rows by `n` columns matrix, B is an `m` by `p` matrix and C is a `p` by `n` matrix. Vectors of length `v` are treated as a `v` by 1 matrix.

```
Proc MATMul(a[]{[]}, const b[]{[]}, const c[]{[]});
```

a A `m` by `n` matrix of reals or a vector of length `m` (`n` is 1) to hold the result.

b A `m` by `p` matrix or a vector of length `m` (`p` is 1).

c A `p` by `n` matrix or a vector of length `p` (`n` must be 1).

If you pass any of `a`, `b` or `c` as a vector, they are treated as a `n` by 1 matrix, where `n` is the length of the vector. Use the `trans()` operator to convert a vector to a 1 by n matrix.

**See also:**
trans() operator, `ArrMul(), MATInv()`

# MATSolve()

This function solves the matrix equation **Ax** = **y** for **x**, given **A** and **y**. Both **x** and **y** are vectors of length `n` and **A** is an `n` by `n` matrix.

```
Func MATSolve(x[], const a[][], const y[]);
```

x A one dimensional real array of length `n` to hold the result.

a    A two dimensional (n by n) array of reals holding the matrix.

y    A one dimensional real array of length n.

Returns  The functions returns 0 if all is OK or -1 if a is a singular matrix.

**See also:**
ArrMul(), MATInv()

# MATTrace()

This evaluates the trace of a matrix, that is the sum of the diagonal of the matrix. If the matrix is not square, for example arr[i][j] and the smaller dimension is of size n, the trace is of the arr[:n][:n].

```
Func MATTrace(const arr[][]);
```

arr    A matrix (two-dimensional array) that is either square, or that is treated as square using the smaller dimension. The array can be either real or integer. The result is always a real value.

Returns  The trace of the matrix, that is the sum of the diagonal elements.

This is equivalent to ArrSum(diag(arr));

**See also:**
ArrSum(), MATTrans()

# MATTrans()

This transposes a matrix (a two dimensional array), swapping the rows and columns. This procedure physically moves the data, unlike the trans() or ` operator, which remaps the matrix without moving any data. It is usually much more efficient to use trans().

```
Proc MATTrans(mat[][]{, const src[][]});
```

mat    A m by n matrix returned holding the transpose of src. If src is omitted, m must be equal to n and the rows and columns of mat are swapped.

src    Optional, a n by m matrix to transpose.

**See also:**
trans() operator, ArrAdd(), MATMul()

# Max()

This function returns the index of the maximum value in an array, or the maximum of several real and/or integer variables.

```
Func Max(const arr[]|const 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()

This returns the maximum x axis value in the frame or a specified channel, or the latest time reached within the frame or the specified channel in a sampling document view. For the end of the visible x axis use XHigh().

```
Func MaxTime({chan%});
```

chan%   An optional channel number (1 to n). If present, and if the channel exists, the function gets the x axis value for the last item sampled in the channel or the maximum x axis value in the frame if no items are found on the channel, or if no channel was specified. If `chan%` is zero, the value returned is the frame length limit – the maximum X axis value for a frame regardless of the points that happen to be currently stored – this is useful for frame 0 of a file being sampled.

Returns   The value returned is the maximum x axis value for the frame or a specified channel. If the current view is of the wrong type, or if a specified channel number does not exist, the script stops with an error.

**See also:**
```
Len(), LastTime(), NextTime(), Mintime(), Seconds(), XHigh()
```

# 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 measurements settings dialog.

```
Func MeasureChan(chan%, name${, pts%});
```

chan%   This is 0 to add a new channel or the number of an existing channel to change settings. `MeasureToXY()` creates an XY view with one channel, so you will usually call this function with `chan%` set to 1. You can have up to 32 measurement channels in the XY view.

name$   This sets the name of the channel and can be up to 9 characters long.

pts%    Sets the maximum number of points for this channel, if omitted or set to zero then all points are used. When a points limit is set and more points are added, the oldest points are deleted.

Returns   The channel number these `settings` were applied to or a negative error code.

**See also:**
```
XY measurements, CursorActiveSet(), MeasureToXY(), MeasureX(), MeasureY()
```

# MeasureToChan()

This creates a new marker or real marker memory channel with an associated measurement process and cursor 0 iteration method, the channel is created in the current data view. The `ProcessFrames()` command adds items to the new channel based on active cursor measurements. Before calling this function, use `MeasureX()` to set the measurement method to 102 (Time) and set expr1$ to generate the time stamps of the generated marker data. If you select any other method this command will return an error code. The iteration must produce times in ascending order unless the output is to a memory channel.

If you are creating a real marker channel, you must call `MeasureY()` before this function to define the measurement to attach to each data item added to the new channel.

These commands implement the functionality of the Measurements to data channel dialog.

```
Func MeasureToChan(dest%, name$, type%, iter%, chan%, st|st$, en|en$, min|
exp$
      {, lv|lv$|pts%{ ,hy{, flgs%{, qu${, width{, lv2|lv2$}}}}}});
```

dest%   This is the output memory channel number or zero for the lowest numbered unused memory channel. Set to 201 to 400 for specific memory channels. It is a fatal error to set a channel that is in use or to specify a non-memory channel.

name$   The output channel name. Channel units, if required, are inferred from `chan%` in the `MeasureY()` call and the measurement method.

type%   This sets the output memory channel type, set 0 for a marker channel and 1 for a real marker channel. Only real marker channels with a single real value are generated.

iter%   This is the cursor 0 iteration mode. Modes are the same as in `CursorMode()` but not all modes can be used. Valid modes are:

| | | | |
|---|---|---|---|
| 4  Peak find | 11  Slope peak | 17  Turning point | 23  Data points |
| 5  Trough find | 12  Slope trough | 20  Expression | |

| 7 | Rising threshold | 14 | Slope +ve thr | 21 | Outside levels |
|---|---|---|---|---|---|
| 8 | Falling threshold | 15 | Slope -ve thr | 22 | Inside levels |

`chan%` This is the channel that is searched by the cursor 0 iterator. In expression mode (20), this is ignored and should be set to 0.

`st` This is the start position within the frame for cursor 0 iteration, either as a number or a string. In expression mode, this is the first iteration position, for other modes the start of the search.

`en` This is the end position within the frame for cursor 0 iteration, again either as a number or a string.

`min` This is the minimum allowed step for cursor 0, it is used in all modes except expression mode (20).

`exp$` This is the string expression that is evaluated in expression mode (20).

`lv` This number or string expression sets the threshold level for threshold modes (7, 8, 14, 15, 21 and 22). It is in y axis units for data channel `chan%` or y axis units per x axis unit for slope threshold modes (14 and 15). This argument is ignored and should be set to 0 or omitted for modes that do not require it.

`pts%` The number of points to advance by in Data points mode.

`hy` This sets the hysteresis level for threshold search modes (7, 8, 14, 15, 21 and 22) and the minimum amplitude for peak and trough modes (4, 5, 11 and 12). It is in y axis units for data channel `chan%` for normal modes or y axis units per x axis unit for slope modes. This argument is ignored and should be set to 0 or omitted for modes that do not require it.

`flgs%` This is the sum of the measurement option flags. Add 2 for user checks on the cursor positions and add 8 to overdraw the new channel on top of the source channel for the Y measurement (if the output channel type is real marker and the Y measurement type is appropriate). The default value is zero.

`qu$` This sets the qualification expression for the iteration. If left blank then all iteration positions will be used. If not blank, and it evaluates to non-zero, then the iteration is skipped.

`width` The width for measurements in X axis units; set to 0 or omit it in modes that do not require it. For slopes it sets the time over which the slope is measured. For peaks and troughs, it sets the maximum peak width (use 0 for no maximum). For threshold crossing modes (7, 8, 21 and 22) it sets the minimum crossing time (Delay in the dialog).

`lv2` This is a number or string expression, it sets the second threshold level for outside and inside dual threshold cursor iteration modes (21 and 22).

Returns The function result is a memory channel number or a negative error code.

Arguments passed as strings are not evaluated until data is processed. Invalid strings generate invalid measurements and no data points in the XY view.

### Example
This generates a channel holding peak values from channel 1 of the current data 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 ch%;                          'Memory channel number
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
ch%:=MeasureToChan(0, "Peaks", 1, 4, 1, 0.0, 1.0, 0.1, 1); 'Peak, chan 1
ProcessFrames(-1);                'Process all the data
```

### See also:
Measurements to data channel, CursorActiveSet(), MeasureX(), MeasureY(), ProcessFrames()

## MeasureToXY()

This creates a new XY view with a trend plot or measurement process and optional cursor 0 iteration method. It creates one output channel in the XY view with a default measurement method. Use `MeasureX()`, `MeasureY()` and `MeasureChan()` to edit the method and add channels. Use `ProcessFrames()` to generate the plot. The new XY view will be the current view and is invisible. Use `WindowVisible(1)` to make it visible. These commands implement the functionality of the Measurements to XY View and Measurements to XY View(Trend plot) dialogs.

```
Func MeasureToXY(iter%|tpf%{, chan%, st|st$, en|en$, min|exp$
      {, lv|lv$|pts%{ ,hy{, flgs%{, qu${, width{, lv2|lv2$}}}}}}});
```

iter%    This is the cursor 0 iteration mode. Modes are the same as in `CursorMode()` but not all modes can be used. Valid modes are:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 4 | Peak find | 11 | Slope peak | 17 | Turning point | 23 | Data points |
| 5 | Trough find | 12 | Slope trough | 20 | Expression | | |
| 7 | Rising threshold | 14 | Slope +ve thr | 21 | Outside levels | | |
| 8 | Falling threshold | 15 | Slope -ve thr | 22 | Inside levels | | |

tpf%     This is the only argument if you want trend-plot-style processing with one measurement per frame and no cursor zero iteration. It is the sum of the option flags for trend plot processing; add 1 for common X values, add 2 for user checks on the cursor positions or leave it set to the default value of zero.

chan%    This is the channel that is searched by the cursor 0 iterator. In expression mode (20), this is ignored and should be set to 0.

st       This is the start position within the frame for cursor 0 iteration, either as a number or a string. In expression mode, this is the first iteration position, for other modes the start of the search.

en       This is the end position within the frame for cursor 0 iteration, again either as a number or a string.

min      This is the minimum allowed step for cursor 0, it is used in all modes except expression mode (20).

exp$     This is the string expression that is evaluated in expression mode (20).

lv       This number or string expression sets the threshold level for threshold modes (7, 8, 14, 15, 21 and 22). It is in y axis units for data channel `chan%` or y axis units per x axis unit for slope threshold modes (14 and 15). This argument is ignored and should be set to 0 or omitted for modes that do not require it.

pts%     The number of points to advance by in Data points mode.

hy       This sets the hysteresis level for threshold search modes (7, 8, 14, 15, 21 and 22) and the minimum amplitude for peak and trough modes (4, 5, 11 and 12). It is in y axis units for data channel `chan%` for normal modes or y axis units per x axis unit for slope modes. This argument is ignored and should be set to 0 or omitted for modes that do not require it.

flgs%    This is the sum of the measurement option flags. Add 1 to force common X values, add 2 for user checks on the cursor positions and add 4 to generate one averaged measurement per frame. The default value is zero.

qu$      This sets the qualification expression for the iteration. If left blank then all iteration positions will be used. If not blank, and it evaluates to non-zero, then the iteration is skipped.

width    The width for measurements in X axis units; set to 0 or omit it in modes that do not require it. For slopes it sets the time over which the slope is measured. For peaks and troughs, it sets the maximum peak width (use 0 for no maximum). For threshold crossing modes (7, 8, 21 and 22) it sets the minimum crossing time (Delay in the dialog).

lv2      This is a number or string expression, it sets the second threshold level for outside and inside dual threshold cursor iteration modes (21 and 22).

Returns  The function result is an XY view handle or a negative error code.

Arguments passed as strings are not evaluated until data is processed. Invalid strings generate invalid measurements and no data points in the XY view.

### Example
This generates a plot of peak values in channel 1 of the current data 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, 0.0, 1.0, 0.1, 1); 'Peak, chan 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
ProcessFrames(-1);               'Process all the data
```

# MeasureX()

MeasureX() and MeasureY() set the x and y part of a measurement. 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 measurements settings dialog. The current view must be the target of the measurements.

```
Func MeasureX(type%{, chan%{, expr1|coef%{, expr2{, width}}}});
```

type%    This sets the x or y measurement type. Types less than 100 are cursor regions measurements matching the ChanMeasure() function:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | Curve area | 6 | Modulus | 11 | Standard deviation | 16 | Standard error |
| 2 | Mean | 7 | Maximum | 12 | Extreme | 17 | RMS error |
| 3 | Slope | 8 | Minimum | 13 | Peak | | |
| 4 | Area | 9 | Peak to peak | 14 | Trough | | |
| 5 | Sum | 10 | RMS amplitude | 15 | Point count | | |

Types from 100 up are special values:

| | | | | | |
|---|---|---|---|---|---|
| 100 | Value at point | 105 | Frame abs time | 110 | Value product |
| 101 | Value difference | 106 | Frame state value | 111 | Value above baseline |
| 102 | Time at point | 107 | 0-based fit coefficient | 112 | Iteration count |
| 103 | Time difference | 108 | User entered value | 113 | Expression |
| 104 | Frame number | 109 | Value ratio | | |

Types from 1000 up select frame variable values with the frame variable number being type%-1000.

chan%    This is the channel number from which the measurements are taken. For time, user entered and frame-based measurements it is ignored and should be set to 0 or omitted.

expr1    Either a real value or a string expression that sets the start time for measurements over a time range, the position for time (102) and value measurements and the expression used for measurement type 106.

coef%    The zero-based fit coefficient number for measurement type 107.

expr2    Either a real value or a string expression that sets the end time for measurements over a time range and the reference time for single-point measurements and differences. Set an empty string when width is required and this is not.

width    This is the measurement width for value and value difference measurements. The default value is zero.

Returns  The function return value is zero or a negative error code.

# 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}});
```

# MemChan()

This function creates a new memory channel attached to the file in the current data view. You can have up to 200 memory channels per file. There are two variants: the first lets you specify the channel type and settings, the second copies settings from an existing channel.

```
Func MemChan(type%{, size%});
Func MemChan(-1, copy%);
```

type%   The type of memory channel to create. The available types are:
1   Marker
5   Idealised trace
6   Real marker

size%   For real marker channels it sets the number of reals to attach to each item, if omitted the default value of 1 is used.

copy%   A channel from which to copy settings, including the channel type, title, units and size. The channel must exist.

Returns The new memory channel number, or 0 if there is no free channel, or a negative error code.

Channels created by the first variant have default titles and units. You can set these with ChanTitle$() and ChanUnits$(). This example code copies a channel of any type:

```
func CopyWave%(chan%)
var mc%;
mc% := MemChan(-1, chan%);              ' Create channel of the same type
if mc%>0 then                           ' Created OK?
   var f%;
   for f% := 1 to FrameCount() do       ' Work through the frames
      Frame(f%);                        ' Select the frame
      MemImport(mc%, chan%, MinTime(), MaxTime()); ' Copy data into memory channel
   next;
   ChanShow(mc%);                       ' Display new channel
endif;
return mc%;                             ' Return the new memory channel number
end;
```

**See also:**
MemDeleteItem(), MemDeleteTime(), MemGetItem(), MemImport(), MemSetItem(), MarkInfo(), ChanIndex()

# MemDeleteItem()

This function deletes one or more items from a memory channel created by MemChan() or by other means. To delete the actual channel use ChanDelete().

```
Func MemDeleteItem(chan% {,index% {,num%}});
```

chan%   The channel number of a channel created by MemChan() or by other means, including normal channels in memory frames. This channel should hold marker or real marker data; this function does not operate on waveform or idealised trace channels.

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(), MemSetItem()

# MemDeleteTime()

This function deletes one or more items from a memory channel based on an X axis range.

```
Func MemDeleteTime(chan%, mode%, t1 {,t2});
```

chan%   The channel number of a channel created by `MemChan()` or by other means, including normal channels in memory frames. This channel should hold marker or real marker data; this function does not operate on waveform or idealised trace channels.

mode%   This sets the items to delete and how to interpret the values `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`.

t1, t2   Two times, in seconds, that set the X axis (normally time) range for items to delete. Times outside the frame X axis range are limited to the range.

Returns   The number of items deleted, or a negative error code.

**See also:**
MemChan(), MemDeleteItem(), MemGetItem(), MemImport(), MemSetItem()

# MemGetItem()

This returns information about a memory channel item. The item is identified by its ordinal position in the channel, not by time.

```
Func MemGetItem(chan% {,index% {,code%[]|&code% {,data[]|&data}}});
```

chan%   The channel number of a channel created by `MemChan()` or by other means, including normal channels in memory frames. This channel should hold marker or real marker data; this function does not operate on waveform or idealised trace channels.

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 that is returned holding the item marker codes in the first four values, or a single integer variable that is returned holding the first code byte value. For an array, the codes returned is the smaller of the size of the array and four.

data   This is a real array that is returned holding the real marker data from the item or a single real variable that is returned holding the first real marker value. If the channel data is not real marker, 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.

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 a value less than `MinTime()`. Otherwise it returns the time of the item.

**See also:**
MemChan(), MemDeleteItem(), MemDeleteTime(), MemImport(), MemSetItem()

# MemImport()

This function imports data into a memory channel created by `MemChan()` or by other means. 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 |
|---|---|---|
| Marker | Waveform Marker Real marker All others | Can extract marker times, coded for peak/trough, etc. Marker times and codes copied Marker times and codes copied Not available |
| Real marker | Waveform Marker Real marker All others | Can extract marker times, coded for peak/trough, etc., real values set to zero Marker times and codes copied, real values set to zero Marker times, codes and real values copied, real data truncated or zero padded as needed Not available |
| Idealised trace | Idealised trace All others | Idealised trace data copied Not available |

You can extract markers and real markers from waveform data using peak search and level crossing techniques.

### Import compatible channel

The first command variant imports data from a compatible channel within the specified time range and frames. All marker-type 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. Idealised trace data is compatible with itself.

```
Func MemImport(chan%, inCh%, start, end, frm%|frm%[]|frm$);
```

`chan%`   The channel number of a channel created by `MemChan()` or by other means, including normal channels in memory frames.

`inCh%`   The channel number to import data from.

`start`   The start time (X axis value) to collect data from.

`end`   The end time (X axis value) to collect data up to (and including).

`frm%`   Frame number or a negative code as follows:

-1   All frames in the file.

-2   The current frame.

-3   Only tagged frames.

-6   Only untagged frames.

`frm$`   A frame specification string. This option specifies a list of frames using a string such as `"1..32,40,50"`.

`frm%[]`   An array of frame numbers. This option provides a list of frame numbers. The first element holds the number of frames in the list.

### Extracting markers from waveforms

The extra `mode%`, `time`, `level` and `code%` arguments are used when extracting markers or real markers from waveform data. `inCh%` must be a waveform channel. 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.

```
Func MemImport(chan%, inCh%, start, end, frm%|frm%[]|frm${ ,mode%, time,
level{, code%}});
```

mode%  The mode of data extraction. The modes are:

    0   Extract markers based on the time of a peak in the waveform, these are coded as 2 unless `code%` is set.

    1   Extract markers based on the time of a trough in the waveform, these are coded  3 unless `code%` is set.

    2   Extract markers based on times when the waveform rises through level, these are coded 4 unless `code%` is set.

    3   Extract markers based on times when the waveform falls through level, these are coded 5 unless `code%` is set.

time  The minimum time between detected markers. Use this to filter noisy signals.

level  In modes 0 and 1, this is the minimum peak or trough amplitude - 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 markers.

code%  If present and positive it overrides the codes based on `mode%` that are applied to the markers. The low byte of `code%` sets the first marker code; the remaining marker codes are always 0.

Returns  It returns the number of items added to the channel (zero if no items were added), or a negative error code.

**See also:**
```
MemChan(), MemDeleteItem(), MemDeleteTime(), MemGetItem(), MemSetItem()
```

# MemSetItem()

This function edits or adds an item in a memory channel. The item is identified by its ordinal position in the channel.

```
Func MemSetItem(chan%, index%, time{, code%[]|code%{, data[]|data}});
```

chan%  The channel number of a channel created by `MemChan()` or by other means, including normal channels in memory frames. This channel should hold marker or real marker data; this function does not operate on waveform or idealised trace channels.

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`.

time  The item time, or less that `MinTime()` for no change. If index% is 0, you must supply a valid time between `MinTime()` and `MaxTime()`.

code%  This is an integer array that holds the marker codes for the channel data item or a single integer value that holds the first marker code for the item. marker code values can be from 0 to 255. If the array is less than four long then upper code bytes are left unchanged.

data  This is a real array holding the item data for real marker items or a single real value holding the first real marker value. If the channel data type is not real marker it is ignored. The number of points set is the smaller of the number passed and the number expected. If the array is too short, the extra values are unchanged when `index%` > 0 (editing) and have the value 0 if `index%` is 0.

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.

**Example**
This example creates an array of markers at time 1, 2, 3...10 seconds. It assumes that the current view is a data view; if not the `MakeMarkChan%()` function will return -1.

```
'Create a new memory event channel holding the times passed in an array.
'times       is an array of times to add to a new memory event channel
'returns     event channel or 0 if no data or -ve if an error
func MakeMarkChan%(times[])
```

```
var mc%, n%, i%;
n% := Len(times[]);                   ' number of points to add
if n% <= 0 then return 0 endif;       ' no data
mc% := MemChan(1);                    ' create a new memory channel
if (mc% < 0) then return -1 endif;    ' failed to create
for i% := 0 to n%-1 do
    MemSetItem(mc%, 0, times[i%]);
next;
return mc%;
end;

var times[10], memChan%;
ArrConst(times, 1);ArrIntgl(times); ' create 1,2,3..10
memChan% := MakeMarkChan%(times);   ' make hidden marker channel
if (memChan% > 0) then
    ChanShow(memChan%);             ' show hidden channel
endif;
```

**See also:**
MemChan(), MemDeleteItem(), MemDeleteTime(), MemGetItem(), MemImport(), MinTime(), MaxTime()

# 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${, const arg1{, const arg2...}});**

form$   A string that defines the output format as for Print(). If the string includes a vertical bar, the text before the vertical bar is used as the window title.

arg1,2 The arguments used to replace %d, %f and %s type formats.

You can split the message into multiple lines by including \n in the form$ string. Long messages are truncated.

**See also:**
Print(), Input(), Query(), DlgCreate(), DlgFont()

# 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$(), ReadStr(), Right$(), 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(const arr[]|const 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.

An example finding the minimum in a sub-array holding 10 items of the original data:

```
var data[70], minPos%, minVal;
...
minPos:=Min(data[40:10]);  ' returns a position between 0 and 9
minVal:=data[40+minPos];   ' value of minimum
```

**See also:**
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 data view channels with a y axis, or the minimum and maximum intervals for a marker channel handled as dots or lines. The values returned for marker channels as a rate histogram are measured from the histogram with partial bins included.

**Func Minmax(chan%, start, finish, &min, &max{, &minP{, &maxP {, mode%{, binSz}}}});**

chan%   The channel number (1 to n) on which to find the maximum and minimum.

start   The start position in x axis units.

finish  The end position in x axis units.

min     The minimum value is returned in this variable or zero if no data found.

max     The maximum value is returned in this variable or zero if no data found.

minP    The position of the minimum is returned in this variable or zero if no data found.

maxP    The position of the maximum is returned in this variable or zero if no data found.

mode%   This will have no effect for a waveform channel. If present for a marker channel, this sets the effective drawing mode in which to find the minimum and maximum. If mode% is absent or inappropriate, the current display mode is used. The modes are:

    0    The current mode for the channel. Any additional arguments are ignored.

    1    Dots mode for markers, returns the position of the marker at or after pos.

    2    Lines mode for markers, result is the same as mode 1.

    3    Rate mode for markers. The binSz argument sets the width of each bin.

    9    Plot mode for real marker channels only, using the currently selected channel data index selected for that channel.

binSz   This sets the width of the rate histogram bins when specifying rate mode.

Returns  1 if data points were found, 0 if no data was found or a negative error code.

**See also:**
Min(), Max(), XYRange(), ChanIndex()

# Mintime()

In a data view, this returns the minimum x axis value in the frame or in a channel. For the end of the visible x axis use XLow().

**Func MinTime({chan%});**

chan%   An optional channel number (1 to n). If present, and if the channel exists, the function gets the x axis value for the earliest item in the channel or the minimum x axis value in the frame if no items are found on the channel or if no channel was specified. If chan% is zero, the value returned is the frame length limit – the minimum X axis value for a frame regardless of the points that happen to be currently stored – this is useful for frame 0 of a file being sampled.

Returns  The value returned is the minimum x axis value in the frame or channel. If the current view is of the wrong type, or if the channel number is illegal the script stops with an error.

**See also:**
```
Len(), BinZero(), ChanRange(), LastTime(), NextTime(), Maxtime(), Seconds(), XLow()
```

# Modified()

This command lets you get (and in many cases, set) the modified state of a view and detect if it is read only. Beware that clearing the modified flag for views that support this will allow you to close the view without being prompted to save changes. This function was added to Signal at version 4.07 and its behaviour adjusted in version 6.03.

```
Func Modified({what%{, new%}});
```

what%    0 (or omitted) to get or set the modified state, 1 to get or set the read only state.

new%     The new state. -1 (or omitted) for no change, 0 to clear the state, 1 to set the state.

Returns  The state at the time of the call, before any change.

The meaning and effect of this routine depends on the type of the current view.

### Text view
A text view is considered modified if changes have been made since the last save; such changes will be interactive edits, changes made by a script and typing. You can use `Modified(0,0)` to clear the modified state without saving the file; you cannot set the modified flag with this command.

Text views (but not the Log view) can be set read only with `Modified(1,1)` and the read only state can be cleared with `Modified(1,0)`. Note that output sequence and script files open read only if they are marked read only on disk. This command does not change the read only status of the file on disk.

### Data view
A data view counts as modified if you make a change to it that would be written to the underlying `.cfs` file. Changes made to virtual channels or to idealised traces do not count as modifications, but changes made (in the current frame) to the frame tag, flags, state or user variables do - however note that such changes made to other frames are written to disk immediately and do not set the modified flag, while changes to the current frame data are only written when the file is closed or the data view switches to a different frame. You can force the file to commit any changes written to disk, update the file headers and ensure the directory information is up to date with `Modified(0,0)`. Committing changes in this way does not clear the modified flag. It also does not write any changes made to the current frame data or any appended frames to disk, this should be done using `FrameSave()` and `FileSave()` respectively.

`Modified(1)` reports the read only state. You cannot change the read only state of a data view; it depends on the read only state of the underlying `.cfs` file.

### Memory, Grid and XY views
A memory, grid or XY view is modified if it has been changed since the last save. You can set and clear the modified flag using `Modified(0, new%)`, note that this changes the flag but does not save any changed data to disk. There is currently no concept of a read only state for these views and `Modified(1)` always returns 0.

### Other view types
The command is not implemented for other view types and will return 0.

**See also:**
```
FrameSave(), FileSave(), FrameFlag(), FrameState(), FrameTag(), FrameUserVar()
```

# MousePointer()

This command loads permanent mouse pointers from external files, creates new temporary mouse pointers and can delete temporary mouse pointers when they are no longer needed. Signal maintains a list of standard mouse pointers (see `ToolbarMouse()` for the list); any pointers added by this function are added to this list and are available for use by the mouse `Down%()`, `Move%()` and `Up%()` routines associated with `ToolbarMouse()`.

```
Func MousePointer(text$|nDel%);
```

text$    This is a text string that defines a new mouse pointer. This is either the path to a data file that holds a cursor or an animated cursor, ending in ".cur" or ".ani" (not case sensitive) or it is a text string that defines a monochrome mouse pointer, as described below. The function returns the number of the new mouse pointer, or 0 if it was not created. Cursors loaded from a file are permanent and exist until Signal closes. There is a limit on the number of cursors that Signal will manage (currently set to 60); this should be more than enough for any reasonable purpose.

nDel%    The number of a mouse pointer to delete, or -1 to delete all user-defined mouse pointers. You can only delete temporary mouse pointers, and you cannot delete a mouse pointer that is currently in use.

Returns    The number of a newly created mouse pointer or 0 if it was not created or the number of mouse pointers that were deleted.

**Text format to create a new pointer**

Mouse pointers defined by this function are a 32 x 32 pixel image. Inside this image is the hot spot, being the actual position where the mouse is deemed to be pointing. Each pixel in the image can be screen coloured, black, white or the inverse of the screen. In addition, you can designate a pixel to be the hot spot. This is coded in text as follows:

| Character | Screen | Black | White | Inverse |
|---|---|---|---|---|
| Normal | space | b | w | i |
| Hot-spot | h | B | W | I |

Cursors are defined by pixel by pixel, starting at the top left, moving horizontally and starting a new row every 32 characters. However, most mouse pointers are much smaller than 32 x 32 pixels, so you can stop a line early by adding a vertical bar character "|" or by a line feed character "\n". Any character that is not a space, b, B, w, W, h, i, I, | or line feed is treated as a space. You do not need to provide 32 rows; any rows that are not defined are treated as if they were filled with spaces. It is not an error to include more than one hot-spot character; the hot-spot position is set by the last hot-spot character in the string.

The following example creates a small square cursor:

```
var mp% := MousePointer(
"bbbbbbb|"
"bwwwwwb|"
"bw   wb|"
"bw h wb|"
"bw   wb|"
"bwwwwwb|"
"bbbbbbb");
```

This could be written as:

```
var mp% := MousePointer("bbbbbbb|bwwwwwb|bw i wb|bwihiwb|bw i wb|bwwwwwb|bbbbbbb");
```

but the first arrangement is much easier to understand.

**Loading mouse pointers from files**

Cursors created with text strings are monochrome and not animated. You can also load coloured (sometimes called 3D) cursors and animated cursors from files. If you want to experiment with this, you can find suitable files in the Windows Cursors folder. On my machine, the following loads an animated stopwatch cursor:

```
var ani% := MousePointer("c:\\WINDOWS\\Cursors\\stopwtch.ani");
```

**See also:**
```
ToolbarMouse()
```

# MoveBy()

This command can be used in both Text views and Grid views to move relative to the current position and to report the current position without moving..

### Text view

This gets and sets the position of the text caret. You can move the text caret in a text window relative to the current position by lines and/or a character offset and you can extend or cancel the current selection. The caret position can be read as a position in the entire document or as a line and a column. If you use this to move past the end of a line, beware that in Windows, this is usually marked by two characters (`\r`, `\n`, CR LF).

```
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. You can use the `Draw()` command to position a text view to start at a given 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. You can use `XLow()` and `XHigh()` to get the first visible line and the line past the last fully visible line.

### *Unicode*

When you use a Unicode build of Signal, the text editor works in terms of byte positions in the text, not in terms of Unicode characters. If your text only contains ASCII characters with codes in the range 1 to 127 (0x01 to 0x7f), then positions in the text correspond with displayed characters. However, if your text holds characters with higher codes, each character may occupy from 1 to 4 bytes. We do not allow you to position the text caret in the middle of Unicode characters as then if you used `Selection$()`, the result could be rubbish. So the rules are that if you move the caret and the move ends in the middle of a multi-byte character, then the move continues in the same direction until the caret lies between Unicode characters. This means that character moves by 1 or -1 will always move the text caret by 1 Unicode character, as you would expect, though the position may change by more than 1.

### Grid view

In a grid view, this command is used to select cells in the grid relative to the last position. There is (currently) no script concept of selecting text ranges within a cell. The grid allows selection of multiple rectangular regions.

```
Func MoveBy(sel%{, col%{, row%}});
```

sel%       If both `col%` and `row%` are omitted and this is 1, no change is made and the current row is returned. If this is 2, the current column is returned. If this is 0, all selections are cleared.

           If `col%` is present, `col%` (and `row%` if present) are added to the current column and row to generate a new position. If `sel%` is 0, all selections are cleared and the `row%` and `col%` start a new rectangular selection. If 1, the current selection is extended to the cell set by `row%` and `col%`. If 2, a new selection is started and added to any existing selection.

col%       Optional. This is added to the last column position. The result is limited to the range 0 to the number of columns -1.

row%       Optional, taken as 0 if omitted. This is added to the last column position. The result is limited to the range 0 to the number of rows-1.

Returns    If `col%` is present, or if `sel%` is 0, the return value is the cell number that we moved to. This is calculated as the row number times the number of columns plus the column number.If `col%` is omitted, `sel%` = 0 returns the current cell number, `sel%` = 1 returns the current column number and `sel%` = 2 returns the current row number.

### See also:

```
Draw(),EditSelectAll(), MoveTo(), Selection$(), XHigh(), XLow()
```

# MoveTo()

This command can be used in both Text views and Grid views.

### Text view

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. See `MoveBy()` to get the caret position as a line or column.

**`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). You can use the `Draw()` command to position a text view to start at a given line.

Returns  The function returns the new position in the file. You can use `XLow()` and `XHigh()` to get the first visible line and the line past the last fully visible line.

*Unicode*

In a Unicode build of Signal, you can only position the caret between characters.

### Grid view

In a grid view, this command is used to select cells in the grid. There is (currently) no script concept of selecting text ranges within a cell. The grid allows selection of multiple rectangular regions.

**`Func MoveTo(sel%, col%, row%);`**

sel%     If 0, all selections are cleared and the `row%` and `col%` start a new rectangular selection. If 1, the current selection is extended to the cell set by `row%` and `col%`. If 2, a new selection is started and added to any existing selection.

col%     If set negative, the entire row set by `row%` is selected unless `row%` is too large when the command is ignored. If this is positive, it sets the column unless it is too large, when it is limited to the available columns.

row%     If set negative, the entire column set by `col%` is selected (equivalent to selecting from (0, `col%`) to (maxRow-1, `col%`) unless col% is too large, when the command is ignored. If this is positive, it sets the row (unless it is greater than the number of rows when it is limited to the maximum value.

To select the entire grid, use `EditSelectAll()` or set both `col%` and `row%` to -1.

### See also:
```
Draw(),EditFind(),EditSelectAll(),MoveBy(),Selection$(),XLow(),XHigh()
```

# N

# NextTime()

This function is used to find the next item on a channel after a particular x axis position.

**`Func NextTime(chan%, &pos{, &val|&code%|code%[]{, &rval|rval[]}});`**

chan%    The channel number (1 to n) to use for the search.

pos      The x axis position to start the search after. Items at the start position are ignored. To ensure that items at the `Mintime()` are found, set position to `Mintime()-1`. pos is updated to contain the x axis position of the next item. It is left unchanged if no more items are found or there is an error.

val     This optional argument is used with waveform channels. It is returned holding the waveform value.

code%   This optional parameter is only used if the channel is a marker type, it can be an array that is filled in with the marker code values or a single integer variable that is updated with the first code value. Each marker has four code values, the data returned in `code%[]` is the lesser of 4 and the array size.

rval    This optional parameter is only used if the channel is real marker type, it is either an array that is filled in with the real marker float values or a single real variable that is updated with the first real marker float value. Real markers can have one or more float values, the data returned in `rval[]` is the lesser of the data available and the array size.

Returns The function returns 1 if a data item is found, 0 if there are no more items to be found, or a negative error code.

**See also:**
`LastTime(), MaxTime(), MinTime()`

# O

# OpClEventGet()

This function gets the details of a particular event in an idealised trace.

```
Func OpClEventGet(chan%, meth%, &time{, &per{, &amp{, &flags%}}});
```

chan%   The channel number of the idealised trace.

meth%   The indexing method used to determine which event we are addressing. Possible values are:
    0    Use the selected event and ignore the `time` parameter.
    1    Find the first event starting before the specified time or the first event in the trace if none exist before `time`.
    2    Find the event with a start time closest to the specified time.
    3    Find the first event starting after the specified time.

time    The time used to address the event. This will be set to the start time of the event if an event is found.

per     This will be set to the duration of the event if found.

amp     This will be set to the amplitude of the event if found.

flags%  This will contain the events flags if found. A full list of flags can be found in the description of `SetOpClHist()`.

Returns The function returns 1 if an event was found otherwise it returns 0.

**See also:**
`OpClEventSet() OpClEventDelete() OpClEventSplit() SetOpClHist()`

# OpClEventChop()

This function splits the specified event in an idealised trace into two, each having a period equal to half that of the original. If the event has an amplitude between those of the preceding and following events then the amplitudes and flags of the first and second new events will be taken from the following and preceding events respectively. In this case the new events will also be flagged as having assumed amplitudes.

```
Func OpClEventChop(chan%, meth%{, time{, opt%}});
```

chan%   The channel number of the idealised trace.

meth%   The indexing method used to determine which event we are addressing. Possible values are:
    0    Use the selected event and ignore the `time` parameter.
    1    Find the first event starting before the specified time or the first event in the trace if none exist before `time`.
    2    Find the event with a start time closest to the specified time.
    3    Find the first event starting after the specified time.

time     The time used to address the event (not needed if `meth%` is 0).

opt%     Omit this or set it to zero to split the entire event in half, set it to 1 to split the event half-way through the visible portion of the event (to match the interactive behaviour).

Returns  The function returns 1 if an event is found otherwise it returns 0.

**See also:**
`OpClEventDelete(), OpClEventGet(), OpClEventMerge(), OpClEventSet(), OpClEventSplit()`

# OpClEventDelete()

This function deletes a specified event from an idealised trace and amalgamates its neighbours to produce a single event covering the time range of all three events and an amplitude calculated as an average of the amplitudes of all three events weighted by their durations. The flags for this event will be taken from the earliest of the three original events.

`Func OpClEventDelete(chan%, meth%{, time{, opt%}});`

chan%    The channel number of the idealised trace.

meth%    The indexing method used to determine which event we are addressing. Possible values are:
  0    Use the selected event and ignore the `time` parameter.
  1    Find the first event starting before the specified time or the first event in the trace if none exist before `time`.
  2    Find the event with a start time closest to the specified time.
  3    Find the first event starting after the specified time.

time     The time used to address the event (not needed if `meth%` is 0).

opt%     Omit this or set it to zero to always delete the event, set it to 1 to only delete if some portion of the event is visible (to match the interactive behaviour).

Returns  The function returns 1 if an event was found and deleted, otherwise it returns 0.

**See also:**
`OpClEventGet(), OpClEventSet(), OpClEventSplit()`

# OpClEventMerge()

This function amalgamates the specified event with the following event to produce a single event covering the time range of both events and an amplitude calculated as an average of the amplitudes of both events weighted by their durations. The flags for this event will be taken from the original specified event.

`Func OpClEventMerge(chan%, meth%{, time{, opt%}});`

chan%    The channel number of the idealised trace.

meth%    The indexing method used to determine which event we are addressing. Possible values are:
  0    Use the selected event and ignore the `time` parameter.
  1    Find the first event starting before the specified time or the first event in the trace if none exist before `time`.
  2    Find the event with a start time closest to the specified time.
  3    Find the first event starting after the specified time.

time     The time used to address the event (not needed if `meth%` is 0).

opt%     Omit this or set it to zero to always merge the events, set it to 1 to only merge if some portion of the event is visible (to match the interactive behaviour).

Returns  The function returns 1 if an event was found and merged, otherwise it returns 0.

**See also:**
`OpClEventChop(), OpClEventDelete(), OpClEventGet(), OpClEventSet(), OpClEventSplit()`

# OpClEventSet()

This function sets the details of a particular event in an idealised trace. Neighbouring events will be adjusted to accommodate the new values by altering the start time or duration acordingly. If you attempt to modify an event beyond the time range of the immediate neighbours then the function will fail and 0 will be returned.

```
Func OpClEventSet(chan%, meth%, time, start, period{, amp{, flags%}});
```

chan%   The channel number of the idealised trace.

meth%   The indexing method used to determine which event we are addressing. Possible values are:
- 0    Use the selected event and ignore the time parameter.
- 1    Find the first event starting before the specified time or the first event in the trace if none exist before time.
- 2    Find the event with a start time closest to the specified time.
- 3    Find the first event starting after the specified time.

time    The time used to address the event.

start   The new start time for the event being indexed.

period  The new duration for the event.

amp     The new amplitude for the event. If omited the amplitude will be left unchanged.

flags%  The new flag values. If omitted the flags are left unchanged. A full list of flags can be found in the description of SetOpClHist().

Returns The function returns 1 if an event was found and set, otherwise it returns 0.

**See also:**
OpClEventGet(), OpClEventDelete(), OpClEventSplit(), SetOpClHist()

# OpClEventSplit()

This function splits the specified event in an idealised trace into three, each having a period equal to one third that of the original.

```
Func OpClEventSplit(chan%, meth%{, time{, opt%}});
```

chan%   The channel number of the idealised trace.

meth%   The indexing method used to determine which event we are addressing. Possible values are:
- 0    Use the selected event and ignore the time parameter.
- 1    Find the first event starting before the specified time or the first event in the trace if none exist before time.
- 2    Find the event with a start time closest to the specified time.
- 3    Find the first event starting after the specified time.

time    The time used to address the event (not needed if meth% is 0).

opt%    Omit this or set it to zero to split the entire event into three, set it to 1 to split only the visible portion of the event (to match the interactive behaviour).

Returns The function returns 1 if a data item is found, 0 if there are no more items to be found, or a negative error code.

**See also:**
OpClEventGet(), OpClEventSet(), OpClEventDelete()

# OpClFitRange()

This function fits an idealised trace so that the convolution with the step response function of the filter used to sample the original data overlays the observed raw data trace.

```
Func OpClFitRange(chan%, start, end);
```

chan%   The channel number of the idealised trace.

start   The start time of the range to fit.

end     The end time of the range to fit

Returns The function returns 1 if a data item is found, 0 if there are no more items to be found, or a negative error code.

**See also:**
`OpClEventChop(), OpClEventDelete(), OpClEventGet(), OpClEventMerge(), OpClEventSet()`

# OpClNoise()

This function is used to measure an area of baseline and obtain measurements of the RMS noise and RMS noise in the first derivative. You will need to call this function before you use SetOpClScan(); in addition to returning these values it saves information internally for use by the SCAN analysis.

```
Proc OpClNoise(chan%, st, end{, &base{, &rms {, &rmsDrv}}});
```

chan%   The channel to measure

st      The start time of the area to measure

end     The end time of the area to measure

base    Returned holding the mean data value within the time range

rms     Returned holding the standard deviation of the data within the time range from the mean

rmsDrv  Returned as the root mean square of the first derivative of the data within the time range

**See also:**
SetOpClScan()

# Optimise()

This has the same effect as the optimise button in the YAxis dialog and can be used in a data or XY view. Optimising a channel that is not displayed is not an error. If you give a channel number that is not displayed, we assume that you know what you are doing, so it is optimised in the display mode that would be used if the channel were turned on.

```
Proc Optimise(cSpc{, start{, finish}});
```

cSpc    A channel specifier for the channels to optimise.

start   The start of the region to optimise. If omitted, this is the start of the window.

finish  The end of the region to optimise. If omitted, this is the end of the window.

**See also:**
`YRange(), YLow(), YHigh(), MinMax(), XYRange()`

# OutputReset()

This command is equivalent to the Output Reset and Application Output Reset dialogs. It allows you to specify DAC and digital output levels to be set before and after sampling. There are two command versions: the first sets values and the second returns the set values.

```
Func OutputReset(flags%, dacs%[], dacv[], dig%[]{, rampt{, n1401%}});
Func OutputReset(&dacs%[], &dacv[], &dig%[]{, &rampt{, n1401%}})
```

flags%  When to apply the settings. This is the sum of: 1 = at configuration load/program start, 2 = before sampling, 4 = after sampling.

dacs%   An array of up to 8 elements. Element n corresponds to DAC n. When setting values, set each element to 1 to apply the associated DAC value, 0 to not apply the associated value. Values of 0 and 1 are returned when reading back values.

dacv   An array of up to 8 DAC values in Volts to apply if the corresponding element of dacs%[] is not zero.

dig%   An array of up to 16 elements, element n corresponding to digital output bit n. Set values as: 1=high, 0=low, -1 no change. Elements 0-7 correspond with the output sequencer DIGLOW command, elements 8-15 correspond with the DIGOUT command.

rampt   Ramp time in seconds, default 0. This relates to a currently unimplemented feature. It will allow you to specify how long to take to ramp the DAC outputs to their final values for use in situations where a sudden DAC change could cause a problem.

n1401%   Currently, set this to 0 to set/get values in the Output Resets dialog (sampling configuration) and 1024 for the Application Output Resets dialog (application preferences). We have plans to allow sampling with multiple 1401s. When this is enabled you will add the 1401 number to n1401%. If this is omitted, this value is taken as 0, meaning set the value in the current sampling configuration.

Return   Both function versions return the flags% value at the time of the call.

The dacs%, dacv and dig% arrays can be any length without error. If longer arrays are supplied, the extra values are ignored. If shorter arrays are supplied, only the elements present are used. For the supplied values to have any effect, at least one of the flags% values must be set and the desired output must be enabled with dacs% or dig%.

This command was added to Signal at version 5.08.

# Overdraw()

This function turns overdraw mode on and off for the current view. It also returns the current overdraw mode. With overdraw mode on, a view will display not only the current frame, but also all of the other frames in the overdraw frame list.

```
Func Overdraw({mode%{, cols%{, maxf%{, maxt}}}});
Func Overdraw({get%});
```

mode%   0 for off, 1 for 2D, 3 for 3D (2 acts as 0).

cols%   The colour selection option, this is the same options as for interactive control:

   0   Draw all overdrawn (non-current) frames using the Overdraw colour defined in the colour settings.
   1   Draw overdrawn frames using the Signal colour cycling table. Each frame is drawn using the next colour in the table, wrapping round when the end of the table is reached.
   2   Draw overdrawn frames using a colour halfway between the channel trace colour and the channel background.
   3   Draw overdrawn frames using the same colour as the current frame – the standard trace colour.
   4   Draw overdrawn frames in a colour that fades from the trace colour to the channel background colour.
   5   Draw overdrawn frames in a colour that fades from the trace colour to the defined overdraw colour.
   6   Draw overdrawn frames in a colour that fades from the trace colour to the channel secondary colour.
   7   Draw overdrawn frames using a colour taken from the colour cycling table, each frame is drawn using a table colour indexed by the frame state, wrapping round the end of the table as necessary

maxf%   The maximum number of frames to overdraw. Set to 0 or omit for no limit.

maxt   The maximum time range. Set to 0 or omit for no limit.

get%   Alters the meaning of the return value. Possible values are:

   -1 or omitted to get current mode%
   -2 to get current cols%
   -3 to get current maxf%
   -4 to get current maxt

Returns   By default, the state of overdraw mode at the time of the call, or a negative error code. get% may be used to alter the return. Changes made by this function do not cause an immediate redraw.

**See also:**

OverdrawFrames(), OverdrawGetFrames(), Overdraw3D()

# Overdraw3D()

This command controls 3D overdrawing, it is the equivalent of the View menu Overdraw settings dialog. This command may only be used in a data view. There are two command variants. The first sets the overdraw values, the second reads back the current settings.

```
Func Overdraw3D(xProp, yProp{, xScale{, yScale{, zMode%}}});
Func Overdraw3D(get%);
```

xProp   The proportion of the available x space (in the range 0 to 1.0) to use for the 3D effect. Values outside the range 0 to 1 are limited to this range.

yProp   The proportion of the available y space to use for the 3D effect. Values outside the range 0 to 1 are limited to this range.

xScale  This optional argument sets how much to shrink the display width when going from the front to the back to give a perspective effect. Values are limited to the range 0 (shrink to nothing) to 1 (no shrink). If omitted, no change is made.

yScale  How much to shrink the display height when going from the front to the back to give a perspective effect, in the range 0 to 1. Values are limited to the range 0 (shrink to nothing) to 1 (no shrink). If omitted, there is no change.

zMode%  This optional argument sets the Z axis scaling, if it is omitted, no change is made. It is one of:

    0   Z position is set by the frame's position in the list of overdrawn frames

    1   Z position set by the frame number

    2   Z position set by the frame start time

get%    The variant of the command with one argument uses this value to indicate the value to read back:

    -1   to get current xProp

    -2   to get current yProp

    -3   to get current xScale

    -4   to get current yScale

    -5   to get current zMode%

Returns 0 when setting a value or the value requested by get%.

**See also:**

Overdraw(), OverdrawFrames(), OverdrawGetFrames()

# OverdrawFrames()

This function is used to set or modify the list of frames to overdraw in the data view. You can specify a range of frame numbers or a list of frames. If the function is used with no arguments it clears the overdraw frame list.

```
Func OverdrawFrames({sFrm%{, eFrm%{, mode%{, add%}}}});
Func OverdrawFrames(frm$|frm%[]{, mode%{, add%}});
```

sFrm%   First frame to include. This option processes a range of frames. sFrm% can also be a negative code as follows:

    -101    The overdraw list is cleared and the frame buffer is added.

    0    All sampled frames (on-line only).

    -1    All frames in the file are included.

    -2    The frame current at the time of this call.

    -3    Frames must be tagged.

    -5    Last N frames (on-line only).

    -6    Frames must be untagged.

Choosing a negative code with `add%` set to 0 will allow Signal to modify the overdraw status of individual frames as they are subsequently tagged/untagged. etc If this command is used with add% absent or set to non-zero then this dynamic behaviour will be lost.

eFrm%   Last frame to include. If this is -1 the last frame is the last in the data view. This argument is ignored if `sFrm%` is less than 1.

frm$    A frame specification string. This option specifies a list of frames using a string such as "1..32,40,50".

frm%[]  An array of frame numbers. This option provides a list of frame numbers in an array, the first array element holding the number of frames in the list.

mode%   If `mode%` is present it is used to supply an additional criterion for including each frame in the range, list or specification. If `mode%` is absent all frames are included. As with `sFrm%`, these modes will be applied dynamicaly if `add%` is 0. If `sFrm%` is –5 then this is the value of N. If `sFrm%` is 0 then this value is ignored, otherwise the modes are:

    0-n  Frames must have a state matching the value of `mode%`.

    -1   All frames in the range, list are included.

    -2   Only the frame current at the time of drawing, if in the list, is included.

    -3   Frames must be tagged.

    -6   Frames must be untagged.

add%    If `add%` is present and non-zero it determines whether the specified frames are to be added to or removed from the existing display list for the view, as follows:

    -1   Remove the frames from the existing display list.

    0    Clear the display list before adding these frames.

    1    Add the frames to the existing display list (the default).

    If `add%` is absent the new frame list will be added to the existing display list.

Returns The number of frames in the new overdraw list or a negative error code.

**See also:**
```
Overdraw(), OverdrawGetFrames()
```

# OverdrawGetFrames()

This function is used to get the list of frames overdrawn in the data view.

```
Func OverdrawGetFrames({list%[]});
```

list%   An optional array of frame numbers to hold the list of frame numbers. If the array is too short, enough frames are returned to fill the array. Element zero holds the number of frames returned in the array.

Returns The number of frames that would be returned if the array was of unlimited length, or zero if the view is not a data view.

**See also:**
```
Overdraw(), OverdrawFrames()
```

# P

# 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.

This function is now deprecated. You can replace `PaletteGet(col%, r,g,b)` with `ColourGet(-1, col%, r, g, b)` and dividing `r`, `g` and `b` by 100. `PaletteGet()` should only be used when compatibility with older version of Signal is required

**See also:**
Colour dialog, ColourGet(), 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%`.

This function is now deprecated. You can replace `PaletteSet(col%,r,g,b)` with `ColourSet(-1, col%, r/100, g/100, b/100)`. There is no equivalent of the `solid%` argument. `PaletteSet()` should only be used when compatibility with older version of Signal is required.

**See also:**
Colour dialog, ColourSet(), ChanColour(), Colour(), PaletteGet(), XYColour()

# PCA()

This command performs Principal Component Analysis on a matrix of data. This can take a long time if the input matrix is large.

```
Func PCA(flags%, x[][]{, w[]{, v[][]}});
```

flags%  Add the following values to control pre-processing of the input data:
  1   Subtract the mean value of each row from each row
  2   Normalise each row to have mean 0.0 and variance 1.0
  4   Subtract the mean value of each column from each column
  8   Normalise each column to have mean 0.0 and variance 1.0

You would normally pre-process the rows or the columns, not both. If you set flags for both, the rows are processed first.

x[][]   A `m` rows by `n` columns matrix of input data that is replaced by the output data. The first array index is the rows; the second is the columns. There must be at least as many rows as columns (`m >= n`). If you have insufficient data you can use a square matrix and fill the missing rows with zeros. If you were computing the principal components of spike data, on input, each row would be a spike waveform. On output, each row holds the proportion of each of the `n` principal components scaled by the `w[]` array that, when added together, would best (in a least-squares error sense) represent the input data.

w[]     This is an optional array of length at least `n` that is returned holding the variance of the input data that each component accounts for. The components are ordered such that `w[i] >= w[i+1]`.

v[][]   This is an optional square matrix of size `n` by `n` that is returned holding the `n` principal components in the rows.

Returns  0 if the function succeeded, -1 if `m < n`, -2 if `w` has less than `n` elements or `v` has less than `n` rows or columns.

# 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, Signal prints the arguments without a format specifier in a standard format and adds a new line character at the end. If you provide a format string and you require a new line character at the end of the output, include `\n` at the end of the format string.

```
Func Print(form$|const arg0{, const arg1{, const arg2...}});
```

`form$`    A string that specifies how to treat the following arguments. The string contains two types of characters: ordinary text that is copied to the output unchanged and format specifiers that convert the following arguments to text. Format specifiers start with `%` and end with one of the letters d, x, c, s, f, e or g in upper or lower case. For a `%` in the output, use `%%` in the format string.

`arg1,2`  The arguments used to replace `%c`, `%d`, `%e`, `%f`, `%g`, `%s` and `%x` type formats.

Returns  0 or a negative error code. Fields that cannot be printed are filled with asterisks.

### Format specifiers
The full format specifier is:  `%{flags}{width}{.precision}format`

*flags*
The `flags` are optional and can be placed in any order. They are single characters that modify the format specification as follows:

- –        Specifies that the converted argument is left justified in the output field.
- +        Valid for numbers, and specifies that positive numbers have a + sign.
- *space*  If the first character of a field is not a sign, a space is added.
- 0        For numbers, causes the output to be padded on the left to the field width with 0.
- #        For `x` format, `0x` is prefixed to non-zero arguments. For e, f and g formats, the output always has a decimal point. For `g` formats, trailing zeros are not removed.

*width*
If this is omitted, the output field will be as wide as is required to express the argument. If present, it is a number that sets the minimum output field width. If the output is narrower than this, the field is padded on the left (on the right if the – flag was used) to this width with spaces (zeros if the 0 flag was used). The maximum width for numbers is 100.

*precision*
This number sets the maximum number of characters to be printed for a string, the number of digits after the decimal point for e and f formats, the number of significant figures for g format and the minimum number of digits for d format (leading zeros are added if required). It is ignored for c format. There is no limit to the size of a string. Numeric fields have a maximum `precision` value of 100.

*format*

The format character determines how the argument is converted into text. Both upper and lower cased version of the format character can be given. If the formatting contains alphabetic characters (for example the `e` in an exponent, or hexadecimal digits `a-f`), if the formatting character is given in upper case the output becomes upper case too (`e+23` and `0x23ab` become `E+23` and `0X23AB`). The formats are:

c   The argument is printed as a single character. If the argument is a numeric type, it is converted to an integer, then the low byte of the integer (this is equivalent to `integer mod 256`) is converted to the equivalent ASCII character. You can use this to insert control codes into the output. If the argument is a string, the first character of the string is output. The following example prints two tab characters, the first using the standard tab escape, the second with the ASCII code for tab (8):

```
Print("\t%c", 8);
```

d   The argument must be a numeric type and is printed as a decimal integer with no decimal point. If a string is passed as an argument the field is filled with asterisks. The following prints "  23,0002":

```
Print("%4d,%.4d", 23, 2.3);
```

e   The argument must be a numeric type; otherwise the field is filled with asterisks. The argument is printed as `{-}m.ddddddde±xx{x}` where the number of `d`'s is set by the precision (which defaults to 6). A precision of 0 suppresses the decimal point unless the `#` flag is used. The exponent has at least 2 digits (in some implementations of Signal there may always be 3 digits, others use 2 digits unless 3 are required). The following prints "`2.300000e+01,2.3E+00`":

```
Print("%4e,%.1E", 23, 2.3);
```

f   The argument must be a numeric type; otherwise the field is filled with asterisks. The argument is printed as `{-}mmm.ddd` with the number of `d`'s set by the precision (which defaults to 6) and the number of `m`'s set by the size of the number. A precision of 0 suppresses the decimal point unless the `#` flag is used. The following prints "+23.000000,0002.3":

```
Print("%+f,%06.1f", 23, 2.3);
```

g   The argument must be a numeric type; otherwise the field is filled with asterisks. This uses `e` format if the exponent is less than -4 or greater than or equal to the precision, otherwise `f` format is used. Trailing zeros and a trailing decimal point are not printed unless the `#` flag is used. The following prints "`2.3e-06,2.300000`":

```
Print("%g,%#g", 0.0000023, 2.3);
```

s   The argument must be a string; otherwise the field is filled with asterisks.

x   The argument must be a numeric type and is printed as a hexadecimal integer with no leading `0x` unless the `#` flag is used. The following prints "`1f,0X001F`":

```
Print("%x,%#.4X", 31, 31);
```

### Arrays in the argument list

The `d`, `e`, `f`, `g`, `s` and `x` formats support arrays. One dimensional arrays have elements separated by commas; two dimensional arrays use commas for columns and new lines for rows. Extra new lines separate higher dimensions. If there is a format string, the matching format specifier is applied to all elements.

### Infinity and Not a Number

The floating point number format normally stores a number within the floating point range. However, the format can also store positive infinity, negative infinity and Not a Number values. These are tricky to generate inside Signal, but you can do it by dividing by zero in some circumstances, for example. If you try to print these values in f or g format you get #IND for a NaN (for example the result of 0.0/0.0) or #INF or -#INF for an infinity (for example 1.0/0.0 or -1.0/0.0).

### 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$(form$|const arg0{, const arg1{, const arg2...}}});
```

form$  An optional string with formatting information. See `Print()` for a description.

arg1,2  The data to format into a string.

Returns  It returns the string that is the result of the formatting operation. Fields that cannot be printed are filled with asterisks.

**See also:**
```
Asc(), Chr$(), DelStr$(), LCase$(), Left$(), Len(), Mid$(), Print(), PrintLog(),
Right$(), Str$(), UCase$(), Val()
```

# PrintLog()

This command prints to the log window. The syntax is identical to `Print()`. The output always goes to the log views and is always placed at the end of the view contents.

```
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()

This function processes data into the current memory or XY view or to a memory channel in the current data view. The view or channel must have been generated by using `SetXXXX()` or `MeasureToChan()` on a source data view which must not have been closed. This function takes data starting from a specified position in the current frame in the source data view and processes it.

```
Func Process(start{, clear%{, opt%{, optx%{, chan%}}}});
```

start   The source view x axis position from which to start processing. Positions less than `MinTime()` are treated as `MinTime()`. Positions greater than `Maxtime()` mean no processing is done. The offset from the start of frame in `SetXXX()` will be ignored. If the start position specified plus the width of data required goes past the end of the source data then no data is processed.

clear%  If present, and non-zero, the memory or XY view or destination channel data is cleared before the results of the analysis are added to the view and `Sweeps()` result is reset.

opt%    If present, and non-zero, the display of data in the memory or XY view or destination channel is optimised after processing the data.

optx%   For XY views only, if present and non-zero, the X axis in the XY view is optimised after processing the data.

chan%   For memory channels only, if present and a valid memory channel number, it is the channel number to process to.

Returns  One if all is OK, zero if no data was processed or a negative error code.

A common mistake is to forget that the current view is not the source view and to use `View(0).xxx` when `View(ViewSource()).xxx` was intended.

**See also:**
```
SetXXX(), SetAverage(), SetPower(), ProcessAll(), ProcessFrames(), ProcessOnline(),
Sweeps(), Optimise()
```

# ProcessAll()

This function is used in a data view to process all memory views, XY views or memory channels that are derived from it.

```
Func ProcessAll(sFrm%{, eFrm%{, chans%}});
```

sFrm%   The first frame to process.

eFrm%   If this is present, a range of frames is processed, from `sFrm%` to `eFrm%` inclusive. If omitted only `sFrm%` is processed.

chans%  If this is present and set to –1 then all memory channels will be processed otherwise all memory or XY views will be processed.

Returns  Zero if no errors or a negative error code.

For each derived memory view, the settings of the `clear%` and `opt%` arguments are taken from the last call of `Process()` or `ProcessFrames()`. If a memory view had not yet been processed `clear%` is zero and `opt%` is non-zero.

**See also:**
`Process(), ProcessFrames(), ProcessOnline()`

# ProcessFrames()

This function is used in a derived memory or XY view to process specified frames from the source data view or in a data view to process specified frames of a memory channel. You can process a range of frame numbers or specify a list of frames.

```
Func ProcessFrames(sF%{,  eF%{, mode%{, clear%{, opt%{, optx%{, chan%{, auto
%}}}}}}});
Func ProcessFrames(frm$|frm%[]{, mode%{, clear%{, opt%{, optx%{, chan%{,
auto%}}}}}});
```

sF%     First frame to process. This option processes a range of frames. `sFrm%` can also be a negative code as follows:

    -1   All frames in the file are included.

    -2   The current frame.

    -3   Frames must be tagged.

    -6   Frames must be untagged.

eF%     Last frame to process. If this is -1 the last frame in the data view is used. This argument is ignored if `sF%` is a negative code.

frm$    A frame specification string. This option specifies a list of frames using a string such as "1..32,40,50".

frm%[]  An array of frame numbers to process. This option provides a list of frame numbers in an array. The first element in the array holds the number of frames in the list.

mode%   If `mode%` is present, it is used to supply an additional criterion for including each frame in the range, list or specification. If `mode%` is absent all frames are included. The modes are:

    0-n  Frames must have a state matching the value of `mode%`.

    -1   All frames in the specification are processed.

    -2   Only the current frame, if in the list, will be processed.

    -3   Frames must also be tagged.

    -6   Frames must also be untagged.

clear%  If present, and non-zero, the memory or XY view or destination channel data is cleared before the results of the analysis are added to the view and `Sweeps()` result is reset.

opt%    If present, and non-zero, the display of data in the memory or XY view or destination channel is optimised after processing the data.

optx%   For XY views only, if present and non-zero, the X axis in the XY view is optimised after processing the data.

chan%   For memory channels only, this is the channel number to process to. It is ignored if it is not a valid memory channel number.

auto%   Set this to 1 for automatic reprocessing if the source data changes, otherwise omit it or set it to zero.

Returns  Zero if no errors or a negative error code.

**See also:**
```
Process(), ProcessAll(), ProcessOnline()
```

# ProcessOnline()

This function is equivalent to the process dialog for a memory or XY view derived from a sampling document or a memory channel in the sampled document. It does not cause any processing, but sets up processing so that when the next automatic processing can happen, the parameters set by this command are used.

```
Func ProcessOnline(mode%{, val%{, up%{, opt%{, optx%{, chan%(, clear%(,
leeway}}}}}});
```

mode%  The modes are:

    0    All sampled sweeps are processed regardless of whether they are written to disk. This mode will not work if you are using Fast triggers or Fast Fixed interval sampling modes.

    -1    All sweeps saved to disk are processed.

    -3    All tagged frames written to disk are processed.

    -4    Sweeps with a state of val% are processed. A state of 0 is used if val% is not provided.

    -5    Processes the last val% sweeps including the latest. The result is cleared and Sweeps() count is reset to 0 before each update.

    -6    All untagged frames written to disk are processed.

val%  With mode% set to -4 or -5 this provides the value for the frame state or the number of frames respectively. With other mode% values it sets a frame subset value that qualifes the frames selected by mode%; positive val% values set a frame state code while using -1, -3 or -6 selects all, tagged and untagged frames respectively. Other negative values will give undefined results.

up%  This provides the number of frames before the next process or zero for no gap.

opt%  If present, and non-zero, the display of data in the memory or XY view or destination channel is optimised after processing the data.

optx%  For XY views only, if present and non-zero, the X axis in the XY view is optimised after processing the data.

chan%  For memory channels only, this is the channel number for which we are changing the settings. It is ignored if it is not a valid memory channel number.

clear%  Set to 1 to clear destination data before each process. Set to 0 or omit to leave destination data unchanged before each process.

leeway  This is used when processing to a memory channel as set up using MeasureToChan(). It sets how close to the end of the incoming data cursor 0 is allowed to get, so as to allow space after cursor zero for measurements or for other cursors to position themselves.

Returns  0 or a negative error code.

**See also:**
```
MeasureToChan(), Process(), ProcessAll(), ProcessFrames()
```

# Profile()

This command can create and delete keys and store and read integer and string values within the Signal section of the registry. Signal stores information within the `HKEY_CURRENT_USER\Software\CED\Signal` section of the system registry. The registry is organised as a tree of keys with lists of values attached to each key. If you think of the registry as a filing system, the keys are folders and the values are files. Keys and values are identified by case-insensitive text strings.

You can view and edit the registry with the `regedt32` program, which is part of your system. Select Run from the start menu and type `regedt32`, then click OK. Please read the `regedt32` help information before making any registry changes. It is a very powerful program; careless use can severely damage your system.

Do not write vast quantities of data into the registry; it is a system resource and should be treated with respect. If you must save a lot of data, write it to a text or binary file and save the file name in the registry. If you think that you may have messed up the Signal section of the registry, use `regedt32` to locate the `Signal` section and delete it. The next time you run Signal the section will be restored; you will lose any preferences you had set.

```
Proc Profile(key${, name${, val%{, &read%}}});
Proc Profile(key${, name${, val${, &read$}}});
```

key$    This string sets the key to work on inside the `Signal` section of the registry. If you use an empty string, the `Signal` key is used. You can use nested keys separated by a backslash, for example `"My bit\ \stuff"` to use the key `stuff` inside the key `My bit`. The key name may not start with a backslash. Remember that you must use two backslashes inside quote marks; a single backslash is an escape character. It is never an error to refer to a key that does not exist; the system creates missing keys for you.

name$   This string identifies the data in the key to read or write. If you set an empty name, this refers to the `(default)` data item for the key set by key$.

val     This can be either a string or an integer value. If `read` is omitted, this is the value to write to the registry. If `read` is present, this is the default value to return if the registry item does not exist.

read    If present, it must have the same type as `val`. This is a variable that is set to the value held in the registry. If the value is not found in the registry, the variable is set to the value of the `val` argument.

`Profile()` can be used with 1 to 4 arguments. It has a different function in each case:

1   The key identified by key$ is deleted. All sub-keys and data values attached to the key and sub-keys are also deleted. Nothing is done if key$ is empty.
2   The value identified by name$ in the key key$ is deleted.
3   The value identified by name$ in the key key$ is set to `val%` or `val$`.
4   The value identified by name$ in the key key$ is returned in `val%` or `val$`.

The following script example collects values at the start, then saves them at the end:

```
var path$, count%;
Profile("My data", "path", "c:\\work", path$); 'get initial path
Profile("My data", "count", 0, count%); 'and initial count
...                                      'your script...
Profile("My data","path", path$);        'save final value
Profile("My data","count", count%);      'save final count
```

### Registry use by Signal
The `HKEY_CURRENT_USER\Software\CED\Signal` key contains the following keys that are used by Signal:

*BarList*
This key holds the list of scripts to load into the script bar when Signal starts.

*Edit*
This key holds the editor settings for scripts, output sequences and general text editing.

*PageSetup*
This key holds the margins in units of 0.01 mm for printing data views, and the margins in mm for text-based views. It also holds header and footer text for text-based views.

*Preferences*
The values in this key are mainly set by the Edit menu preferences. If you change any Edit menu Preferences value in this key, Signal will use the changed information immediately. The values are all integers except the file path, which is a string:

| | |
|---|---|
| 1401 ADC range | 5000=5 Volt 1401 hardware, 10000=10 Volt 1401 hardware, -1=Use last seen hardware settings |
| Assume Power | 0=Do not assume Power1401 hardware, 1=do assume. |
| Decorated states text online | 0=Don't show extra states information in sampling window title, 1= show extra states information. |
| Defer Optimise | 0=Y-axis optimised on data acquired so far when on-line, 1=Y-axis optimise deferred to sweep end when on-line. |

| | |
|---|---|
| Disable AuxTel | 0=use auxiliary telegraph support if installed, 1=ignore any installed auxiliary telegraph support (mostly for demonstrating the software!) |
| Enhanced MetaFile | 0=Windows metafile, 1=enhanced metafile for clipboard. |
| Enter debug on error | 1=Enter the script debugger when an error occurs. 0=Do not. |
| Error log level | Sets the minimum level of information that will be logged; 0=information, 1=warnings, 2=errors, 3=critical errors. |
| File shorts | 0=Waveform data to be written to CFS file as 16 bit integers, 1=to be written as floating point values. Add 256 if calibrated zero is to be kept at zero volts. |
| File update | 0=Discard changes to data, 1=query the user, 2=always save changes. |
| Font Italic | 0=Use non-italic font as default for file and memory views. 1=use italic font. |
| Font Name | Name of font to use as default for file and memory views. |
| Font Pitch | 0=Use default-pitch font for the above, 1=use a fixed-pitch font (all characters the same width), 2=use a variable-pitch font. To this value you should add: 4=don't care which family of font used, 8=use a serifed, variable-width font, 16=sans-serif, variable width font, 32=constant-width font, 64=cursive font, 128=decorative font. |
| Font Size | Size of font to use as default for file and memory views. |
| Font Weight | 400=Normal font, 700=bold font. |
| Force idle cycles | 0 – 65535 number of time a script idle is called before handing time back to the System. 0=No limit. |
| Force idle time | Number of ms to force Signal to idle for before giving time back to System (0-200). 0=No limit. |
| Frame 1 on sample finish | 0=Show last filed frame when sampling finishes, 1=show frame 1 when sampling finishes. |
| Frame time mode | 0=Seconds, 1=HH:MM:SS, 2=Time of day]]. |
| Keep ADC range | 0=Maintain always, 1=Keep showing full ADC range if already doing so, 2=Maintain ADC range percentage. |
| Kill dynamic clamp | 0=allow dynamic clamp usage, 1=ignore all dynamic clamp setups (mostly for use when demonstrating the software!). |
| 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. |
| Maximum Log lines | Sets the maximum number of text lines in the log window or 0 for no limit. |
| MetaFile Scale | 0-11 selects from the list of allowed scale factors. |
| MetaFile NoCompress | 0=allow compressed metafiles, 1=no metafile compression. |
| New file path | New data file directory or blank for current folder. |
| No background colour check | 0=check all colours against background & enforce visibility, 1=no checks - just use the colours set. |
| No flicker free drawing | 0=Use flicker free drawing, 1=Do not. |
| No save prompt | 0=Prompt to save derived views, 1=no prompt. |
| No Y axis invert on drag | 0=Allow dragging to invert a Y axis, 1=don't allow inversion. |
| Prompt comment | 1=Prompt for File Comment when sampling stops. 0=Do not. |
| Provide clamp features | 0=No clamping features are provided in user interface, 1=clamping features are provided & used. |
| Save modified scripts | 1=Save modified scripts before running them. 0=Do not save. |
| Text print mode | Controls colour usage in text printing; 0=use screen colours, 1=invert if light on dark, 2=black on white, 3=colour on white. |
| Time mode | 0=Display seconds, 1=display ms, 2=display µ]]s. |
| Update interval | Number of ms between on-line updates or script idles. |

The keys with names starting `"Bars-"` are used by system code to restore dockable toolbars. You can delete them all safely; any other change is likely to crash Signal.

### Recent file list
This key holds the list of recently used files that appear at the bottom of the file menu.

### Recover
This key holds the information to recover data from interrupted sampling sessions.

### Settings
This is where the evaluate bar saves the last few evaluated lines.

### Tip
The *Tip of the Day* dialog uses this key to remember the last tip position.

### Version
Signal uses this key to detect when a new version of the program is run for the first time.

### Win32
This key holds the desired working set sizes. The working set sizes in use are displayed in the Help menu About Signal dialog. Click the Help button in this dialog to read more about using these registry values. The values are as follows:

| | |
|---|---|
| Minimum working set | Minimum size in kB (units of 1024 bytes), default is 800. |
| Maximum working set | Maximum size in kB, default is 4000 (4 MB). |

**See also:**
About the Working Set, `ViewUseColour()`

# ProgKill()

This function terminates a program started using `ProgRun()`. This can be dangerous, as it will terminate a program without giving it the opportunity to save data.

```
Func ProgKill(pHdl%);
```

`pHdl%`   A program handle returned by `ProgRun()`.

Returns  Zero or a negative error code.

**See also:**
`ProgRun()`, `ProgStatus()`

# ProgRun()

This function runs a program using command line arguments as if from a command prompt. Use `ProgRun()` to test the program status, `ProgKill()` to terminate it. The program inherits its environment variables from Signal, see `System$()` for details.

```
Func ProgRun(cmd${, code% {,xLow, yLow, xHigh, yHigh}});
```

`cmd$`    The command string as typed at a command prompt. To run shell command x use `"cmd /c x"`. To run another copy of Signal, use `"cfsview.exe /M {filename}"`.

`code%`   If present, this sets the initial application window state: 0=Hidden, 1=Normal, 2=Iconised, 3=maximised. Some programs set their own window state so this may not work. The next 4 arguments set the Normal window position:

`xLow`    Position of the left window edge as a percentage of the screen width.

`yLow`    Position of the top window edge as a percentage of the screen height.

`xHigh`   The right hand edge as a percentage of the screen width.

`yHigh`   The bottom edge position as a percentage of the screen height.

Returns  A program handle or a negative error code. `ProgStatus()` releases resources associated with the handle when it detects that the program has terminated.

**See also:**
```
FileCopy(), FileDelete(), ProgKill(), ProgStatus(), System$()
```

# ProgStatus()

This function tests if a program started with `ProgRun()` is still running. If it is not, resources associated with the program handle are released.

**`Func ProgStatus(pHdl%);`**

pHdl%   The program handle returned by `ProgRun()`.

Returns  1=program is running, 0=terminated, resources released, handle now invalid. A negative error code (-1525) means that the handle is invalid.

**See also:**
```
ProgKill(), ProgRun()
```

# Protocol commands

# ProtocolAdd()

This function adds a new protocol to the list of protocols defined in the sampling configuration. This function normally operates on the stored sampling configuration but if used during sampling it operates upon the on-going sampling.

**`Func ProtocolAdd(name$);`**

name$   The name for the new protocol, which must not be blank.

Returns  The number for the new protocol or a negative error code.

**See also:**
```
Protocols(), ProtocolDel(), ProtocolClear(), ProtocolName$()
```

# ProtocolClear()

This function initialises a protocol defined in the sampling configuration. This function normally operates on the stored sampling configuration but if used during sampling it operates upon the on-going sampling.

**`Func ProtocolClear(num%|name$);`**

num%   The number of the protocol to clear, from 1 to the number returned by `Protocols()`.

name$   The name of the protocol to be cleared.

Returns  Zero or a negative error code.

**See also:**
```
Protocols(), ProtocolDel(), ProtocolAdd(), ProtocolName$()
```

# ProtocolDel()

This function deletes a protocol from the list defined in the sampling configuration. This function normally operates on the stored sampling configuration but if used during sampling it operates upon the on-going sampling.

**`Func ProtocolDel(num%|name$);`**

num%   The number of the protocol to delete, from 1 to the number returned by `Protocols()`.

name$   The name of the protocol to be deleted.

Returns  Zero or a negative error code.

**See also:**

Protocols(), ProtocolAdd(), ProtocolClear(), ProtocolName$()

# ProtocolEnd()

This sets what happens when the end of a protocol is reached. This function normally operates on the stored sampling configuration but if used during sampling it operates upon the on-going sampling.

```
Func ProtocolEnd(num%|name${, atEnd%});
```

num%    The number of the protocol to delete, from 1 to the number returned by `Protocols()`.

name$   The name of the protocol to be deleted.

atEnd%  Set to 0 for the protocol finishing, or 1 to n to select chaining to protocol 1 to n. No protocol is selected if `next` value is above current count of protocols.

Returns  The previous value of `atEnd%`.

**See also:**
Protocols(), ProtocolAdd(), ProtocolClear(), ProtocolName$()

# ProtocolFlags()

This function gets the flags for a protocol defined in the sampling configuration and optionally sets the flags to a new value. This function normally operates on the stored sampling configuration but if used during sampling it operates upon the on-going sampling - though it will probably not operate correctly when used on a protocol that is already running.

```
Func ProtocolFlags(num%|name${, new%});
```

num%    The number of the protocol to use, from 1 to the number returned by `Protocols()`.

name$   The name of the protocol to use.

new%    If present, the new protocol flags value. This is the sum of values for each flag option, the values are:

    1    Initialise pulse variations when protocol starts.

    2    Sampling switches to state 0 when protocol finishes.

    4    Turn on writing to disk when protocol starts.

    8    Selects not turning off writing to disk on finishing.

    16  Selects creation of a button in the control bar for this protocol.

    32  Selects cycling protocol states only on writing, otherwise always.

    64  Selects running this protocol automatically at the start of sampling.

    128 Enables the use of the optional per-step write flags.

Returns  The flags for this protocol before the call.

**See also:**
Protocols(), ProtocolClear(), ProtocolName$(), ProtocolRepeats()

# ProtocolName$()

This function gets the name for a protocol defined in the sampling configuration and optionally sets a new name. This function normally operates on the stored sampling configuration but if used during sampling it operates upon the on-going sampling.

```
Func ProtocolName$(num%|name${, new$});
```

num%    The protocol number to use, from 1 to the number returned by `Protocols()`.

name$   The name of the protocol to use.

new$    If present, this sets the new protocol name.

Returns  The name of the protocol before this call.

**See also:**

```
Protocols(), ProtocolAdd(), ProtocolDel(), ProtocolClear()
```

# ProtocolRepeats()

This function gets the number of repeats set for a protocol defined in the sampling configuration and optionally sets a new repeat count. This function normally operates on the stored sampling configuration but if used during sampling it operates upon the on-going sampling.

```
Func ProtocolRepeats(num%|name${, new%});
```

num%    The protocol number to use, from 1 to the number returned by `Protocols()`.

name$    The name of the protocol to use.

new%    If present, sets the new repeat count. Any repeat count from 1 to 1000 can be set, or zero for a protocol which repeats forever.

Returns  The repeat count for the protocol before the call.

**See also:**
```
Protocols(), ProtocolFlags(), ProtocolEnd()
```

# Protocols()

This function returns the number of protocols defined in the sampling configuration. This function normally operates on the stored sampling configuration but if used during sampling it operates upon the on-going sampling.

```
Func Protocols();
```

Returns  The number of protocols defined in the sampling configuration.

**See also:**
```
ProtocolAdd(), ProtocolDel(), ProtocolClear(), ProtocolName$()
```

# ProtocolStepGet()

This function gets information about a specific step within a protocol defined within the sampling configuration. This function normally operates on the stored sampling configuration but if used during sampling it operates upon the on-going sampling.

```
Func ProtocolStepGet(num%|name$, step, &state, &repeat, &next{, &write});
```

num%    The protocol number to use, from 1 to the number returned by `Protocols()`.

name$    The name of the protocol to use.

step    The step in the protocol, from 1 to 10.

state    This parameter is updated with the state for this step.

repeat  This parameter is updated with the repeat count for this step.

next    This parameter is updated with the next step value for this step.

write    If present this is updated with the per-step write flag; 1 for writing and 0 for not writing. The per-step write flags have no effect unless the appropriate enable bit is set in the protocol flags.

Returns  Zero or a negative error code.

**See also:**
```
Protocols(), ProtocolFlags(), SampleStates(), ProtocolClear(), ProtocolName$()
```

# ProtocolStepSet()

This function sets up a specific step within a protocol defined within the sampling configuration. This function normally operates on the stored sampling configuration but if used during sampling it operates upon the on-going sampling - though it will probably not operate correctly when used on a protocol that is already running.

```
Func ProtocolStepSet(num%|name$, step%, state, repeat, next{, write});
```

num%     The number of the protocol to use, from 1 to the number returned by `Protocols()`.

name$    The name of the protocol to use.

Step%    The step in the protocol, from 1 to 10.

state    This parameter sets the state for this step, from 0 to the maximum state number in use. State numbers higher than the maximum are set to zero.

repeat   This parameter sets the repeat count for this step, from 1 to 1000.

next     This parameter sets the next step value for this step, from 0 to 10. A value of zero terminates protocol execution.

write    If present this value sets the per-step write flag; use 1 for writing and 0 for not writing. If `write` is omitted the per-step write flag is set to not writing. The per-step write flags have no effect unless the appropriate enable bit is set in the protocol flags.

Returns  Zero or a negative error code.

**See also:**
```
Protocols(), ProtocolFlags(), SampleStates(), ProtocolClear(), ProtocolName$()
```

# Pulse output commands

The PulseXXX family of commands can be used to control the pulse outputs generated during sampling sweeps. Pulses can be generated on up to eight 1401 DACs and on 8 bits of dedicated digital output. For Micro1401s only two DACs are available while on Power1401 systems normally four DACs are available but more can be added (up to the maximum of 8) by extending the 1401 hardware with top boxes. These functions normally operate on the stored sampling configuration but if used during sampling they operate upon the on-going sampling.

As part of the Signal multiple states facilities, each state can have a separate set of pulse outputs. Because of this, all script functions that access the pulses information have a parameter to select the state. For sampling situations that do not involve multiple states, this parameter should be set to zero.

Individual pulses can be specified by their number or by name. For access by number the pulses for a given output are kept in a sorted list in order of their start time. The (always present) initial level is pulse number zero and subsequent pulses are number one and upwards. Though access by number seems straightforward, it does have some drawbacks. Firstly, when the start time of a pulse is changed the ordering of the list can change and the pulse number will be changed. Secondly, for complex reasons, the arbitrary waveform output item is always attached to the control track list, and does not appear in the output lists for the DACs to which the waveform output is sent. This can make things very confusing! Therefore we recommend that, for non-trivial pulse output arrangements with a lot of pulse movement or manipulation, individual pulses are accessed by name.

**See also:**
```
PulseAdd(), PulseClear(), PulseDataGet(), PulseDataSet(), PulseDel(), PulseFlags(),
PulseName$(), Pulses(), PulseTimesGet(), PulseTimesSet(), PulseType(), PulseVarGet(),
PulseVarSet(), PulseWaveformGet(), PulseWaveformSet(), PulseWaveGet(), PulseWaveSet()
```

# PulseAdd()

This function adds a new pulse to the output pulses for a given state and output. The pulse created will use a default set of parameters depending upon the type of pulse and the outputs used (for digital outputs the pulse is created on all enabled outputs, for DAC output pulses the pulse created will have a default amplitude and other values), you will need to use `PulseDataSet()` to change these values to what is actually wanted. Note that, if you add a waveform output item to a set of pulses that already contains another waveform output item, the output DACs and waveform rate are set to match the existing waveform(s) as these cannot vary between the waveform items. The ability to add multiple waveform items was added in Signal version 5.00.

```
Func PulseAdd(state%, out%, type%, name$, time, len{, flags%});
```

state%   The state (set of pulses) to which this applies, from 0 to 256. Use 0 if multiple states are not in use.

out%     The output to which this applies. Values of zero upwards select the corresponding DAC output, -1 selects the digital outputs, -2 selects the control track, though for items that can be added to the control track (waveform and marker generation items) out% is ignored and the control track is always used.

type%    A code for the type of pulse. The legal codes are:
1    A simple square pulse.
2    A square pulse with varying amplitude.
3    A square pulse with varying duration.
4    A ramp pulse.
5    A cosine wave segment.
6    An arbitrary waveform (for the control track only, out% is ignored), the rate is initialised to 100Hz.
7    A pulse train; the number of pulses is set to 2
8    A marker generation item (for the control track only, out% is ignored). This type of pulse was added in Signal version 5.00.

name$    The name for the new pulse, which can be blank. Pulse names are intended to make it easier to manipulate pulses from a script as changing the start time of a pulse can cause the pulse number to spontaneously change.

time     The start time for the pulse in seconds from the start of the outputs.

len      The length of the pulse, in seconds. For a pulse train, this is the length of the individual pulses in the train, not the length of the entire train.

flags%   If present, this sets the flags for the pulse. Flag bit 0 is set for varying-width pulses to push following pulses back, bit 1 is set for pulses to stay up at the end, bit 3 is set for a marker item to read the digital inputs to set the marker data. If this parameter is omitted, the pulse flags are cleared.

Returns  The number of the new pulse or a negative error code. The initial level item is always present as pulse zero, so the smallest successful return value is 1.

**See also:**
Pulses(), PulseDataSet(), PulseDel(), PulseClear(), SampleStates(),
SampleOutLength(), PulseName$()

# PulseClear()

This function deletes all the pulses for a given state and output (or all outputs) and sets the initial levels of the outputs to zero.

**Func PulseClear(state%{, out%});**

state%   The state (set of pulses) to which this applies, from 0 to 256. Use 0 if multiple states are not in use.

out%     The output to which this applies. Values of zero upwards select the corresponding DAC output, -1 selects the digital outputs, -2 the control track. If this parameter is omitted, then all outputs for the selected state are cleared. Note that to remove arbitrary waveform outputs you should clear the control track and not the DACs on which the waveforms are output.

Returns  Zero or a negative error code.

**See also:**
Pulses(), PulseAdd(), PulseDel(), SampleStates()

# PulseDataGet()

This function retrieves the amplitude and other values for a pulse in the outputs for a given state and output. Up to four data values can be retrieved, the meaning of most of these varies with the pulse type. A separate function, PulseWaveGet(), retrieves the settings for waveform outputs.

**Func PulseDataGet(state%, out%, num%|name$, &amp{, &val1{, &val2{,
&val3}}});**

state%   The state (set of pulses) to which this applies, from 0 to 256. Use 0 if multiple states are not in use.

`out%`   The output to which this applies. Values of zero upwards select the corresponding DAC output, -1 selects the digital outputs, -2 the control track.

`num%`   The number of the pulse in question, from 0 to the number of pulses-1.

`name$`   The name of the pulse to use.

`amp`   This is updated with the amplitude or level of the pulse, or the bit value for digital pulses.

`val1`   This is updated with the end amplitude for ramps, the initial phase for cosines, the number of pulses for pulse trains and the marker code for digital marker items.

`val2`   This is updated with the step mode for ramps, the centre value for sines, and the inter-pulse gap for pulse trains. The step mode value is 0 for both ends, 1 for start only and 2 for end only.

`val3`   This is updated with the cycle period for sines only.

Returns   Zero or a negative error code.

**See also:**
`Pulses(), PulseAdd(), PulseDel(), PulseDataSet(), PulseName$()`

# PulseDataSet()

This function sets the amplitude and other values for a pulse in the outputs for a given state and output. Up to four data values can be set, the meaning of most of these varies with the pulse type. A separate function, `PulseWaveSet()`, is used to change the settings for arbitrary waveform output.

`Func PulseDataSet(state%, out%, num%|name$, amp{, val1{, val2{, val3}}}});`

`state%` The state (set of pulses) to which this applies, from 0 to 256. Use 0 if multiple states are not in use.

`out%`   The output to which this applies. Values of zero upwards select the corresponding DAC output, -1 selects the digital outputs, -2 the control track.

`num%`   The number of the pulse in question, from 0 to the number of pulses-1.

`name$`   The name of the pulse to use.

`amp`   This sets the amplitude or level of the pulse, or the bit mask value for digital pulses. The bit mask sets the digital outputs on which the pulse appears, see `SampleDigOMask()` for details of bit mask usage.

`val1`   This sets the end amplitude for ramps, the initial phase for cosines,the number of pulses for pulse trains and the marker code for digital marker items.

`val2`   This sets the step mode for ramps, the centre value for sines, and the inter-pulse gap for pulse trains. The step mode value is 0 for both ends, 1 for start only and 2 for end only.

`val3`   This sets the cycle period for sines only.

Returns   Zero or a negative error code.

**See also:**
`Pulses(), PulseAdd(), PulseDel(), PulseDataGet(), PulseName$(), SampleDigOMask()`

# PulseDel()

This function deletes a pulse from the output pulses for a given state and output.

`Func PulseDel(state%, out%, num%|name$);`

`state%` The state (set of pulses) to which this applies, from 0 to 256. Use 0 if multiple states are not in use.

`out%`   The output to which this applies. Values of zero upwards select the corresponding DAC output, -1 selects the digital outputs, -2 the control track.

`num%`   The number of the pulse to delete, from 1 to the number of pulses-1 (you cannot delete pulse zero - the initial level).

`name$`   The name of the pulse to delete. You cannot delete  the initial level.

Returns  Zero or a negative error code.

**See also:**
`Pulses(), PulseClear(), PulseAdd(), SampleStates(), SampleOutMode()`

# PulseFlags()

This function retrieves, and optionally sets, the options flags for a pulse in the outputs for a given state and output.

`Func PulseFlags(state%, out%, num%|name${, flags});`

`state%` The state (set of pulses) to which this applies, from 0 to 256. Use 0 if multiple states are not in use.

`out%` The output to which this applies. Values of zero upwards select the corresponding DAC output, -1 selects the digital outputs, -2 the control track.

`num%` The number of the pulse in question, from 0 to the number of pulses-1.

`name$` The current name of the pulse in question.

`flags` If present, this sets the new flags for the pulse. Flag bit 0 is set for varying-width pulses to push following pulses back, bit 1 is set for pulses to stay up at the end and bit 3 is set for digital marker items to read the digital inputs to set the marker code (bit 2 is reserved).

Returns  The flags for the pulse at the time of the function call.

**See also:**
`Pulses(), PulseAdd(), PulseDel(), SampleStates(), SampleOutMode()`

# PulseName$()

This function retrieves, and optionally sets, the name of a pulse in the outputs. Pulse names are intended to make it easier to manipulate pulses from a script as changing the start time of a pulse can cause the pulse number to spontaneously change.

`Func PulseName$(state%, out%, num%|name${, new$});`

`state%` The state (set of pulses) to which this applies, from 0 to 256. Use 0 if multiple states are not in use.

`out%` The output to which this applies. Values of zero upwards select the corresponding DAC output, -1 selects the digital outputs, -2 the control track.

`num%` The number of the pulse in question, from 0 to the number of pulses-1.

`name$` The current name of the pulse in question.

`new$` The new name for the pulse. Blank pulse names are legal.

Returns  The name of the pulse at the time of the function call.

**See also:**
`Pulses(), PulseAdd(), PulseDel(), SampleStates(), SampleOutMode()`

# Pulses()

This function returns the number of pulses for a given state and output. This number is usually straightforward to use, but can be complicated by the fact that waveform output items always appear on control track and not on any DAC used for the waveform output.

`Func Pulses(state%, out%);`

`state%` The state (set of pulses) to which this applies, from 0 to 256. Use 0 if multiple states are not in use.

`out%` The output to which this applies. Values from 0 upwards select the corresponding DAC, -1 selects the digital outputs, -2 the control track.

Returns The number of pulses on this output. This value will be 1 or more as there is always one pulse defined for an output, the initial level.

**See also:**
`PulseDel(), PulseAdd(), PulseWaveSet(), PulseName$()`

# PulseTimesGet()

This function retrieves the times for a pulse in the outputs for a given state and output.

`Func PulseTimesGet(state%, out%, num%|name$, &time, &len);`

state% The state (set of pulses) to which this applies, from 0 to 256. Use 0 if multiple states are not in use.

out% The output to which this applies. Values of zero upwards select the corresponding DAC output, -1 selects the digital outputs, -2 the control track.

num% The number of the pulse in question, from 1 to the number of pulses-1. Zero is not meaningful here because there are no times for the initial level.

name$ The name of the pulse to use.

time This is updated with the start time for the pulse, in seconds from the start of the pulse outputs.

len This is updated with the length of the pulse (this is the individual pulse length for trains, not the train length), in seconds.

Returns Zero or a negative error code.

**See also:**
`Pulses(), PulseAdd(), PulseDel(), PulseTimesSet(), SampleOutMode()`

# PulseTimesSet()

This function sets the times for a pulse in the outputs for a given state and output.

`Func PulseTimesSet(state%, out%, num%|name$, time, len);`

state% The state (set of pulses) to which this applies, from 0 to 256. Use 0 if multiple states are not in use.

out% The output to which this applies. Values of zero upwards select the corresponding DAC output, -1 selects the digital outputs, -2 the control track.

num% The number of the pulse in question, from 1 to the number of pulses-1. Zero is not usable because there are no times for the initial level.

name$ The name of the pulse to use.

time This sets the start time for the pulse, in seconds from the start of the pulse outputs.

len This sets the length of the pulse (the individual pulses for trains) in seconds. This does not affect arbitrary waveform items; use `PulseWaveSet()`.

Returns Zero or a negative error code.

**See also:**
`Pulses(), PulseAdd(), PulseDel(), PulseTimesGet(), SampleOutMode()`

# PulseType()

This function returns a code for the type of a pulse defined in the outputs.

`Func PulseType(state%, out%, num%|name$);`

state% The state (set of pulses) to which this applies, from 0 to 256. Use 0 if multiple states are not in use.

out% The output to which this applies. Values of zero upwards select the corresponding DAC output, -1 selects the digital outputs, -2 the control track.

num%   The number of the pulse in question, from 0 to the number of pulses-1.

name$   The name of the pulse in question.

Returns   A code for the type of pulse, as follows:

    0    The initial level for the output.

    1    A simple square pulse.

    2    A square pulse with varying amplitude (DACs only).

    3    A square pulse with varying duration.

    4    A ramp (DACs only).

    5    A cosine wave segment (DACs only).

    6    An arbitrary waveform (control track only).

    7    A pulse train.

    8    A marker generation item (control track only).

**See also:**
Pulses(), PulseAdd(), PulseDel(), PulseDataGet(), PulseDataSet()

# PulseVarGet()

This function retrieves the values controlling the automatic variation of a pulse in the outputs for a given state and output. Note that, though these values can be retrieved and set for all types of pulse, they will only have any effect on pulse types that allow variations; the two varying square pulses, ramps and the initial levels for DACs.

`Func PulseVarGet(state%, out%, num%|name$, &step{, &repeat{, &steps}});`

state%   The state (set of pulses) to which this applies, from 0 to 256. Use 0 if multiple states are not in use.

out%   The output to which this applies. Values of zero upwards select the corresponding DAC output, -1 selects the digital outputs, -2 the control track.

num%   The number of the pulse in question, from 0 to the number of pulses-1.

name$   The name of the pulse in question.

step   This is updated with the step value for the pulse.

repeat   This, if present, is updated with the repeat count for each step in the variation.

steps   This, if present, is updated with the total steps for the variation. Note that the number of values for the variation is steps+1 as the initial value is also used.

Returns   Zero or a negative error code.

**See also:**
Pulses(), PulseAdd(), PulseDel(), PulseVarSet(), PulseName$()

# PulseVarSet()

This function sets values controlling the automatic variation of a pulse in the outputs for a given state and output. Note that, though these values can be retrieved and set for all types of pulse, they will only have any effect on pulse types that allow variations; the two varying square pulses, ramps and the initial levels for DACs.

`Func PulseVarSet(state%, out%, num%|name$, step{, repeat{, steps}});`

state%   The state (set of pulses) to which this applies, from 0 to 256. Use 0 if multiple states are not in use.

out%   The output to which this applies. Values of zero upwards select the corresponding DAC output, -1 selects the digital outputs, -2 the control track.

num%   The number of the pulse in question, from 0 to the number of pulses-1.

name$   The name of the pulse to use.

step   This sets the step value for the pulse.

repeat   This, if present, sets the repeat count for each step in the variation.

steps    This, if present, sets the total steps for the variation. Note that the number of values for the variation is `steps+1` as the initial value is also used.

Returns  Zero or a negative error code.

**See also:**
`Pulses(), PulseAdd(), PulseDel(), PulseVarGet(), PulseName$()`

# PulseWaveformGet()

This function retrieves the waveform data values sent to a given DAC as part of an arbitrary waveform item in the pulses for a given state.

`Func PulseWaveformGet(state%{, num%}, dac%, dat%[]|dat[]);`

state%   The state (set of pulses) to which this applies, from 0 to 256. Use 0 if multiple states are not in use.

num%     The number of the waveform output within the state, from 1 to the number of waveform output items. This item can be omitted for compatibility with earlier versions of Signal, in which case waveform item number 1 will be used. This optional parameter was added in Signal version 5.00.

dac%     The DAC number for which we want data. If the DAC is not used, no data will be returned.

dat%[]   An integer array that will be filled with the data for the DAC. The values are the 16-bit integer values that would be written to the DAC. A value of -32768 corresponds to an output of -5 volts, 32767 corresponds to +5 volts (assuming 5-volt 1401 hardware). If the array is too short to hold all of the waveform, it will be filled with the initial points of the waveform. If the array is longer than the waveform, all the points will be copied and the rest of the array left unchanged.

dat[]    A real array that will be filled with the data for the DAC. The values are calibrated using the appropriate DAC scaling factors. If the array is too short to hold all of the waveform, it will be filled with the initial points of the waveform. If the array is longer than the waveform, all the points will be copied and the rest of the array left unchanged.

Returns  The number of points copied or a negative error code.

**See also:**
`Pulses(), PulseAdd(), PulseDel(), PulseWaveSet(), PulseWaveGet(), PulseWaveformSet()`

# PulseWaveformSet()

This function sets the waveform data values sent to a given DAC as part of an arbitrary waveform item in the pulses for a given state.

`Func PulseWaveformSet(state%{, num%}, dac%, dat%[]|dat[]);`

state%   The state (set of pulses) to which this applies, from 0 to 256. Use 0 if multiple states are not in use.

num%     The number of the waveform output within the state, from 1 to the number of waveform output items. This item can be omitted for compatibility with earlier versions of Signal, in which case waveform item number 1 will be used. This optional parameter was added in Signal version 5.00.

dac%     The DAC number for the data. If the DAC is not used, the function will do nothing.

dat%[]   An integer array that holds the new data for the DAC. The values are the 16-bit integer values that would be written to the DAC. A value of -32768 corresponds to an output of -5 or -10 volts as determined by your DAC hardware, similarly 32767 corresponds to +5 or +10 volts. If the array used is shorter than the waveform output points, only the earlier points in the waveform output will be changed. If the array is longer than the waveform, the extra data in the array is ignored.

dat[]    A real array that holds the new data for the DAC. The values are converted into DAC output values using the appropriate DAC scaling factors. If the array used is shorter than the waveform output points, only the earlier points in the waveform output will be changed. If the array is longer than the waveform, the extra data in the array is ignored.

Returns  The number of waveform points changed or a negative error code.

**See also:**
```
Pulses(), PulseAdd(), PulseDel(), PulseWaveSet(), PulseWaveGet(), PulseWaveformGet()
```

# PulseWaveGet()

This function retrieves the values controlling an arbitrary waveform item in the pulses for a given state. Note that the `mask%` and `rate` values are the same for all waveform items.

```
Func PulseWaveGet(state%{, num%}, &mask%, &rate, &pnts%);
```

`state%` The state (set of pulses) to which this applies, from 0 to 256. Use 0 if multiple states are not in use.

`num%`   The number of the waveform output within the state, from 1 to the number of waveform output items. This item can be omitted for compatibility with earlier versions of Signal, in which case waveform item number 1 will be used. This optional parameter was added in Signal version 5.00.

`mask%`  This is updated with the DAC mask value for the output. This has one bit set for each DAC used; bit 0 for DAC 0, bit 1 for DAC 1 and so forth. In integer values bit zero is represented by the value 1, bit one by the value 2, bit 2 by the value 4 and so forth. So if DACs 0 and 1 are used `mask%` will be 3 (1+2), if only DAC 3 is used `mask%` will be 8.

`rate`   This is updated with the output rate for the waveform data, in Hz.

`pnts%`  This is updated with the number of points of data for each DAC.

Returns  Zero or a negative error code.

**See also:**
```
Pulses(), PulseAdd(), PulseDel(), PulseWaveSet(), PulseWaveformGet(),
PulseWaveformSet()
```

# PulseWaveSet()

This function sets the values controlling an arbitrary waveform items in the pulses for a given state. Note that, as the `mask%` and `rate` values are the same for all waveform items, changing either of these in one waveform item changes all such items.

```
Func PulseWaveSet(state%{, num%}, mask%, rate, pnts%);
```

`state%` The state (set of pulses) to which this applies, from 0 to 256. Use 0 if multiple states are not in use.

`num%`   The number of the waveform output within the state, from 1 to the number of waveform output items. This item can be omitted for compatibility with earlier versions of Signal, in which case waveform item number 1 will be used. This parameter was added in Signal version 5.00.

`mask%`  This sets the DAC mask value for the output, which controls which DACs are used. This has one bit set for each DAC used; bit 0 for DAC 0, bit 1 for DAC 1 and so forth. In integer values bit zero is represented by the value 1, bit one by the value 2, bit 2 by the value 4 and so forth. So to use DACs 0 and 1 set `mask%` to 3 (1+2), to only use DAC 3 set `mask%` to 8.

`rate`   This sets the output rate for the waveform data, in Hz.

`pnts%`  This sets the number of points of data for each DAC in the mask.

Returns  Zero or a negative error code.

**See also:**
```
Pulses(), PulseAdd(), PulseDel(), PulseWaveGet(), PulseWaveformGet(),
PulseWaveformSet()
```

# Q

# 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(), DlgFont()`

# R

# Rand()

This returns pseudo-random numbers with a uniform density in a set range. The values returned are R * `scl` + `off` where R is in the range 0 up to, but not including, 1. Signal initialises the generator with a random seed based on the time. You must set the seed for a repeatable sequence. The sequence is independent of `RandExp()` and `RandNorm()`.

```
Func Rand(seed);
Func Rand({scl, off});
Func Rand(arr[]{[]}{, scl{, off}});
```

seed   If present, this is a seed for the generator in the range 0 to 1. If `seed` is outside this range, the fractional part of the number is used. If you use 0.0 as the seed, the generator is initialised with a seed based on the system time.

arr   This 1 or 2 dimensional real or integer array is filled with random numbers. If an integer array is used, the random number is truncated to an integer.

scl   This scales the random number. If omitted, it has the value 1.

off   This offsets the random number. If omitted it has the value 0.

Returns  If the first argument is not an array, the return value is a random number in the range `off` up to `off+scl`. If a seed is given, a random number is still returned. If the first argument is an array, the return value is 0.0.

**See also:**
`RandExp(), RandNorm()`

# RandExp()

This function returns pseudo-random numbers with an exponential density, suitable for generating Poisson statistics. The values returned are *R* * `mean` + `off` where *R* is a random number with the density function $p(x) = \exp(-x)$. When you start Signal, the generator is initialised with a random seed based on the time. For repeatable sequences, you must set a seed. The sequence is independent of `Rand()` and `RandNorm()`.

```
Func RandExp(seed);
Func RandExp({mean, off});
Func RandExp(arr[]{[]}{, mean{, off}});
```

seed   If present, this is a seed for the generator in the range 0 to 1. If `seed` is outside this range, the fractional part of the number is used. If you use 0.0 as the seed, the generator is initialised with a seed based on the time.

arr     This 1 or 2 dimensional real or integer array is filled with random numbers. If an integer array is used, the random number is truncated to an integer.

mean     This scales the random number. If omitted, it has the value 1.

off     This offsets the random number. If omitted it has the value 0.

Returns     If the first argument is not an array, the return value is a random number. If a seed is given, a random number is still returned. If the first argument is an array, the return value is 0.0.

The following example fills an array with event times with a mean interval t:

```
RandExp(arr[], t);           'Fill arr with event intervals
ArrIntgl(arr[]);             'convert intervals to times
```

**See also:**
Rand(), RandNorm()

# RandNorm()

This function returns pseudo-random numbers with a normal density. The values returned are $R$ * scl + off where $R$ is a random number with a normal probability density function $p(x)$ = exp(-*x*\**x*/2)/sqrt(2\*pi); this has a mean of 0 and a variance of 1. When you start Signal, the generator is initialised with a random seed based on the time. For a repeatable sequence, you must set a seed. The sequence is independent of Rand() and RandExp().

```
Func RandNorm(seed);
Func RandNorm({scl, off});
Func RandNorm(arr[]{[]}{, scl{, off}});
```

seed     If present, this is a seed for the generator in the range 0 to 1. If seed is outside this range, the fractional part of the number is used. If you use 0.0 as the seed, the generator is initialised with a seed based on the time.

arr     This 1 or 2 dimensional real or integer array is filled with random numbers. If an integer array is used, the random number is truncated to an integer.

scl     This scales the random number. If omitted, it has the value 1.

off     This offsets the random number. If omitted it has the value 0.

Returns     If the first argument is not an array, the return value is a random number. If a seed is given, a random number is still returned. If the first argument is an array, the return value is 0.0.

**See also:**
Rand(), RandExp()

# Read()

This function reads the next line from the current text view or external text file and converts the text into variables. The read starts at the beginning of the line containing the text cursor. The text cursor moves to the start of the next line after the read.

```
Func Read({&var1{, &var2{, &var3 ...}}});
```

varn     Arguments must be variables. They can be any type. One dimensional arrays are allowed. The variable type determines how to convert the string data. In a successful call, each variable matches a field in the string, and the value of the variable changes to the value found in the field. A call to Read() with no arguments skips a line.

Returns     The function returns the number of fields in the text string that were successfully extracted and returned in variables, or a negative error code. Attempts to read past the end of the file produce the end of file error code.

It is not an error to run out of data before all the variables have been updated. If this is a possibility you must check that the number of items returned matches the number you expected. An array of length n is treated as n individual values.

The source string is expected to hold data values as real numbers, integer numbers (decimal or hexadecimal introduced by 0x) and strings. Strings can be delimited by quote marks, for example `"This is a string"`, or they can be just text. If a string is not delimited, it is deemed to run to the end of the source string, so no other items can follow it. You can use `ReadSetup()` to change the characters that delimit a string and also to define hard separator characters within non-delimited strings. String delimiters are not returned as part of the string.

Normally, the fields in the source string are separated by white space (tabs and spaces) and commas. Space characters are "soft" separators. You can have any number of spaces between fields. Tabs and commas are treated as "hard" separators. Two consecutive hard separators (with or without intervening soft separators), imply a blank field. You can use `ReadSetup()` to redefine the soft and hard separators. When reading a field, the following rules are followed:

1.  Soft separator (space) characters are skipped over
2.  If the field is a string and the next character is a delimiter, it is skipped.
3.  Characters that are legal for the destination variable are extracted until a non-legal character or a separator or a required string delimiter or end of data is found. The characters read are converted into the variable type. If an error occurs in the translation, the function returns the error. Blank fields assigned to numbers are treated as 0. Blank fields assigned to strings produce empty strings.
4.  Characters are skipped until a separator character is found or end of data. If a soft separator is found, it and any further soft separators are skipped. If the next character is a hard separator it is also skipped.
5.  If there are no more variables or no more data, the process stops, else back to step 1.

The source string is expected to hold data values as:

| Type | Examples | Description |
|---|---|---|
| Real | `1.23e-23`<br>`-23`<br>`1:23:12:58.23`<br>`12:34.56` | A number with an optional decimal point and an optional exponent. The number can start with a minus sign.<br><br>From version 6.04 you can read a time as `days:hours:minutes:seconds`. You may omit the days, or days and hours. The time is converted to seconds. Times of more than 100 years are not recognised. |
| Integer | `123, -1234567`<br>`0x1234abcd` | A 32-bit integer value formatted as an optional minus sign followed by a list of decimal digits. It also accepts a hexadecimal value, being 0x followed by 1 to 8 hexadecimal digits (0-9, a-f, A-F). |
| String | `Some text`<br>`"Quoted text"` | Strings can be delimited by double quote marks, which allows the strings to hold separators and to be followed by other fields. If a string is not delimited, it is deemed to run to the end of the source string, so no other items can follow it unless you set a hard string separator with `ReadSetup()`. You can use `ReadSetup()` to change the characters that delimit a string and also to define hard separator characters for non-delimited strings. String delimiters are not returned as part of the string. |

## Reading exact contents of text file

If you want to use `Read()` to fill a string with the line by line contents of a text file, including leading white space, you should call `ReadSetup("","")` before using `Read()`. This cancels all soft separators, which allows leading white space (spaces and tab characters) to be read.

### 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);
```

is equivalent to:

```
n := 7;
fred$ := "This is text";
jim[1] := 2; jim[2] := 3; sam := 4; dick% := 0; tom% := 4;
sally$ := "Text too 3 4 5"
a, b and c are not changed
```

### See also:

EditCopy(), FileOpen(), ReadSetup(), ReadStr(), Selection$()

# ReadSetup()

This sets the separators and delimiters used by `Read()` and `ReadStr()` to convert text into numbers and strings. You can also set string delimiters and set a string separator.

```
Proc ReadSetup({hard${, soft${, sDel${, eDel${, sSep$}}}}});
```

hard$   The characters to use as hard separators between all fields. If this is omitted or the string is empty, the standard hard separators of comma and tab are used.

soft$   The characters to use as soft separators. If this is omitted, the space character is set as a soft separator. If `soft$` is empty, no soft separators are used.

sDel$   The characters that delimit the start of a string. If omitted, a double quote is used. If empty, no delimiter is set. Delimiters are not returned in the string.

eDel$   The characters that delimit the end of a string. If omitted, a double quote is used. If empty, no delimiter is set. If `sDel$` and `eDel$` are the same length, only the end delimiter character that matches the start delimiter position is used. For example, to delimit strings with `<text>` or `'text'` set `sDel$` to `"<'"` and `eDel$` to `">'"`. You can repeat a character to force different lengths.

sSep$   The list of hard separator characters for strings that have no start delimiter. For example, setting `"|"` lets you read `one|two|three` into three separate strings.

**See also:**
Read(), ReadStr(), Val()

# ReadStr()

This function extracts data fields from a string and converts them into variables.

```
Func ReadStr(text$, &var1{, &var2{, &var3...}});
```

text$   The string used as a source of data.

var   The arguments must all be variables. The variables can be of any type, and can be one dimensional arrays. The type of each variable determines how the function tries to extract data from the string. See `Read()` for details.

Returns   The function returns the number of fields in the text string that were successfully extracted and returned in variables, or a negative error code.

It is not an error to run out of data before all the variables have been updated. If this is a possibility you must check the returned value. If an array is passed in, it is treated as though it was the number of individual values held in the array.

**See also:**
Read(), ReadSetup(), Val()

# 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:**
```
Abs(), ATan(), Cos(), Exp(), Frac(), Ln(), Log(), Max(), Min(), Pow(), Rand(), Sin(),
Sqrt(), Tan(), Trunc()
```

# S

# Sample....()

# SampleAbort()

This function cancels sampling and deletes the views that were being sampled and any associated result and cursor views. If a memory view derived from the data has been saved, the saved file remains. It is equivalent to the Abort button on the floating sampling control window.

```
Func SampleAbort({noask%});
```

noask% If provided and non-zero, this causes the normal querying of the user about this potentially destructive operation to be bypassed.

Returns 0 if sampling was aborted, or a negative error code.

**See also:**
```
SampleReset(), SampleStart(), SampleStop(), SampleStatus()
```

# SampleAbsLevel()

This function sets and gets the absolute levels flag as seen in the sampling configuration dialog Outputs page.

```
Func SampleAbsLevel({new});
```

new  If present, this sets the new absolute levels flag if non-zero, clears the flag if zero.

Returns The absolute levels flag from the configuration at the time of the call.

**See also:**
```
SampleClear(), Pulses()
```

# SampleAbsTime()

This function sets and gets the absolute times flag as seen in the sampling configuration dialog Outputs page. This function was added to Signal in version 7.01.

```
Func SampleAbsTime({new});
```

new  If present, this sets the new absolute times flag if non-zero, clears the flag if zero.

Returns The absolute times flag from the configuration at the time of the call.

**See also:**
```
SampleClear(), Pulses()
```

# SampleAccept()

This flags the current frame 0 data in a sampling document to be accepted or rejected.

```
Func SampleAccept({yes%});
```

`yes%`    If this is zero, the frame is rejected, otherwise the frame is written to disk. This is equivalent to the Accept check box in the sampling control dialog.

Returns  0 if sweep was written successfully or a negative error code.

**See also:**
```
SamplePause(), SampleStatus(), SampleWrite()
```

# SampleArtefactGet()

This command returns the parameters used in automatic artefact rejection during sampling. Artefact rejection consists of testing all points within a given time range and rejecting or tagging frames where the points at the ADC limit exceed a set threshold. This function normally operates on the stored sampling configuration but if used during sampling it operates upon the on-going sampling.

```
Func SampleArtefactGet(&mode%, &start, &end, &per{, &lev});
```

`mode%`   Returned holding the artefact rejection mode: 0 for none, 1 for Tag and 2 for Reject.

`start`   Returned holding the start time for the search for out-of-range points.

`end`     Returned holding the end time for the search for out-of-range points.

`per`     Returned holding the percentage of out-of-range points that can be tolerated, frames with more than this are deemed to contain artefacts.

`lev`     Returned holding the percentage of the ADC range (from zero to full scale) above or below which will be considered an artefact.

Returns  Zero or a negative error code.

**See also:**
```
SampleArtefactSet(), SampleAccept(), FrameTag(), SamplePortFull()
```

# SampleArtefactSet()

This command sets the parameters used in automatic artefact rejection during sampling. Artefact rejection consists of testing all points within a given time range and rejecting or tagging frames where the points at the ADC limit exceed a set threshold. This function normally operates on the stored sampling configuration but if used during sampling it operates upon the on-going sampling.

```
Func SampleArtefactSet(mode%, start, end, per{, lev});
```

`mode%`   Sets the artefact rejection mode: 0 for none, 1 for Tag and 2 for Reject.

`start`   The start time for the search for out-of-range points.

`end`     The end time for the search for out-of-range points.

`per`     The percentage of out-of-range points (value limited to 0 to 100) that can be tolerated, frames with more than this are deemed to contain artefacts.

`lev`     The percentage of the ADC range (from zero to full scale, value limited to 1 to 100) above or below which will be considered an artefact. The default is 100.

Returns  Zero or a negative error code.

**See also:**
```
SampleArtefactGet(), SampleAccept(), FrameTag(), SamplePortFull()
```

# SampleAutoFile()

This gets or sets the flag for file auto-filing as seen in the sampling configuration dialog.

```
Func SampleAutoFile({yes%});
```

yes%    If present and non-zero, this turns on automatic filing of data when sampling finishes.  If zero or missing it turns automatic filing off.

Returns  The automatic filing flag at the time of the function call.

**See also:**
```
SampleAutoName$(), FileNew()
```

# SampleAutoName$()

This gets or sets the template for file auto-naming as in the sampling configuration dialog. The path used for auto-naming can be set using `FilePathSet()` and read using `FilePath$()`.

```
Func SampleAutoName$({name$});
```

name$   If present, this sets the new template string for file auto-naming (this should not be more than 22 characters long), or turns off auto-naming if it is a blank string. See the sampling configuration documentation for details on the template string.

Returns  The auto-naming template at the time of the function call.

**See also:**
```
FilePath$, FilePathSet(), SampleAutoFile(), FileNew()
```

# SampleAutoOpt()

This function sets and gets the automatic Y axis optimisation flag as seen in the sampling configuration. This function was added to Signal in version 7.05.

```
Func SampleAutoOpt({new});
```

new     If present, this sets the new automatic optimisation flag if non-zero, clears the flag if zero.

Returns  The automatic optimisation flag from the configuration at the time of the call.

**See also:**
```
SampleClear(), SampleAutoFile()
```

# SampleAuxStateNew()

This function is used to add an auxiliary states device to the list associated with the current sampling configuration.

```
Func SampleAuxStateNew(type%{, label$});
```

type%   Selects the type of auxiliary device wanted; 1=Magstim, 2=CED 3304 current stimulator, 3=MagPro, 4=PowerMAG, 5=DuoMAG, 6=DS8R.

label$  If provided, this sets the device label for the new auxiliary states device.

Returns  The  new number of auxiliary states devices in the list if the function succeeded or zero if the function failed.

**See also:**
```
SampleAuxStateRemove(), SampleAuxStateParX(), SampleAuxStateValX(), SampleStates(),
SampleState()
```

# SampleAuxStatePar$()

This function is used to get or set string values for a specified auxiliary device. The meaning and use of these parameters may vary according to the type of auxiliary states device. At the time of writing the auxiliary devices supported are the Magstim, MagPro, PowerMAG, DuoMAG, DS8R and the CED 3304. The meaning of the `num%` parameter in this function can depend entirely upon the type of auxiliary states device.

```
Func SampleAuxStatePar$(dev%|lab$, num%{, val$});
```

`dev%`  The number of the auxiliary device to be used, the first auxiliary device in the list of those used with the current sampling configuration list is number 1.

`lab$`  The label of the auxiliary device to be used, the device is found by searching for a matching label using a case-insensitive comparison, if there are multiple devices with the same label the first match found will be used.

`num%`  Selects the string to read or set:

  0   The name of this auxiliary states device. You cannot set this string value.

  1   The device label.

`val$`  If present, this sets the new value of the selected string, otherwise it will be unchanged. It is not possible to set a new value for parameter zero, this is set by the type of auxiliary device for example "Magstim".

Returns  The string value selected at the time of the function call.

**See also:**
```
SampleAuxStateParX(), SampleAuxStateValX(), SampleAuxStateNew(),
SampleAuxStateRemove(), SampleStates(), SampleState()
```

# SampleAuxStateParam()

**Deprecated**. This is provided for backwards compatibility with old scripts and only gives access to the first auxiliary states device. Use `SampleAuxStateParX()` and `SampleAuxStateValX()` in new scripts.

This function is used to get or set parameters for the first auxiliary states device associated with the current sampling configuration. The meaning and use of these parameters varies according to the type of auxiliary states device. At the time of writing the auxiliary devices supported are the Magstim, MagPro, PowerMAG, DuoMAG, DS8R and the CED 3304. The meaning of the `num%` parameter in this function will depend entirely upon the type of auxiliary states device installed. It is therefore very important that the value returned by `SampleAuxStateParam(0)` is checked before using these functions (for a Magstim you should call `SampleAuxStateParam(1)` as well to check the Magstim type) and that nothing is done by the script if the returned values do not match the supported devices.

```
Func SampleAuxStateParam(num%{, val});
```

`num%`  Selects the parameter to read or set. The meaning of all parameters except 0 depends on the auxiliary device type.

  0   The type of auxiliary states device support loaded. -1 = no device, 1=Magstim, 2=CED 3304 current stimulator, 3=MagPro, 4=PowerMAG, 5=DuoMAG, 6=DS8R. You cannot set this parameter.

  **For a Magstim only:**

  1   The device in use. -1 = Do not use, 0 = 200, 1 = BiStim, 2 = Rapid, 3 = dual 200.

  2   The device flags. This is the sum of 1 for the Rapid coil interlock ignore, 2 for BiStim high-resolution timing, 4 for Rapid single pulse mode, 8 for assume BiStim independent triggers, 16 to prevent generation of textual information for the sampling window title and 128 to allow simulated operation without Magstim hardware being present. Not all of these values are meaningful with any particular Magstim type; your script code needs to be carefully written.

  3   The serial line port (COM port) used to control the Magstim, from 1 to 19.

  4   The second serial line port for dual 200 mode, from 1 to 19.

  5   The digital output used to trigger the Magstim (for Rapid only) from 0 to 7 or -1 for not set. Setting this to -1 is not a good idea as it defeats the power safety checks and will cause warning messages.

  6   The current Magstim device status, if used while sampling is in progress, else zero. The bottom 8 bits of the value returned are the standard Magstim device status byte, the next 8 bits are the Rapid status for a Rapid device or the device status for the second Magstim for dual 200s. Consult your Magstim

documentation for the meaning of this information. Needless to say, this parameter value cannot be set.

**For a MagPro only:**

1   The device in use. -1 = Do not use, 0 = MagPro R30, 1 = MagPro R30 + MagOption, 2 =MagPro X100, 3 = MagPro X100 + MagOption.

2   The device flags. This is the sum of 1 for assume MagPro software version 7.1 or later, 16 to prevent generation of textual information for the sampling window title.

3   The serial line port (COM port) used to control the MagPro, from 1 to 19.

**For a PowerMAG only:**

1   The device in use. -1 = Do not use, 0 = PowerMAG pp-TMS.

2   The device flags. This is the sum of 16 to prevent generation of textual information for the sampling window title.

3   The serial line port (COM port) used to control the PowerMAG, from 1 to 19.

**For a DuoMAG only:**

1   The device in use. -1 = Do not use, 0 = XT or MP device, 1 = Dual MP devices.

2   The device flags. This is the sum of 8 for independent triggers, 16 to prevent generation of textual information for the sampling window title.

**For a DS8R only:**

1   The device in use. -1 = Do not use, 0 = DS8R.

2   The device flags. This is the sum of 16 to prevent generation of textual information for the sampling window title.

**For a CED 3304 only:**

1   CED 3304 in use. -1=Do not use, 0=Use CED 3304.

2   The device flags. This is 1 for the high level trigger, 0 for low trigger. Add 2 to prevent generation of textual information for the sampling window title.

3   The serial line port (COM port) used to control the 3304, from 1 to 19.

4   The range switch setting, from 0 (10 microamps) to 3 (10 milliamps).

`val`   If present, this sets the new value of the selected parameter, otherwise it will be unchanged. It is not possible to set a new value for parameters zero, this is set by the type of auxiliary device support installed.

Returns   The parameter value selected at the time of the function call.

**Warning**

Many auxiliary states devices, in particular transcranial magnetic stimulators such as the Magstim, are potentially dangerous devices. If you use these script functions carelessly, they could be set to deliver dangerously high levels of stimulation or magnetic field. CED cannot accept any responsibility whatsoever for any harm or damage resulting - it is up to you to write your script carefully and make sure that you are programming the device you think you are using, and that the parameter values used are correct.

**See also:**
`SampleAuxStateParX(), SampleAuxStateValX(), SampleAuxStateNew(),`
`SampleAuxStateRemove(), SampleStates(), SampleState()`

# SampleAuxStateParX()

This function is used to find the number of auxiliary states devices in use and to get or set parameters for a specified auxiliary device. The meaning and use of these parameters varies according to the type of auxiliary states device. At the time of writing the auxiliary devices supported are the Magstim, MagPro, PowerMAG, DuoMAG, DS8R and the CED 3304. The meaning of the `num%` parameter in this function will depend entirely upon the type of auxiliary states device. It is therefore very important that the value returned by `SampleAuxStateParX(dev%, 0)` is checked before using these functions (for a Magstim you should call `SampleAuxStateParX(dev%, 1)` as well to check the Magstim type) and that nothing is done by the script if the returned values do not match the supported devices.

This function was added in Signal version 7.01 to allow access to multiple auxiliary states devices; it replaces `SampleAuxStateParam()` which is now deprecated.

`Func SampleAuxStateParX();`

Returns This form of the function returns the number of auxiliary states devices associated with the current sampling configuration.

`Func SampleAuxStateParX(dev%|lab$, num%{, val});`

`dev%`  The number of the auxiliary device to be used, the first auxiliary device (in the list of those used with the current sampling configuration) is number 1.

`lab$`  The label of the auxiliary device to be used, the device is found by searching for a matching label using a case-insensitive comparison, if there are multiple devices with the same label the first match found will be used.

`num%`  Selects the parameter to read or set. The meaning of all parameters except 0 depends on the auxiliary device type.

0 The type of this auxiliary states device. -1 = no device (invalid `dev%` value), 1=Magstim, 2=CED 3304 current stimulator, 3=MagPro, 4=PowerMAG, 5=DuoMAG, 6=DS8R, 7=Neuro-MS. You cannot set this parameter.

**For a Magstim only:**

1 The device in use. -1 = Do not use, 0 = 200, 1 = BiStim, 2 = Rapid, 3 = dual 200.
2 The device flags. This is the sum of 1 for the Rapid coil interlock ignore, 2 for BiStim high-resolution timing, 4 for Rapid single pulse mode, 8 for assume BiStim independent triggers, 16 to prevent generation of textual information for the sampling window title and 128 to allow simulated operation without Magstim hardware being present. Not all of these values are meaningful with any particular Magstim type; your script code needs to be carefully written.
3 The serial line port (COM port) used to control the Magstim, from 1 to 19.
4 The second serial line port for dual 200 mode, from 1 to 19.
5 The digital output used to trigger the Magstim (for Rapid only) from 0 to 7 or -1 for not set. Setting this to -1 is not a good idea as it defeats the power safety checks and will cause warning messages.
6 The current Magstim device status, if used while sampling is in progress, else zero. The bottom 8 bits of the value returned are the standard Magstim device status byte, the next 8 bits are the Rapid status for a Rapid device or the device status for the second Magstim for dual 200s. Consult your Magstim documentation for the meaning of this information. Needless to say, this parameter value cannot be set.
7 The flag for power as a percentage of the resting motor threshold (RMT) value, 0 for not set, 1 for set.
8 The current resting motor threshold (RMT) value as a percentage of the device power level, from 0 to 100 percent (or 110 percent for Rapid 2 single pulse mode).

**For a MagPro only:**

1 The device in use. -1 = Do not use, 0 = MagPro R30, 1 = MagPro R30 + MagOption, 2 =MagPro X100, 3 = MagPro X100 + MagOption.
2 The device flags. This is the sum of 1 for assume MagPro software version 7.1 or later, 16 to prevent generation of textual information for the sampling window title.
3 The serial line port (COM port) used to control the MagPro, from 1 to 19.
4 The flag for power as a percentage of the resting motor threshold (RMT) value, 0 for not set, 1 for set.
5 The current resting motor threshold (RMT) value as a percentage of the device power level, from 0 to 100 percent.
6 The current device status, this is currently either 0 for device not ready, or 1 for device ready. Needless to say, this parameter value cannot be set.

**For a PowerMAG only:**

1 The device in use. -1 = Do not use, 0 = PowerMAG pp-TMS.
2 The device flags. This is the sum of 16 to prevent generation of textual information for the sampling window title.
3 The serial line port (COM port) used to control the PowerMAG, from 1 to 19.
4 The flag for power as a percentage of the resting motor threshold (RMT) value, 0 for not set, 1 for set.

5    The current resting motor threshold (RMT) value as a percentage of the device power level, from 0 to 100 percent.

**For a DuoMAG only:**

1    The device in use. -1 = Do not use, 0 = XT or MP device, 1 = Dual MP devices.

2    The device flags. This is the sum of 8 for independent triggers, 16 to prevent generation of textual information for the sampling window title.

3    The flag for power as a percentage of the resting motor threshold (RMT) value, 0 for not set, 1 for set.

4    The current resting motor threshold (RMT) value as a percentage of the device power level, from 0 to 100 percent.

**For a DS8R only:**

1    The device in use. -1 = Do not use, 0 = DS8R.

2    The device flags. This is the sum of 16 to prevent generation of textual information for the sampling window title.

**For a CED 3304 only:**

1    CED 3304 in use. -1=Do not use, 0=Use CED 3304.

2    The device flags. This is 1 for the high level trigger, 0 for low trigger. Add 2 to prevent generation of textual information for the sampling window title.

3    The serial line port (COM port) used to control the 3304, from 1 to 19.

4    The range switch setting, from 0 (10 microamps) to 3 (10 milliamps).

**For a Neuro-MS only:**

1    The device in use. -1 = Do not use, 0 = Neuro-MS device, 1 = Paired Neuro-MS devices.

2    The device flags. This is the sum of 16 to prevent generation of textual information for the sampling window title.

3    The flag for power as a percentage of the resting motor threshold (RMT) value, 0 for not set, 1 for set.

4    The current resting motor threshold (RMT) value as a percentage of the device power level, from 0 to 100 percent.

val    If present, this sets the new value of the selected parameter, otherwise it will be unchanged. It is not possible to set a new value for parameters zero, this is set by the type of auxiliary device support installed.

Returns  The parameter value selected at the time of the function call.

**Warning**

Many auxiliary states devices, in particular transcranial magnetic stimulators such as the Magstim, are potentially dangerous devices. If you use these script functions carelessly, they could be set to deliver dangerously high levels of stimulation or magnetic field. CED cannot accept any responsibility whatsoever for any harm or damage resulting - it is up to you to write your script carefully and make sure that you are programming the device you think you are using, and that the parameter values used are correct.

**See also:**
```
SampleAuxStateValX(), SampleAuxStatePar$(), SampleAuxStateNew(),
SampleAuxStateRemove(), SampleStates(), SampleState()
```

# SampleAuxStateRemove()

This function is used to remove an auxiliary states device from the list associated with the current sampling configuration.

```
Func SampleAuxStateRemove(dev%|lab$);
```

dev%    The number of the auxiliary device to be removed, the first auxiliary device in the list of those used with the current sampling configuration list is number 1.

lab$    The label of the auxiliary device to be removed, the device is found by searching for a matching label using a case-insensitive comparison, if there are multiple devices with the same label the first match found will be removed.

Returns  Non-zero if the function succeeded and the device was removed, zero if it failed.

**See also:**
`SampleAuxStateNew()`, `SampleAuxStateParX()`, `SampleAuxStateValX()`, `SampleStates()`,
`SampleState()`

# SampleAuxStateValue()

**Deprecated**. This is provided for backwards compatibility with old scripts and only gives access to the first auxiliary states device. Use `SampleAuxStateParX()` and `SampleAuxStateValX()` in new scripts.

This function is used to get or set auxiliary states device settings for individual states for the first auxiliary states device associated with the current sampling configuration. The meaning and use of these settings varies according to the type of auxiliary states device, it is vitally important that you use `SampleAuxStateParam(0)` in your script to check that the correct auxiliary states device is installed. You will also need to be very familiar with the device(s) in question as in many cases certain modes or options are not available in all circumstances. At the time of writing the auxiliary devices supported are the Magstim, MagPro, PowerMAG, DuoMAG, DS8R and the CED 3304.

```
Func SampleAuxStateValue(state%, num%{, val});
```

`state%`  The state number for which the settings are being read or written, from zero to the number of extra states enabled.

`num%`  This selects the setting that will be read or set. The meaning and use of all settings varies with the type of auxiliary states device in use.

**For a Magstim only:**
0    The state flags. This is 1 if manual control is selected, otherwise 0.
1    The power level from 0 to 100 percent (or 110 percent for Rapid 2 single pulse mode).
2    The secondary power level for BiStim and dual 200 devices.
3    The pulse interval for BiStim devices.
4    The pulse frequency in Hz for Rapid devices.
5    The number of pulses for Rapid devices.

**For a MagPro only:**
0    The state flags. This is the sum of 1 if manual control is selected and 2 if reversed pulses are to be generated.
1    The power A level from 0 to 100 percent.
2    The power B level from 0 to 100 percent.
3    The pulse interval in milliseconds from 1 to 3000.
4    The power B/A value for Twin mode from 0.2 to 5.
5    The number of pulses for biphasic bursts, from 2 to 5.
6    The device mode. This is 0 for Standard, 1 for Power, 2 for Twin and 3 for Dual mode.
7    The waveform type. This is 0 for Monophasic, 1 for Biphasic, 2 for Halfsine and 3 for Biphasic burst.
8    The charge delay time, in milliseconds.

**For a PowerMAG only:**
0    The state flags. These are currently unused and will always be zero.
1    The S1 power level from 0 to 100 percent.
2    The S2 power level from 0 to 100 percent.
3    The number of sets of pulse train information defined for the state.

**For a DuoMAG only:**
0    The state flags. This is 1 if manual control is selected, otherwise 0. Add 2 to enable manual triggering mode.
1    The stimulator power level from 0 to 100 percent.
2    The TTL OUT pulse delay, in milliseconds.
3    The recharge delay, in milliseconds.
4    The power level for the secondary stimulator, from 0 to 100 percent.

5    The recharge delay for the secondary stimulator, in milliseconds.

**For a DS8R only:**

0    The state flags. This the sum of 1 for the stimulator being disabled, 2 for external control of the stimulator current, 4 to enable warning of out of compliance stimulation and 8 for manual control of the stimulator.

1    The stimulation current in milliamps, from 2 to 1000.

2    The monophasic stimulation selection; 1 for monophasic, zero for biphasic.

3    The stimulation duration in microseconds, from 50 to 2000.

4    The dwell time duration in microseconds, from 1 to 990.

5    The stimulation polarity option; 1 for positive, 2 for negative and 3 for alternating.

6    The recovery percentage, from 10 to 100.

**For a CED 3304 only:**

0    The state flags. Unused with the 3304 and read as 0.

1    The current in uA from zero to the maximum allowed by the range setting.

val    If present, this sets the new value of the selected setting, otherwise it will be unchanged.

Returns  The setting value selected at the time of the function call.

### Warning

Many auxiliary states devices, in particular transcranial magnetic stimulators such as the Magstim, are potentially dangerous devices. If you use these script functions carelessly, they could be set to deliver dangerously high levels of stimulation or magnetic field. CED cannot accept any responsibility whatsoever for any harm or damage resulting - it is up to you to write your script carefully and make sure that you are programming the device you think you are using, and that the parameter values used are correct.

### See also:
`SampleAuxStateParX(), SampleAuxStateValX(), SampleAuxStateNew(),`
`SampleAuxStateRemove(), SampleStates(), SampleState()`

# SampleAuxStateValX()

This function is used to get or set auxiliary states device settings for individual states from an auxiliary states device associated with the current sampling configuration. The meaning and use of these settings varies according to the type of auxiliary states device, it is vitally important that you use `SampleAuxStateParX(dev%, 0)` in your script to check that the correct auxiliary states device is installed. You will also need to be very familiar with the device(s) in question as in many cases certain modes or options are not available in all circumstances. At the time of writing the auxiliary devices supported are the Magstim, MagPro, PowerMAG, DuoMAG, DS8R and the CED 3304.

This function was added in Signal version 7.01 to allow access to multiple auxiliary states devices; it replaces `SampleAuxStateValue()` which is now deprecated.

`Func SampleAuxStateValX(dev%|lab$, state%, num%{, val});`

dev%    The number of the auxiliary device to be used, the first auxiliary device (in the list of those used with the current sampling configuration) is number 1.

lab$    The label of the auxiliary device to be used, the device is found by searching for a matching label using a case-insensitive comparison, if there are multiple devices with the same label the first match found will be used.

state%  The state number for which the settings are being read or written, from zero to the number of extra states enabled.

num%    This selects the setting that will be read or set. The meaning and use of all settings varies with the type of auxiliary states device in use.

**For a Magstim only:**

0    The state flags. This is 1 if manual control is selected, otherwise 0.

1     The power level from 0 to 100 percent (or 110 percent for Rapid 2 single pulse mode). If power is specified as a percentage of the resting motor threshold, this value is 0 to 100 percent of the RMT value.

2     The secondary power level for BiStim and dual 200 devices. If power is specified as a percentage of the resting motor threshold, this value is 0 to 100 percent of the RMT value.

3     The pulse interval for BiStim devices.

4     The pulse frequency in Hz for Rapid devices.

5     The number of pulses for Rapid devices.

**For a MagPro only:**

0     The state flags. This is the sum of 1 if manual control is selected and 2 if reversed pulses are to be generated.

1     The power A level from 0 to 100 percent. If power is specified as a percentage of the resting motor threshold, this value is 0 to 100 percent of the RMT value.

2     The power B level from 0 to 100 percent. If power is specified as a percentage of the resting motor threshold, this value is 0 to 100 percent of the RMT value.

3     The pulse interval in milliseconds from 1 to 3000.

4     The power B/A value for Twin mode from 0.2 to 5.

5     The number of pulses for biphasic bursts, from 2 to 5.

6     The device mode. This is 0 for Standard, 1 for Power, 2 for Twin and 3 for Dual mode.

7     The waveform type. This is 0 for Monophasic, 1 for Biphasic, 2 for Halfsine and 3 for Biphasic burst.

8     The charge delay time, in milliseconds.

**For a PowerMAG only:**

0     The state flags. These are currently unused and will always be zero.

1     The S1 power level from 0 to 100 percent. If power is specified as a percentage of the resting motor threshold, this value is 0 to 100 percent of the RMT value.

2     The S2 power level from 0 to 100 percent. If power is specified as a percentage of the resting motor threshold, this value is 0 to 100 percent of the RMT value.

3     The number of sets of pulse train information defined for the state.

**For a DuoMAG only:**

0     The state flags. This is 1 if manual control is selected, otherwise 0. Add 2 to enable manual triggering mode.

1     The stimulator power level from 0 to 100 percent. If power is specified as a percentage of the resting motor threshold, this value is 0 to 100 percent of the RMT value.

2     The TTL OUT pulse delay, in milliseconds.

3     The recharge delay, in milliseconds.

4     The power level for the secondary stimulator, from 0 to 100 percent. If power is specified as a percentage of the resting motor threshold, this value is 0 to 100 percent of the RMT value.

5     The recharge delay for the secondary stimulator, in milliseconds.

**For a DS8R only:**

0     The state flags. This the sum of 1 for the stimulator being disabled, 2 for external control of the stimulator current, 4 to enable warning of out of compliance stimulation and 8 for manual control of the stimulator.

1     The stimulation current in milliamps, from 2 to 1000.

2     The monophasic stimulation selection; 1 for monophasic, zero for biphasic.

3     The stimulation duration in microseconds, from 50 to 2000.

4     The dwell time duration in microseconds, from 1 to 990.

5     The stimulation polarity option; 1 for positive, 2 for negative and 3 for alternating.

6     The recovery percentage, from 10 to 100.

**For a Neuro-MS only:**

0     The state flags. This is 1 if manual control is selected, otherwise 0.

1     The stimulator power level from 0 to 100 percent. If power is specified as a percentage of the resting motor threshold, this value is 0 to 100 percent of the RMT value.

2 The secondary pulse delay, in milliseconds.

3 The power level for the secondary stimulator, from 0 to 100 percent. If power is specified as a percentage of the resting motor threshold, this value is 0 to 100 percent of the RMT value.

**For a CED 3304 only:**

0 The state flags. Unused with the 3304 and read as 0.

1 The stimulation current in uA from zero to the maximum allowed by the range setting.

val     If present, this sets the new value of the selected setting, otherwise it will be unchanged.

Returns  The setting value selected at the time of the function call.

**Warning**

Many auxiliary states devices, in particular transcranial magnetic stimulators such as the Magstim, are potentially dangerous devices. If you use these script functions carelessly, they could be set to deliver dangerously high levels of stimulation or magnetic field. CED cannot accept any responsibility whatsoever for any harm or damage resulting - it is up to you to write your script carefully and make sure that you are programming the device you think you are using, and that the parameter values used are correct.

**See also:**
```
SampleAuxStateParX(), SampleAuxStatePar$(), SampleAuxStateNew(),
SampleAuxStateRemove(), SampleStates(), SampleState()
```

# SampleBar()

This function gives you access to the Sample toolbar. The format of strings passed by this command is the button label (up to 8 characters), followed by a vertical bar, followed by the full path name to a sampling configuration file, including the `.sgcx` (`.sgc` for older files) file extension, followed by a vertical bar, then a comment to display when the mouse pointer is over the button. If you call the command with no arguments it returns the number of buttons in the toolbar.

```
Func SampleBar({n%{, &get$}});
Func SampleBar(set$);
```

n%      If set to -1, `get$` must be omitted, all buttons are cleared and the function returns 0. When set to the number of a button (the first button is 0), `get$` is as described above. In this case, the function returns -1 if the button does not exist, 0 if it exists and is the last button, and 1 if higher-numbered buttons exist.

set$    The string passed in should have the format described above. The function returns the new number of buttons or -1 if all buttons are already used.

Returns  See the descriptions above. Negative return values indicate an error.

For example, the following code clears the script bar and sets two buttons:

```
SampleBar(-1);    'clear all buttons
SampleBar("Fast|C:\\Signal3\\Fast.sgc|Fast 4 channel sampling");
SampleBar("Faster|C:\\Signal3\\FastXX.sgc|Very fast sampling");
```

**See also:**
```
App()
```

# SampleBurst()

This function gets or sets the burst mode sampling flag as seen in the sampling configuration dialog.

```
Func SampleBurst({bMode%});
```

bMode%  If present and non-zero this turns burst mode on in the sampling configuration. Burst mode is often to be preferred, as the actual sampling rate used is more likely to match the preferred rate set.

Returns  1 if burst mode is on, 0 if it is off.

**See also:**
```
SampleClear(), SampleRate(), SamplePoints(), SampleTrigger(), SamplePorts()
```

# SampleClamp()

This function gets or sets the overall clamping information and clamping sets in the sampling configuration. The first two forms of the command are used to set up clamping options, the third form reads settings back.

```
Func SampleClamp(0, meas%, flags%);
Func SampleClamp(set%, type%, stim%, resp%, dac%);
Func SampleClamp(set%, opt%);
```

set%    The clamping set to work with, from 1 to 8, or 0 to operate on the overall clamping information.

meas%   The state to use for resistance measurements (when `set%` is zero), from 0 to 256.

flags%  The clamping option flags (when `set%` is zero), currently this is unused and should be set to zero.

type%   The type of clamping used in this set (`set%` is the clamping set from 1 to 8). The possible clamping types are:

　　0　Clamping is disabled.
　　1　Whole cell voltage clamp.
　　2　Whole cell current clamp.
　　3　Single-channel voltage clamp.
　　4　Single-channel current clamp.

stim%   The stimulus channel for this clamping set (`set%` is the clamping set from 1 to 8), from 1 upwards. No error will be generated by selecting a channel that will not be created by this sampling configuration or which has the wrong channel units, but if these are not correct when the sampling configuration is used then it will not run.

resp%   The response channel for this clamping set (`set%` is the clamping set from 1 to 8), from 1 upwards. No error will be generated by selecting a channel that will not be created by this sampling configuration or which has the wrong channel units, but if these are not correct when the sampling configuration is used then it will not run.

dac%    The control DAC number used in this clamping set (`set%` is the clamping set from 1 to 8), from 0 to 7. No error will be generated by selecting a DAC that is not used in this sampling configuration, or which has the wrong units, but if these are not correct when the sampling configuration is used then it will not run.

opt%    This can be set to the following negative values which cause the function to return the following values. If `set%` is zero the possible `opt%` values are:

　　-1　The state that is used for resistance measurement is returned.
　　-2　The clamping option flags are returned.
　　-3　This special option resets all of the clamping information and returns 0.
　　-4　The number of clamping sets available (currently 8) is returned.

　　If `set%` is 1 to 8 the possible `opt%` values are:

　　-1　The clamping type in this clamping set is returned.
　　-2　The stimulus channel in this clamping set is returned.
　　-3　The response channel in this clamping set is returned.
　　-4　The control DAC used in this clamping set is returned.

Returns  0, the requested value or a negative error code.

**See also:**
```
SamplePorts(), SamplePortUnits$(), SampleDacMask(), SampleDacUnits$(),
SampleClampHP()
```

# SampleClampHP()

This function gets or sets the current holding potential in use with a clamping set. It cannot be used before sampling to preset the holding potential - that is set by the initial levels for the various DACs - but only while sampling is in progress to dynamically access the potentials in use.

```
Func SampleClampHP(set%{, new});
```

set%    The clamping set to work with, from 1 to 8.

new    If provided, this sets the new holding potential. Note that this value and the function result are in whatever units are in use for the relevant control DAC, which will normally be millivolts, but not necessarily.

Returns  the holding potential at the time of the call.

**See also:**
`SamplePorts(), SamplePortUnits$(), SampleDacMask(), SampleDacUnits$(), SampleClamp()`

# SampleClear()

This procedure resets the contents of the sampling configuration dialog to a standard state.  You can use the sampling configuration commands to get or change values in the sampling configuration. There is a full list of the sampling configuration commands in the *Commands by function* section.

```
Proc SampleClear();
```

# SampleConfig$()

This function gets information about the current sampling configuration, currently the only information available is the full name of the file from which the the current configuration was loaded. Sampling can change the configuration, so the current state may not exactly match that stored in the configuration file. The file may have a .sgc or .sgcx file extension. Users can see this path and file name in the title bar of the Sampling Configuration dialog.

```
Func SampleConfig$();
```

Returns The name of the file from which the the current configuration was loaded or an empty string if the name is unknown.

**See also:**
`FileExportAs(), FileOpen(), SampleSequencer$()`

# SampleDacFull()

This function gets the full-scale value used to scale values written to the DACs from the sampling configuration and optionally sets it to a new value.

```
Func SampleDacFull(port{, new});
```

port   The DAC number, from 0 to 7. Sequencer outputs use the DAC 0 settings for all DAC calibration.

new    If present, sets the value in the units for this DAC corresponding to a full-scale value. This value is used throughout Signal to calibrate DAC values.

Returns  The DAC full-scale value before the call.

**See also:**
`Pulses(), SampleDacMask(), SampleDacZero(), SampleDacUnits$(), SampleStateDac()`

# SampleDacMask()

This function gets the mask value used to enable the DAC outputs from the sampling configuration and optionally sets it to a new value.

```
Func SampleDacMask({new%});
```

new%   If present, sets the mask enabling specified DAC outputs. This mask has one bit for each DAC, set bits enable the corresponding DAC output; bit 0 for DAC 0, bit 1 for DAC 1, and so forth. In integer values bit zero is represented by the value 1, bit one by the value 2, bit 2 by the value 4 and so forth. So to enable DACs 0 and 1 set new% to 3 (1+2), to only enable DAC 3 set new% to 8.

Returns  The DAC outputs mask value before the call.

**See also:**
`SampleStatesMode(), SampleDigOMask(), SampleDigIMask(), SampleStateDac()`

# SampleDacUnits$()

This function gets the units string for a DAC from the sampling configuration and optionally sets it to a new value.

```
Func SampleDacUnits$(port{, new$});
```

port    The DAC number, from 0 to 7.

new     If present, sets the units string for this DAC. This value is used throughout Signal to calibrate DAC values.

Returns  The DAC units string before the call.

**See also:**
```
Pulses(), SampleDacMask(), SampleDacFull(), SampleDacZero(), SampleStateDac()
```

# SampleDacZero()

This function gets the zero value used to scale values written to the DACs from the sampling configuration and optionally sets it to a new value.

```
Func SampleDacZero(port{, new});
```

port    The DAC number, from 0 to 7. Sequencer outputs use the DAC 0 settings for all DAC calibration.

new     If present, sets the value in the units for this DAC corresponding to a zero value. This value is used throughout Signal to calibrate DAC values.

Returns  The DAC zero value before the call.

**See also:**
```
Pulses(), SampleDacMask(), SampleDacFull(), SampleDacUnits$(), SampleStateDac()
```

# SampleDigIMask()

This function gets the mask value used to enable the digital inputs for the External digital multiple states mode from the sampling configuration and optionally sets it to a new value.

```
Func SampleDigIMask({new%});
```

new%    If present, sets the mask enabling specific digital inputs. This mask has one bit for each digital input, set bits enabling the corresponding inputs; bit 0 for input bit 0, bit 1 for input bit 1 and so forth. In integer values bit zero is represented by the value 1, bit one by the value 2, bit 2 by the value 4 and so forth. So to enable digital input bits 0 and 1 set new% to 3 (1+2), to enable bits 2 and 6 set new% to 68. This value is only used in External digital states mode.

Returns  The digital inputs mask value before the call.

**See also:**
```
SampleStatesMode(), SampleStateDig(), SampleDacMask(), SampleDigOMask()
```

# SampleDigMark()

This function sets and gets the flag for enabling the digital marker channel in the sampling configuration.

```
Func SampleDigMark({on%});
```

on%     If present and non-zero, enables the marker channel

Returns  1 if the marker channel was on, 0 if it was off.

**See also:**
```
SampleClear(), SampleKeyMark()
```

# SampleDigOMask()

This function gets the mask value used to enable the digital outputs from the sampling configuration and optionally sets it to a new value.

```
Func SampleDigOMask({new%});
```

new    If present, sets the bit mask enabling the digital outputs. This mask has one bit for each digital output, set bits enabling the corresponding outputs; bit 0 for output bit 0, bit 1 for output bit 1 and so forth. In integer values bit zero is represented by the value 1, bit one by the value 2, bit 2 by the value 4 and so forth. So to enable digital output bits 0 and 1 set `new%` to 3 (1+2), to enable bits 2 and 6 set `new%` to 68 (4+64).

Returns  The digital output bit mask value before the call.

**See also:**
```
SampleStatesMode(), SampleStateDig(), SampleDacMask(), SampleDigIMask()
```

# SampleFixedInt()

This function sets and gets the sweep interval for Fixed interval and Fast Fixed interval sweep modes, as stored in the pulses information. Note that this sets the interval after frames with the specified state, not the interval before a frame. This function normally operates on the stored sampling configuration but, if used during sampling, it operates upon the on-going sampling.

```
Func SampleFixedInt(state{, period});
```

state  This sets the state (set of pulses) to which this applies, from 0 to 256. Use 0 if multiple states are not in use.

period If present, this argument sets the fixed interval period, in seconds.

Returns  The fixed interval period at the time of the call.

**See also:**
```
SampleOutMode(), SampleOutTrig(), SampleOutClock(), SampleFixedVar(), Pulses(),
SampleStates()
```

# SampleFixedVar()

This function sets and gets the percentage variation of the sweep interval for Fixed interval sweep mode, as stored in the pulses information. This function normally operates on the stored sampling configuration but, if used during sampling, it operates upon the on-going sampling.

```
Func SampleFixedVar(state{, vary});
```

state  This sets the state (set of pulses) to which this applies, from 0 to 256. Use 0 if multiple states are not in use.

vary   If present, this argument sets the fixed interval variation, from 0 to 100 percent.

Returns  The fixed interval variation percentage at the time of the call.

**See also:**
```
SampleOutMode(), SampleOutTrig(), SampleOutClock(), SampleFixedInt(), Pulses(),
SampleStates()
```

# SampleHandle()

This gets the handle of a view associated with sampling. This can be used to position, show or hide the sampling control panel or the output control panel.

```
Func SampleHandle(which%);
```

which% Selects which view handle to return:

    0    Main file view.

|   |   |
|---|---|
| 1 | Sampling control panel. |
| 2 | Sequencer control panel. |
| 3 | Pulses configuration dialog. |
| 4 | States control bar. |
| 5 | Clamp control bar. |
| 6 | Auxiliary states control bar. |

Returns  The view handle or 0 if the view does not exist.

**See also:**
```
View(), ViewList(), Window(), WindowVisible()
```

# SampleKey()

This procedure adds events to the keyboard marker channel, exactly as if you had typed them (with the sampling document view as the current view). If there is no sampling, the procedure does nothing. If the output sequencer is running, and you add a key that corresponds to a key linked to a sequencer step, the sequencer jumps to the step.

```
Func SampleKey(key$);
```

`key$`   The first character of the string is added to the keyboard marker channel.

Returns  The time in seconds at which the marker was added to the keyboard marker channel.

**See also:**
```
SampleClear(), SampleKeyMark()
```

# SampleKeyMark()

This function turns the keyboard marker channel on or off or gets the current setting of this from the sampling configuration.

```
Func SampleKeyMark({on%});
```

`on%`   If present and non-zero this turns keyboard markers on in the sampling configuration.

Returns  1 if the keyboard channel is on in the sampling configuration, 0 if not.

**See also:**
```
SampleClear(), SampleDigMark(), SampleKey()
```

# SampleLimitFrames()

This function corresponds to the Number of Frames field on the Automation page of the sampling configuration dialog.

```
Func SampleLimitFrames({frame%});
```

`frame%` The number of frames to set as a limit. A positive number sets the limit and enables. A negative or zero value sets the limit to the positive value of `frame%`, but disables the limit. Omitting the argument means no change.

Returns  The function returns the frames limit as it was at the time of the call. If the limit is disabled, the number of frames is returned negated.

**See also:**
```
SampleClear(), SampleLimitSize(), SampleLimitTime(), SampleWrite()
```

# SampleLimitSize()

This function corresponds to the File size (KByte) field on the Automation page of the sampling configuration dialog.

```
Func SampleLimitSize({size%});
```

size% The size limit for the output file, in KB. A positive value sets the size and enables the limit. A negative or zero value sets the limit to the positive value of `size%`, but disables the limit. Omitting the argument means no change.

Returns The limit before the call. If the limit is disabled, the value is returned negated.

**See also:**
SampleClear(), SampleLimitFrames(), SampleLimitTime(), SampleWrite()

# SampleLimitTime()

This function corresponds to the Sampling duration field on the Automation page of the sampling configuration dialog.

```
Func SampleLimitTime({time%});
```

time% The time in seconds to set as a limit. A positive `time%` sets the limit and enables the limit. A negative or zero value sets the limit to the positive value of `time%`, but disables the limit. Omitting the argument means no change.

Returns The function returns the time limit as it was in the sampling configuration at the time of the call. If the limit is disabled, the time is returned negated.

**See also:**
SampleClear(), SampleLimitFrames(), SampleLimitSize(), SampleWrite()

# SampleMaxWaveRates()

This function sets and gets the maximise waveform rates flag as seen in the sampling configuration dialog Outputs page. This function was added to Signal in version 7.01.

```
Func SampleMaxWaveRates({new});
```

new If present, this sets the new maximise waveform rates flag if non-zero, clears the flag if zero.

Returns The maximise waveform rates flag from the configuration at the time of the call.

**See also:**
SampleClear(), Pulses(), SampleOutClock()

# SampleMembAna()

This command reports on the open state or opens the membrane analysis dialog which is available during sampling when carrying out a clamping experiment and makes it the current view. The dialog opens in the last used position unless not usefully on a monitor, when it opens centred on the application window. If the dialog is already open you can use this command to get the dialog view handle, change the clamp set being analysed and make the dialog the current view. Use `FileClose()` to close an opened membrane analysis dialog. This function was added in Signal version 7.00.

```
Func SampleMembAna({set%{, mode%}});
```

set% Set to the clamping set number from 1 to 8 to open the membrane analysis dialog showing analysis for the specified clamping set. Omit or set to zero as the only argument to report on the open state of the dialog.

mode% Set 1 to open the dialog and show it, 0 or omitted to open and hide it. You may wish to open the window invisibly so that you can position it before it is displayed.

Returns If `set%` is 0 or omitted the return value is the view handle of the dialog or zero if it is not currently open. Otherwise the return value is the handle of the opened dialog, or -1 if the dialog failed to open for some reason or the clamp set specified is not in use.

**See also:**
SampleClamp(), FileClose(), Window()

# SampleMode()

This function gets and optionally sets the sampling sweep mode as seen in the sampling configuration dialog.

```
Func SampleMode({mode%});
```

mode%   If supplied, this argument sets the new sampling mode as follows:

    0    Basic
    1    Peri-trigger
    2    Extended
    3    Fixed interval
    4    Fast triggers
    5    Fast fixed interval
    6    Gap-free

Returns  The sweep mode set in the sampling configuration at the time of the call.

**See also:**
SampleClear(), SamplePeriType(), SampleFixedInt(), SampleOutLength(), SampleOutTrig()

# SampleOutClock()

This function sets and gets the outputs clock as seen in the sampling configuration dialog Outputs page.

```
Func SampleOutClock({period{, flags%}});
```

period  If present, this argument sets the outputs clock period, in seconds. This value sets the time resolution for pulses and sequencer output, for measuring sweep absolute start times, for timing sweeps in Fixed interval mode and for measuring the time of marker data. This value should generally not be less than 0.1 ms for a micro1401, 25 μs for a Micro1401 mk II and Micro1401-3 and 10 μs for Power1401s.

flags%  If present, this sets clock options, if omitted the default value of zero is used. Set flags% to 2 to enable the maximum possible waveform output rates at the expense of sequencer timing precision. In versions of Signal before version 4.06, adding 1 to flags% was equivalent to checking the "Synchronise sampling" box in the sampling configuration dialog.

Returns  The outputs clock period from the configuration at the time of the call.

**See also:**
SampleOutMode(), SampleFixedInt(), SampleOutTrig(), Pulses(), SampleMaxWaveRates()

# SampleOutLength()

This function sets and gets the length of the pulse outputs as stored in the pulses information for use in Extended and Fixed interval sweep modes. This function normally operates on the stored sampling configuration but, if used during sampling, it operates upon the on-going sampling.

```
Func SampleOutLength(state{, length});
```

state   This sets the state (pulses set) to which this applies, from 0 to 256. Use 0 if multiple states are not in use.

length  If present, this argument sets the length of the pulses output frame, in seconds.

Returns  The pulses output frame length at the time of the call.

**See also:**
SampleOutMode(), SampleOutTrig(), SampleOutClock(), Pulses(), SampleStates()

# SampleOutMode()

This function sets and gets the outputs mode as seen in the sampling configuration dialog Outputs page.

`Func SampleOutMode({mode%});`

mode%   This argument determines the action of the command:
- 0    Sets None outputs mode.
- 1    Sets Pulses outputs mode.
- 2    Sets Sequencer outputs mode.

Returns  The outputs mode from the configuration at the time of the call.

**See also:**
`SampleClear(), Pulses()`

# SampleOutTrig()

This function sets and gets the sweep trigger time within the pulses frame as stored in the pulses information for use in Extended and Fixed interval sweep modes. This function normally operates on the stored sampling configuration but, if used during sampling, it operates upon the on-going sampling.

`Func SampleOutTrig(state{, time});`

state   This sets the state (set of pulses) to which this applies, from 0 to 256. Use 0 if multiple states are not in use.

time   If present, this argument sets the sweep trigger time within the pulse outputs, in seconds. The value should be from 0 to the pulses outputs length.

Returns  The sweep trigger time at the time of the call.

**See also:**
`SampleOutLength(), SampleFixedInt(), SampleOutClock(), Pulses(), SampleStates()`

# SamplePause()

This function ascertains or sets whether the sampling is set to pause after each sweep, this is equivalent to the Pause at sweep end check box. It can also be used during sampling to control the current pause state.

`Func SamplePause({pause%});`

pause%  If present this is equivalent to changing the state of the Pause at sweep end check box. A non-zero value pauses sampling and zero enables sampling.

Returns  If pause% is present it returns the new state as 1 or 0, or a negative error code. If pause% is absent it returns the current state as 1 or 0.

**See also:**
`SampleSweep(), SampleStart(), SampleStop(), SampleStatus(), SampleWrite(), SampleAbort()`

# SamplePeriBitState()

This function gets or selects the state of the digital input bit required to trigger sampling. It is equivalent to the dropdown selection box in the digital peri-trigger sampling configuration.

`Func SamplePeriBitState({set%});`

set%   If present a value of 1 selects Trigger on bit high in the digital peri-trigger sampling configuration. A value of 0 selects Trigger on bit low.

Returns  The value for the setting at the time of the call.

**See also:**

SamplePeriDigBit(), SamplePeriHyst(), SamplePeriLevel(), SamplePeriLowLev(),
SamplePeriType(), SamplePeriPoints()

# SamplePeriDigBit()

This function gets or sets the digital bit number in the digital peri-trigger information within the sampling configuration.

```
Func SamplePeriDigBit({bit%});
```

bit%    If present and in the range 8-15, this sets the new digital bit number in the digital peri-trigger sampling configuration.

Returns  The value for the peri-trigger digital bit in the sampling configuration at the time of the call.

**See also:**
SamplePeriBitState(), SamplePeriHyst(), SamplePeriLevel(), SamplePeriLowLev(),
SamplePeriType(), SamplePeriPoints()

# SamplePeriHyst()

This function gets or sets the hysteresis value for triggering from an analogue channel level in the peri-trigger information within the sampling configuration. If used while sampling is in progress, this gets and sets the hysteresis value currently in use.

```
Func SamplePeriHyst({level});
```

level   If present this sets the new hysteresis value in the peri-trigger sampling configuration or sampling document.

Returns  The value for hysteresis in the sampling configuration at the time of the call.

**See also:**
SamplePeriDigBit(), SamplePeriBitState(), SamplePeriLevel(), SamplePeriLowLev(),
SamplePeriType(), SamplePeriPoints()

# SamplePeriLevel()

This function gets or sets the threshold level for triggering from an analogue channel level in peri-trigger mode. This is the threshold level for +Analogue and -Analogue peri-trigger types and the upper threshold for the =Analogue peri-trigger type. If used while sampling is in progress, this gets and sets the threshold level currently in use.

```
Func SamplePeriLevel({level});
```

level   If present this sets the peri-trigger threshold level in the sampling configuration or sampling document.

Returns  The value for peri-trigger threshold level in the sampling configuration.

**See also:**
SamplePeriDigBit(), SamplePeriBitState(), SamplePeriHyst(), SamplePeriLowLev(),
SamplePeriType(), SamplePeriPoints()

# SamplePeriLowLev()

This function gets or sets the peri-trigger Lower threshold for =Analogue peri-trigger type in the peri-trigger sampling configuration. If used while sampling is in progress, this gets and sets the lower threshold level currently in use.

```
Func SamplePeriLowLev({level});
```

level   If present this sets the lower threshold for =Analogue peri-trigger type in the peri-trigger sampling configuration or sampling document.

Returns  The value for the lower threshold for =Analogue peri-trigger type in the peri-trigger sampling configuration.

**See also:**
```
SamplePeriDigBit(), SamplePeriBitState(), SamplePeriHyst(), SamplePeriLevel(),
SamplePeriType(), SamplePeriPoints()
```

# SamplePeriType()

This function gets or sets the type of peri-trigger in the peri-trigger information within the sampling configuration.

```
Func SamplePeriType({pts%});
```

pts%    If present this sets the type of trigger in the sampling configuration as follows:

        0   +Analogue.

        1   -Analogue.

        2   =Analogue.

        3   Digital.

        4   Event.

Returns  The trigger type in the sampling configuration at the time of the call.

**See also:**
```
SamplePeriDigBit(), SamplePeriBitState(), SamplePeriHyst(), SamplePeriLevel(),
SamplePeriLowLev(), SamplePeriPoints()
```

# SamplePeriPoints()

This function gets or sets the number of data points in the frame before the trigger as given by Pre-trig. points in the Peri-trigger section of the sampling configuration dialog.

```
Func SamplePeriPoints({pts%});
```

pts%    If present this sets the pre-trigger points in the peri-trigger sampling configuration. This can be any negative number or a positive number less than the points per sweep.

Returns  The value for pre-trigger points in the peri-trigger sampling configuration.

**See also:**
```
SamplePeriDigBit(), SamplePeriBitState(), SamplePeriHyst(), SamplePeriLevel(),
SamplePeriLowLev(), SamplePeriType()
```

# SamplePoints()

This function gets or sets the number of data points per ADC port per frame as given by the Frame points in the sampling configuration.

```
Func SamplePoints({pts%});
```

pts%    If present this sets the number of frame points in the sampling configuration.

Returns  The value for frame points in the sampling configuration.

**See also:**
```
SampleClear(), SampleRate(), SamplePeriPoints(), SampleTrigger()
```

# SamplePortFull()

This function gets and sets the Full value for an input port, as shown in the sampling configuration dialog. Complete calibration of a waveform channel requires Full, Zero and Units to be set up correctly. This function normally operates on the stored sampling configuration but if used during sampling it operates upon the on-going sampling.

```
Func SamplePortFull(port%{, full});
```

port%   The port number (0-127).

full    The value of the data corresponding to the ADC full-scale level (+5 volts or +10 volts for a 10 volt system) at the input.

Returns  The value for the port Full scale value at the time of the call, or zero for illegal port numbers.

**See also:**
```
SampleClear(), SamplePorts(), SamplePortOptions$(), SamplePortUnits$(),
SamplePortZero(), SampleTel()
```

# SamplePortName$()

This function gets and sets the title attached to a port, as shown in the sampling configuration dialog.

**Func SamplePortName$(port%{, new$});**

port%   The port number (0-127).

new$    If present, the new title. If the title is too long, it is truncated.

Returns  The title at the time of the call, or an empty string for illegal channel numbers.

**See also:**
```
SampleClear(), SamplePorts(), SamplePortFull(), SamplePortOptions$(),
SamplePortUnits$(), SamplePortZero()
```

# SamplePortOptions$()

This function gets and sets the online processing options attached to a waveform input port setup, as shown in the sampling configuration dialog.

**Func SamplePortOptions$(port%{, new$});**

port%   The port number (0-127).

new$    If present, the new options string, up to 7 characters long. If the string is too long it will be truncated.

Returns  The options string at the time of the call, or an empty string for illegal channel numbers.

**See also:**
```
SampleClear(), SamplePortFull(), SamplePorts(), SamplePortName$(),
SamplePortUnits$(), SamplePortZero()
```

# SamplePorts()

This function gets or sets the ADC ports to be used as shown in the sampling configuration.

**Func SamplePorts({get%[]|num%{, new%[]}});**

num%    If present this sets the number of ADC ports to take from the new% array. If new% is not present the new ADC ports will be 0..num%-1.

get%[]  If present as a single argument this is filled with ADC ports from the sampling configuration up to the size of the array. If there are insufficient ports to fill the array the unused entries are left unchanged.

new%[]  If present this holds the new ADC ports for the sampling configuration. The number of new ADC ports will be restricted by the size of num% or by the size of this array, whichever is the smaller.

Returns  The number of ADC ports at the time of the call.

**See also:**
```
SampleClear(), SamplePortFull(), SamplePortName$(), SamplePortOptions$(),
SamplePortUnits$(), SamplePortZero()
```

# SamplePortUnits$()

This function gets and sets the units string for an input port, as shown in the sampling configuration dialog. Complete calibration of a waveform channel requires Full, Zero and Units to be set up correctly.

`Func SamplePortUnits$(port%{, new$});`

port%   The port number (0-127).

new$    The units to use. If the string is longer than 7 characters only the first 7 are used.

Returns  The units at the time of the call, or an empty string for illegal channel numbers.

**See also:**
`SampleClear(), SamplePorts(), SamplePortFull(), SamplePortOptions$(),`
`SamplePortZero()`

# SamplePortZero()

This function gets and sets the Zero value for an input port, as shown in the sampling configuration dialog. This function normally operates on the stored sampling configuration but if used during sampling it operates upon the on-going sampling.

`Func SamplePortZero(port%{, zero});`

port%   The port number (0-127).

zero    The value of the data corresponding to a zero-volt input at the ADC.

Returns  The value for the port zero value at the time of the call, or zero for illegal port numbers.

**See also:**
`SampleClear(), SamplePorts(), SamplePortFull(), SamplePortUnits$(),`
`SamplePortOptions$()`

# SampleProtocol()

This function is used during sampling to retrieve the index number of the currently executing protocol and optionally to start off execution of a protocol.

`Func SampleProtocol({num|name$});`

num     If provided, the protocol number to use, from 1 to the number returned by `Protocols()`.

name$   If provided, the name of the protocol to use.

Returns  The number of the protocol in use before this call or a negative error code.

**See also:**
`Protocols(), ProtocolName$(), SampleState()`

# SampleRate()

This function gets the waveform sample rate in Hz from the sampling configuration and optionally sets it to a new value.

`Func SampleRate({new});`

new     If present, the new preferred rate in Hz. The actual sampling rate used will be as close as possible to the new rate, but will not always match it exactly.

Returns  The sampling rate before the call. This is the actual sampling rate that would have been used, not the preferred rate.

**See also:**
`SampleClear(), SamplePoints(), SampleTrigger(), SampleBurst(), BinSize()`

# SampleReset()

This function can be used while sampling is in progress to abandon sampling, delete any data that has been written to disk, and return to the state as if `FileNew()` had just been used to create a new data file.

```
Func SampleReset({noask%});
```

`noask%` If provided and non-zero, this causes the normal querying of the user about this potentially destructive operation to be bypassed.

Returns  0 if the reset operation completed without a problem, or a negative error code.

**See also:**
`SampleAbort(), SampleStart(), SampleStop(), SampleStatus(), SampleWrite(), SamplePause()`

# SampleSeqCtrl()

This function sets and gets the options that control the use of the output sequence form of outputs.

```
Func SampleSeqCtrl(opt%{, new%};
```

`opt%`    This value selects the option which is to be retrieved or set: 1 for the sequencer jump control, 2 for the restart sequence at frame start flag.

`new%`    The new value for the control option. For the jump control: 0 = keyboard, control panel and script, 1 = control panel and script, 2 = script only. For the restart sequence flag a value of 1 selects the sequence restarting at the start of each frame, 0 selects a free-running sequence without restarts.

Returns  If you are setting an option, the function returns the previous value of the option. If you are reading a value, the function returns the value.

**See also:**
`SampleKey(), SampleSeqStep(), SampleSequencer$(), SampleOutClock(), SampleStart()`

# SampleSeqStep()

This function returns the current sequencer step when using the output sequence form of outputs 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$(), SampleOutClock(), SampleSeqVar()`

# SampleSeqTable()

If there is a sampling document sampled using the output sequence form of outputs, you can use this function to find the size of any table set in the sequence by the `TABSZ` directive or to transfer data between an integer array and the table.

```
Func SampleSeqTable({tab%[]{, offs%{, get%}}});
```

`tab%[]` An integer array holding items to transfer to the 1401 sequencer table or to hold items read back from the table. The array size sets the maximum item count.

`offs%`   This sets the index into the sequencer table to start the transfer. The first index in the table is 0. If this value is negative or greater than or equal to the sequencer table size, no data is transferred. If omitted, the value 0 is used.

`get%`    Set 0 or omit this argument to transfer data to the sequencer table, set to 1 to transfer data from the sequencer table.

Returns If you call this with no arguments, the return value is the size of the sequencer table. Otherwise, the returned value is the number of items transferred between the sequencer table and the array. A negative error code is also possible, for example -1 if there is no sampling document.

**See also:**
`SampleKey(), SampleSequencer$(), SampleSeqVar()`

# SampleSequencer()

You can use this function to set the sequencer file to attach to the Sampling configuration. Use `SampleSequencer$()` to get the name of the current sequencer file.

```
Func SampleSequencer(new$);
```

`new$` The name of the sequence file. Pass an empty string to set no sequencer file.

Returns It returns 0 if all was well, or a negative error code.

This function modifies the sampling configuration by setting the sequencer file name and altering the outputs mode to the output sequence form of outputs. In order for it to take effect, this function should be used before the FileNew() script function is used to create a new data file and get ready for sampling.

**See also:**
`SampleKey(), SampleSequencer$(), SampleOutClock(), SampleSeqVar(), FileNew()`

# SampleSequencer$()

This function returns the name of the sequencer file (for the output sequence form of outputs) that is currently attached to the sampling configuration. Use `SampleSequencer()` to set the sequencer 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 function is used just before or during sampling using the output sequence form of outputs to get or set the value of an output sequencer variable. Values set before the sampling window exists (it is created by using FileNew()) are ignored. Values set after FileNew() and before `SampleStart()` set the initial sequencer variable values.

```
Func SampleSeqVar(sVar%{, new%});
```

`sVar%` The sequencer variable to set or read, in the range 1 to 64.

`new%` The new value for the output sequencer variable. If present, the value of the variable is updated. Omit to return the variable value. A common error when setting variables for the DAC instruction is to set a value 65536 times too small.

Returns If you are setting a variable value, or if this function is used at an inappropriate time, the return value is 0. If you are reading a value, the function returns the value of the variable.

**See also:**
Output sequencer DAC instruction, `SampleKey(), SampleSeqStep(), SampleSequencer$(),`
`SampleStart(), SampleSeqWave(), FileNew()`

# SampleSeqWave()

This command can be used to set the number or size of the arbitrary waveform output areas used by the output sequence form of outputs, to retrieve such information, or to load waveform data into or out of a waveform area.

```
Func SampleSeqWave(area%, arr%[]{, offs%{, get%}});
Func SampleSeqWave(area%, arr[]{, offs%{, get%}});
```

These two forms of the command copy data between a script array and the waveform output area. They can be used while sampling is in progress, or after the `FileNew()` function has been used but before `SampleStart()`, but will not have any effect if used when not sampling. If a single DAC is being used for waveform output, the array data is a simple list of values. If more than one DAC is used the values for the various DACs are interleaved so if DACs two and four are being used the data array goes 242424....

area%   The waveform area number, from 1 to 256. Zero is also accepted and taken to mean 1 for compatibility with previous versions of Signal.

arr%[]   The array holding, or to be updated with, the waveform data. For integer arrays values from -32768 to 32767 span the complete DAC range.

arr[]   As for `arr%[]` but an array of real numbers. For real arrays, the DAC scaling values for the sequencer (actually the settings for DAC 0) will be used to convert from user units in the array to DAC values.

offs%   The offset within the waveform area to start the transfer. The size of the data array sets the transfer size. If this parameter is omitted, a value of zero is used.

get%   Set this parameter to 1 to read data from the waveform area, set it to zero or omit it to transfer data into the waveform area.

```
Func SampleSeqWave(areas%, mask%, pnts%, hz%);
```

This form of the command operates on the sampling configuration settings and is used to define the waveform areas (which are all the same). It should therefore be used before sampling has begun.

areas%   The number of waveform output areas, from 1 to 256. Zero is accepted and taken to mean 1 for compatibility with previous versions of Signal.

mask%   A value that defines which DACs are used, DACs from 0 to 7 are supported. The mask has bit zero set if DAC 0 is used, bit 1 for DAC 1, bit 2 for DAC 2 and so on. In integer values bit zero is represented by the value 1, bit one by the value 2, bit 2 by the value 4 and so forth. So to use DACs 0 and 1 set `mask%` to 3 (1+2), to only use DAC 4 set `mask%` to 16. Note that `mask%`, like the other values, applies to all the areas.

pnts%   Sets the number of waveform points per DAC in each waveform area. Note that you can set the sequence to only play part of an area.

hz%   Sets the waveform output rate, in points per second.

```
Func SampleSeqWave(area%{, num%});
```

This form of the command is used to retrieve area settings from the sampling configuration, and so should also be used before sampling has begun.

area%   This argument is currently ignored and should be set to zero, it is present to allow future extensions.

num%   A number which determines the return value of the function, if omitted its value is taken to be zero. Note that none of these values vary between areas, though the sequencer WAVE instruction can specify less than the total number of area points for replay. Possible values are:

   0   `pnts%` * number of DACs, i.e. the total data size for each area in DAC points.
   -1   `mask%`
   -2   `pnts%`
   -3   `hz%`
   -4   The number of DACs in `mask%`
   -5   The number of waveform areas

Returns   For the form used to read back area information, the return value is as shown in the table above. In all other cases the return value is 0 if all went well or a negative error code.

### See also:

```
SampleKey(), SampleSeqStep(), SampleSequencer$(), SampleStart(), SampleSeqVar(),
SampleDacFull(), SampleDacZero()
```

# SampleStart()

This function starts sampling off, it can be used after `FileNew()` has created a new file view based on the current sampling configuration. It starts sampling immediately or on a 1401 event trigger.

**Func SampleStart({trig%});**

trig%   If this is 0 or omitted, sampling starts immediately, otherwise sampling waits for a trigger signal on the 1401 E1 input.

Returns  0 if all went well or a negative error code.

**See also:**
```
SampleAbort(), SampleReset(), SampleStop(), SampleStatus(), SampleWrite(),
SamplePause()
```

# SampleState()

This function is used during sampling to set the current state directly; it is the scripting equivalent of controlling the state manually with the states control bar. The command should be used with states sequencing set to Manual, or the values set will be overridden by the sampling system.

**Func SampleState(num);**

num     The state to use, from 0 to 256.

Returns  The state in use at the time of the call or a negative error code. Note that this is the state currently in use; the function return value will not change until the next sweep is started by calling `SampleSweep()` or an equivalent operation.

**See also:**
```
SampleStatesOrder(), SampleStatesRun(), SampleStates()
```

# SampleStateDac()

This function gets the DAC output value for a specific state from the sampling configuration and optionally sets it to a new value.

**Func SampleStateDac(state, port{, new});**

state   The state for which information is required, from 0 to 256.

port    The DAC number, from 0 to 3.

new     If present, sets the value to be output to the DAC for this state in Static outputs states mode. This value is scaled to a voltage using the scaling factors currently set for the DAC.

Returns  The DAC output value before the call.

**See also:**
```
SampleStatesMode(), SampleStates(), SampleState(), SampleDacMask(), SampleDacFull(),
SampleDacZero()
```

# SampleStateDig()

This function gets the value to be written to the digital outputs for a specific state or the value read from the digital inputs to set a state and optionally sets it to a new value.

**Func SampleStateDig(state{, new%});**

state   The state for which information is required, from 0 to 256.

new%    If present, sets the value to be written to the digital outputs for this state in Static outputs states mode, or the value used to test the digital input data in External digital states mode.

Returns  The digital output or input value before the call.

**See also:**
SampleStatesMode(), SampleStates(), SampleState(), SampleStateDac(), SampleDigIMask()

# SampleStateLabel$()

This function gets the label string for a specific state (this is used to label the state control bar buttons amongst other things) and optionally sets it to a new value.

**Func SampleStateLabel$(state{, new$});**

state  The state for which information is required, from 0 to 256.

new$  If present, sets the new label for this state. A state label should consist of printable text and should not be longer than ten characters.

Returns  The label for the state before the call.

**See also:**
SampleStateRepeats(), SampleStates(), SampleState(), SampleStateDac()

# SampleStateRepeats()

This function gets the number of repeats for a specific state from the sampling configuration and optionally sets it to a new value.

**Func SampleStateRepeats(state{, new});**

state  The state for which information is required, from 0 to 256.

new  If present, sets the number of times the state is repeated in Numeric and Random ordering mode if Individual repeats is enabled.

Returns  The number of repeats before the call.

**See also:**
SampleStateRepeats(), SampleStates(), SampleState()

# SampleStates()

This function gets the number of extra states from the sampling configuration and optionally sets it to a new value.

**Func SampleStates({new});**

new  If present, the new number of extra states. Values from 1 to 256 set the states and turn on Multiple states mode, a value of zero turns off multiple states.

Returns  The number of extra states before the call, or zero if multiple states were disabled.

**See also:**
SampleStatesMode(), SampleStatesOrder(), SampleState(), SampleStatesOptions()

# SampleStatesIdle()

This function gets the number of states ordering cycles to be executed before idling from the sampling configuration and optionally sets it to a new value.

**Func SampleStatesIdle({idle%});**

idle%  If present, this sets the number of cycles of states using Numeric, Random or Semi-random state sequencing that will be executed before idling. A zero value means keep cycling forever. This parameter is ignored for protocol ordering.

Returns  The states cycles before idling before the call.

**See also:**
```
SampleStates(), SampleStatesMode(), SampleStatesOrder(), SampleState(),
SampleProtocol(), ProtocolFlags()
```

# SampleStatesMode()

This function gets the mode for multiple states from the sampling configuration and optionally sets it to a new value.

```
Func SampleStatesMode({new});
```

new      If present, sets the new multiple states mode as follows:

       0   External digital

       1   Static outputs

       2   Dynamic outputs

Returns  The mode for multiple states before the call.

**See also:**
```
SampleStates(), SampleStatesOrder(), SampleState()
```

# SampleStatesOptions()

This function gets the options for non-protocol ordering multiple states from the sampling configuration and optionally sets new options.

```
Func SampleStatesOptions({new%});
```

new%     If present, this sets the new multiple states options and is the sum of:

       1   Start cycling states automatically when sampling starts.

       2   Turn on writing to disk whenever states cycling starts.

Returns  The multiple states options before the call.

**See also:**
```
SampleStates(), SampleStatesIdle(), SampleStatesMode(), SampleState(),
SampleProtocol(), ProtocolFlags()
```

# SampleStatesOrder()

This function gets the ordering mode for multiple states from the sampling configuration and optionally sets it to a new value.

```
Func SampleStatesOrder({new%});
```

new%     If present, this sets the new multiple states ordering mode as follows:

       0   Numeric

       1   Random

       2   Protocol

       3   Semi-random

       4   Random repeats

Returns  The states ordering mode before the call.

**See also:**
```
SampleStates(), SampleStatesIdle(), SampleStatesMode(), SampleState(),
SampleProtocol(), ProtocolFlags()
```

# SampleStatesPause()

This function is used during sampling to get the paused or not-paused state of multiple states cycling and optionally sets it to a new value; it is the scripting equivalent of pressing the Pause button in the states control bar.

```
Func SampleStatesPause({pause%});
```

`pause%` If present, this sets the new paused condition of state cycling:

    0    Not paused

    1    Paused

Returns  The paused condition before the call.

**See also:**
```
SampleStates(), SampleStatesIdle(), SampleStatesMode(), SampleState(),
SampleProtocol(), ProtocolFlags()
```

# SampleStatesRepeats()

This function gets the number of times each state is repeated from the sampling configuration and optionally sets it to a new value, or selects individual repeats mode.

```
Func SampleStatesRepeats({new%});
```

`new%` If present and non-zero, disables individual repeats and sets the number of times each state is repeated in Numeric and Random ordering mode. A value of zero turns on individual repeats.

Returns  The number of repeats before the call, or zero if individual repeats were enabled.

**See also:**
```
SampleStateRepeats(), SampleStatesOrder(), SampleState()
```

# SampleStatesReset()

This function is used during sampling to reset the states-sequencing system and the pulses built-in varations; it is the scripting equivalent of pressing the Reset button in the states control bar.

```
Func SampleStatesReset();
```

Returns  Zero or a negative error code.

**See also:**
```
SampleStatesOrder(), SampleStatesRun(), SampleStates()
```

# SampleStatesRun()

This function is used during sampling to set the state seqencing execution mode; the script equivalent of the Manual, On Write or Cycle buttons in the states control bar.

```
Func SampleStatesRun({mode%});
```

`mode%` The mode of sequencing to use, as follows:

    0    Manual or direct script control of states

    1    States sequencer runs, moves to next state if data written

    2    States sequencer runs, moves to next state unconditionally

Returns  The state sequencing mode in use before this call or a negative error code.

When using protocols, setting mode to 0 will terminate the protocol, though this command cannot be used to start a protocol running.

**See also:**
```
SampleStatesOrder(), SampleStatesReset(), SampleState(), SampleStatesOptions()
```

# SampleStatesStep()

This function returns the current value of the states sequencing counter which is used during sampling to control states seqencing.

```
Func SampleStatesStep();
```

Returns  The states sequencing counter at the time of the call.

When using non-protocol ordering, the step counter runs from zero to states*repeats, in protocol mode it is the count of steps since the protocol started.

**See also:**
SampleStatesOrder(), SampleStatesReset(), SampleState()

# SampleStatus()

This function returns the current state of any sampling.

```
Func SampleStatus();
```

Returns  A code indicating the sampling state or -1 if there is no sampling:

    0   A file view is ready to sample, but it has not been told to start yet.
    1   Sampling is waiting for an Event 1 trigger.
    2   Sampling of a sweep is now in progress or is awaiting a trigger.
    3   Sampling is paused at the end of a sweep.
    4   Sampling is stopped but not finished (changes to -1 when it has finished).

**See also:**
SampleAbort(), SampleReset(), SampleStart(), SampleStop(), SampleWrite(), SamplePause()

# SampleStop()

This function stops sampling in progress and is equivalent to using the Stop and Finish buttons of the sampling control panel. The default behaviour is that there is no intermediate state between stopping and finishing, when sampling is stopped by using this function. The function does not return until sampling has stopped.

```
Func SampleStop({noFin%});
```

noFin% If present and non zero then sampling will stop but not finish. SampleSweep() may then be used to continue sampling.

Returns  0 if sampling stopped correctly or a negative error code.

**See also:**
SampleAbort(), SamplePause(), SampleReset(), SampleStart(), SampleStatus(), SampleSweep(), SampleWrite()

# SampleSweep()

If sampling is paused at the end of a sweep, or stopped but not finished because a limit was reached, this starts sampling of the next sweep. The current sweep will be lost if it is unsaved. This function is the equivalent of the Continue button in the sampling control panel (or More when sampling is stopped).

```
Func SampleSweep();
```

Returns  0 or a negative error code.

**See also:**
SamplePause(), SampleReset(), SampleStart(), SampleStop()

# SampleSweepPoints()

If variable points per sweep sampling is in use, this function gets the number of points sampled for a given state and optionally sets a new points value. If used online, it operates upun the sampling that is in progress.

```
Func SampleSweepPoints(state%{, new%});
```

state   The state for which information is required, from 0 to 256.

new%    If this is provided it sets the number of ADC points sampled per channel for the specified state. This should be an even number no larger than the overall sweep points for the sampling configuration, odd numbers will be rounded down to a value divisible by 2, values larger than overall sweep points will be truncated.

Returns  The number of ADC points sampled per channel before the call.

**See also:**
SamplePoints(), SampleVaryPoints(), SampleStates(), SampleMode(), SampleState()

# SampleTel()

This function gets or sets the standard 1401 telegraph options in the sampling configuration. The first form of the command is used to set telegraph options, the second form reads settings back.

```
Func SampleTel(nSet%, nPort%, nTel%{, volt[]|volt, gain[]|gain});
Func SampleTel(nSet%, opt%);
```

nSet%   The set of telegraph values to work with, from 1 to 4.

nPort%  The port number whose input is to be scaled by the telegraph. If this and nTel% are both set to -1 then these telegraph settings will be turned off.

nTel%   The port number being used to read the telegraph voltage. If this and nPort% are both set to -1 then these telegraph settings will be turned off.

volt    A telegraph voltage. If this is set to 0 and no gain is given then the list of gains and telegraph voltages is cleared. If this matches the voltage in an existing telegraph entry, then this plus the gain value replaces the original entry.

volt[]  An array of telegraph voltages.

gain    The gain associated with a particular voltage.

gain[]  An array of gains associated with the array of telegraph voltages provided.

opt%    This can be set to the following negative values which cause the function to return the following values:

-1   The scaled port number for this set of telegraph values or -1 if they are turned off.
-2   The telegraph port number or 0 if this set is turned off.
-3   The number of items in the telegraph list or 0 if this set is turned off.
-4   The first telegraph voltage in the list.
-5   The first gain in the list.
-n   If n > 3 and n is even, the (n-2)/2 nd voltage in the list is returned. If n > 4 and n is odd, the (n-3)/2 nd gain in the list is returned.

Returns  0, the requested value or a negative error code.

**See also:**
SamplePorts(), SamplePortFull(), SamplePortZero()

# SampleTrigger()

This function gets or sets the external trigger option in the sampling configuration. It can also be used during sampling to control the current trigger state.

```
Func SampleTrigger({trig%});
```

`trig%`   If this is non-zero, sampling of each frame waits for a trigger input pulse. Zero turns trigger mode off.

Returns  0 or 1, or a negative error code.

**See also:**
SampleClear(), SampleRate(), SampleSweep(), SamplePoints(), SampleStatus()

# SampleTriggerInv()

This function gets or sets the rising edge trigger flag as seen in the sampling configuration dialog.

`Func SampleTriggerInv({inv%});`

`inv%`   If this is non-zero, the rising edge sweep trigger option is turned on, zero sets the standard falling-edge trigger.

Returns  0 or 1 to indicate the previous state of the rising edge trigger flag.

**See also:**
SampleClear(), SampleTrigger(), SampleSweep(), SampleStatus()

# SampleVaryPoints()

This function gets the flag enabling variable sweep points from the sampling configuration and optionally sets it to a new value.

`Func SampleVaryPoints({new%});`

`new%`   If this is provided it sets the variable sweep points enable in the sampling configuration; non-zero turns variable sweep points on, zero turns it off. This will not affect sampling unless the sweep mode in use allows variable points mode.

Returns  The variable sweep points flag before the call.

**See also:**
SamplePoints(), SampleSweepPoints(), SampleStates(), SampleMode()

# SampleWrite()

This function controls the automatic writing of data to the file during sampling and is equivalent to the Write to disk at sweep end check boxes.

`Func SampleWrite({write%});`

`write%` If present this sets the state of automatic writing of data at the end of each sweep:
   0   Disable writing to disk at the end of each sweep
   1   Enable writing to disk at the end of each sweep

Returns  The state of automatic writing to file at the end of each sweep: 0 for disabled, 1 for enabled.

**See also:**
SampleClear(), SamplePause(), SampleSweep(), SampleStatus()

# SampleXConv()

This function sets and gets the Use ADC External convert flag as seen in the sampling configuration dialog General page. This function was added to Signal in version 7.01.

`Func SampleXConv({new});`

`new`    If present, this sets the new external convert flag if non-zero, clears the flag if zero.

Returns  The external convert flag from the configuration at the time of the call.

**See also:**

SampleClear(), SampleRate(), SampleXConvUnits$(), SampleXCPointsPerUnits()

# SampleXCPointsPerUnit()

This function sets and gets the number of ADC points corresponding to one external convert unit as set in the sampling configuration dialog General page. This function was added to Signal in version 7.01.

```
Func SampleXCPointsPerUnits({new});
```

new      If present, this sets the new external convert points per unit.

Returns  The external convert points per unit from the configuration at the time of the call.

**See also:**
SampleClear(), SampleXConv(), SampleXCUnits$()

# SampleXCUnits$()

This function sets and gets the X axis units used for external convert sampling as seen in the sampling configuration dialog General page. This function was added to Signal in version 7.01.

```
Func SampleXCUnits${(new$});
```

new$     If present, this sets the new external convert X axis units.

Returns  The external convert X axis units from the configuration at the time of the call.

**See also:**
SampleClear(), SampleXConv(), SampleXCPointsPerUnits()

# SampleZeroOffset()

This function sets and gets the offset value applied to the zero point on the x-axis for newly sampled data. Normally zero will be at the start of the frame or the trigger point for peri-triggered data.

```
Func SampleZeroOffset({offset});
```

offset   If present this sets the amount in seconds by which to offset zero:

Returns  The offset in seconds of zero before the function was called.

**See also:**
SampleClear()

# ScriptBar()

This function controls the Script toolbar. Call the command with no arguments to return the number of toolbar buttons. The first button is numbered 0.

```
Func ScriptBar({nBut%{, &get$}});
Func ScriptBar(set$);
```

nBut%    Set -1 and omit get$ to clear all buttons and return 0. Otherwise it is a button number and returns -1 if the button does not exist, 0 if it is the last button, and 1 if higher-numbered buttons exist. get$ returns the information as for set$.

set$     This holds up to 8 characters of button label, a vertical bar, the path to the script file including .sgs, a vertical bar and a pop-up comment. The function returns the new number of buttons or -1 if all buttons are already used.

Returns  See the descriptions above. Negative return values indicate an error.

For example, the following code clears the script bar and sets a button:

```
ScriptBar(-1);     'clear all buttons
ScriptBar("ToolMake|C:\\Scripts\\ToolMake.sgs|Build a toolbar");
```

**See also:**
App()

# ScriptRun()

This sets the name of a script to run when the current script terminates. You can pass information to the new script using disk files or by using the Profile() command. You can call this function as often as you like; only the last use has any effect.

```
Proc ScriptRun(name${, flags%});
```

name$    The script file to run. You can supply a path relative to the current folder or a full path to the script file. If you supply a relative path, it must still be valid at the end of the current script. Set name$ to "" to cancel running a script.

flags%   Optional flags, taken as 0 if omitted. Sum of: 1 = run even if the current script ends in an error, 2 = keep loaded script in memory

If the file you name does not exist when Signal tries to run it, nothing happens. If the nominated script is not already loaded, Signal will load it, run it and unload it unless the keep loaded script in memory flag is set. If a loaded script calls Yield() or calls any function that allows the system to idle (Toolbar(), DlgShow()...), the script can be unloaded while it is still running. This is usually harmless unless the loaded script attempts to use App(3), which will return 0 if the script is no longer in memory.

**See also:**
App(), Profile()

# Seconds()

This sets or gets the timer in seconds and is used for relative time measurements.   If you want the position reached within the current sweep, use the Maxtime() function.

```
Func Seconds({set{, hiRes%}});
```

set      If present, this sets the time in seconds. The time is zero when Signal starts.

hiRes%   If present, this selects between normal or high-resolution timing. A zero value selects the standard resolution of nominally 1 millisecond (but may be worse on some systems). Set this to 1 for the highest timer resolution available. Note that the default is 1 millisecond resolution, which is set when Signal starts. Changes to the resolution are persistent between scripts.

Returns  If hiRes is not present, the function returns the time in seconds. This is the value before any new time is set. If hiRes is present, the return value is the time resolution, in seconds. This is 1 millisecond for the standard resolution and can be (much) less than 1 microsecond for the high resolution timer.

**High resolution timer caveats**
The function we use for the high-resolution timer is supposed to use a fixed frequency source. However, some machines (incorrectly) use a time based on CPU cycles; this is a problem for many laptops which change the CPU speed to save battery power. There are also reports that some multi-core machines report different times for each core. If you have a problem, the following web links may be useful: South bridge chipset problem, Athlon X2 problem. Note that computers from 2007 onwards are less likely to have problems.

**See also:**
MaxTime()

# Selection$()

This function returns the text in the current view that is currently selected. To find where the currently selected text is in relation to the overall view text, use the MoveBy() function.

```
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(), EditSelectAll(), MoveBy(), MoveTo(), XLow(),
XHigh()
```

# 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. Closing a serial port deletes any data from `SerialWrite()` that has not been transmitted, if this could cause problems you can poll the output buffer space using `SerialWrite()` until all the data is gone.

```
Func SerialClose(port%);
```

`port%`   The serial port to close as defined for `SerialOpen()`.

Returns  0 or a negative error code.

### Safely closing the serial line

As mentioned above, `SerialClose()` will discard any un-transmitted serial line data. The safe way to ensure that all data has been sent is to save the output buffer size at the time the serial line is opened and wait until the available space is equal to the buffer size before closing:

```
var serbufsize%;                                     ' Variable to hold size of serial outp

if (SerialOpen(port%, 9600) < 0)                     ' Open serial line & check that it wo
    halt;
serbufsize% := SerialWrite(port%);                   ' Save the size of the output buffer

    ....                                             ' Do whatever serial line communicati

while (SerialWrite(port%) < serbufsize%) do          ' Wait until output buffer is empty a
wend;
SerialClose(port%);                                  '  before safely closing the serial po
```

This code will ensure reliable operation in normal circumstances but can hang if the hardware handshake or other mechanism prevents the serial line data from being transmitted (which is why we do not wait automatically as part of `SerialClose()`). You do not need to wait for all characters to be sent if delays in your code or other effects ensure that all serial line data will have been sent, if you get problems with hung serial line transmission you could replace the simple while loop used above with one that times out after a suitable period:

```
var stime;
stime := Seconds();                                  ' Get the time at which we started wa
while (SerialWrite(port%) < serbufsize%) do          ' Wait until output buffer is empty a
    if (Seconds() > (stime + 2)) then break; endif;  '  or until two seconds have elapsed
wend;
SerialClose(port%);                                  '  before closing the serial port
```

**See also:**
```
SerialOpen(), SerialWrite(), SerialRead(), SerialCount()
```

# SerialCount()

This counts the characters or items buffered in a serial port opened by `SerialOpen()`. Use this to detect input so your script can do other tasks while waiting for serial data. There is an internal buffer of 1024 characters per port that is filled when you use `SerialCount()`. The size of this buffer limits the number of characters that this function can tell you about. To avoid character loss when you are not using a serial line handshake, do not buffer up more than a few hundred characters with `SerialCount()`.

```
Func SerialCount(port%{, term$});
```

`port%`   The serial port to use as defined for `SerialOpen()`.

`term$`   An optional string holding the character(s) that terminate an input item.

Returns  If `term$` is absent or empty, this returns the number of characters that could be read. If `term$` is set, this returns the number of complete items that end with `term$` that could be read.

**See also:**
```
SerialOpen(), SerialWrite(), SerialRead(), SerialClose()
```

# SerialOpen()

This function opens a serial port and configures it for use by the other serial line functions. It is not an error to call `SerialOpen` more than once on the same port. The serial routines use the host operating system serial line support. Consult your system documentation for information on serial line connections and Baud rate limits.

```
Func SerialOpen(port%{, baud%{, bits%{, par%{, stop%{, hsk%}}}}});
```

port%   The serial port to use, in the range 1 to 256 (1 to 9 before version 5.05). The number of ports actually available depends on the computer. Only one port is found on most PCs but USB serial ports can be used to add more.

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 used to encode a character. Windows supports 4 to 8 bit encoding but you should normally restrict yourself to 7 or 8. If `bits%` is omitted, 8 is set. Standard values are 7 or 8 data bits. If you set 7 data bits, character codes from 0 to 127 can be read. If you set 8 data bits, codes from 0 to 255 are possible.

par%    Set this to 0 for no parity check, 1 for odd parity or 2 for even parity. If you do not specify this argument, no parity is set.

stop%   This sets the number of stop bits as 1 or 2. If omitted, 1 stop bit is set. If you specify 5 data bits, a request for 2 stop bits results in 1.5 stop bits being used.

hsk%    This sets the handshake mode, sometimes called "flow control". 0 sets no handshake, 1 sets a hardware handshake, 2 sets XON/XOFF protocol.

Returns  0 or a negative error code.

**See also:**
```
SerialWrite(), SerialRead(), SerialCount(), SerialClose()
```

# SerialRead()

This function reads characters, a string, an array of strings, or binary data from a nominated serial port that was previously opened with `SerialOpen()`. Binary data can include character code 0; string data never includes character 0.

```
Func SerialRead(port%, &in$|in$[]|&in%|in%[]{, term${, max%}});
```

port%   The serial port to read from as defined for `SerialOpen()`.

in$     A single string or an array of strings to fill with characters. To use an array of strings you must set a terminator or all input goes to the first string in the array.

in%     A single integer (`term$` and `max%` are ignored) or an array of integers (`term$` and `max%` can be used) to read binary data. Each integer can hold one character, coded as 0 up to 255. The function returns the number of characters returned.

term$   If this is an empty string or omitted, all characters read are input to the string, integer array or to the first string in the string array and the number of characters read can be limited by `max%`. The function returns the number of characters read.

If `term$` is not empty, the contents are used to separate data items in the input stream. Only complete items are returned and the terminator is not included. For example, set the terminator to `"\n"` if lines end in line feed, or to `"\r\n"` if input lines end with carriage return then line feed. If `in$` is a string, one item at most is returned. If `in$[]` is an array, one item is returned per array element. The function returns the number of items read unless `in` is an integer (`in%` or `in%[]`) when it returns the number of characters returned.

max%   If present, it sets the maximum number of characters to read into each string or into the integer array. If a terminator is set, but not found after this many characters, the function breaks the input at this point as if a terminator had been found. There is a maximum limit set by the integer array size, the size of the buffers used by Signal to process data and by the size of the system buffers used outside Signal. This is typically 1024 characters.

Returns  The function returns the number of characters or items read or a negative error code. If there is nothing to read, it waits 1 second for characters to arrive before timing out and returning 0. Use `SerialCount()` to test for items to read to avoid a time out.

**See also:**
`SerialOpen(), SerialWrite(), SerialCount(), SerialClose()`

# SerialWrite()

This writes one or more strings or binary data to a serial port opened by `SerialOpen()`. Use the command with a single argument to find out how much space is available in the serial line output buffer (typically 1024 characters).

```
Func SerialWrite(port%{, out$|out$[]|out%|out%[]{, term$}});
```

port%   The serial port to write to as defined for `SerialOpen()`.

out$   A single string or an array of strings to write to the output. The return value is the number of strings written.

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  If only the port argument is present, the return value is the amount of space available in the serial port output buffer. With more arguments, for success the return value is as documented for the out argument. If there is no room in the output buffer for the data the return value is -1 except when out$[] is used when the return values is the count of complete strings actually sent.

`SerialWrite()` does not actually write the strings or data to the serial line; it merely puts it into a buffer for later transmission. If you use the `SerialClose()` command before the system has had time to write buffered characters to the serial port, the buffered characters will be lost. See `SerialClose()` for example code that avoids this problem.

**See also:**
`SerialOpen(), SerialRead(), SerialCount(), SerialClose()`

# SetXXX()

This family of commands creates memory view windows. Memory views which require processing are derived from and attached to the current data view, which should be a file view. This does not apply to `SetCopy()` and `SetMemory()`, which create memory views that do not use a `Process` command. The `MeasureToXY()` function is special in that it creates an XY view which will receive data points from processing a source view which can be either a file or a memory view. The `SetXXX()` functions do not update the display, for which you should use `Draw()` or `DrawAll()`.

All these functions, with the exception of `SetOpCl()` return a positive view handle if they succeed or a negative error code. Possible errors are: bad channel number, illegal number of bins and out of memory. The new derived memory view will be empty until a processing command is executed for it. The processing of a memory view takes data from the source view and replaces or adds to the data in the memory view. `SetOpCl()` creates a idealised trace channel in the source view to hold an idealised trace and returns the channel number.

When these functions create a new view, it is made the current view. The view is created invisibly and must be made visible with `WindowVisible(1)` before it will appear.

**See also:**
`Process(), ProcessAll(), ProcessFrames(), SetAutoAv(), SetAverage(), SetLeak(),`
`SetOpCl(), SetOpClScan(), SetPower(), MeasureToXY(), SetCopy(), SetMemory()`

# SetAmplitude()

This function creates a memory view to hold an amplitude histogram in each channel when it is processed. `Sweeps()` reports the number of sweeps of waveform data accumulated by processing into the memory view. The current view when `SetAmplitude()` is called will be the source view for the data to be processed. In this version of Signal the source view cannot be log-binned.

```
Func SetAmplitude(ch%, bins%{, minA|minA${, maxA|maxA${, sTime|sTime${,
eTime|eTime$}}}});
```

ch%　　A waveform channel to analyse from the current view. Use a channel number (1 to n).

bins%　The number of bins in the resulting histogram.

minA　　The smallest amplitude to be represented in the histogram. The default is the lower limit of the ADC range of the channel.

minA$　The smallest amplitude to be represented in the histogram, as a string. Strings such as `"HCursor(1)"` can be used.

maxA　　The largest amplitude to be represented in the histogram. The default is the upper limit of the ADC range of the channel.

maxA$　The largest amplitude to be represented in the histogram, as a string. Strings such as `"HCursor(1)"` can be used.

sTime　The start time of the data to be included in the analysis. The default is the minimum time in the frame.

sTime$　A string giving the start time of the data to be included in the analysis, e.g. `"Cursor(1)"`.

eTime　The end time of the data to be included in the analysis. The default is the maximum time in the frame.

eTime$　A string giving the end time of the data to be included in the analysis e.g. `"Cursor(2)"`.

Returns　The function returns a handle for the new view, or a negative error code.

**See also:**
`SetXXX(), Process(), ProcessAll(), ProcessFrames(), Sweeps(), View()`

# SetAutoAv()

This function creates a memory view to hold a sum or average in each channel when it is processed. The memory view will hold multiple frames, with set numbers of source frames being averaged into each destination frame. This allows you to set up averaging with, for example, every ten source frames processed into a new average. The amount moved-on between averages can be separately controlled for extra flexibility. A separate auto-averaging mode uses the source frame state number to determine the frame in the average to which the source data will be added. The current view when `SetAutoAv()` is called will be the source view for the data to be processed. `SetAutoAv()` is very similar to `SetAverage()`.

```
Func SetAutoAv(cSpc, perAv%, betAv%{, width, offs{, sum%{, xzero%{, cntEx%{,
doErr%{, mode%{, maxFr%}}}}}}});
```

cSpc　　A channel specifier for the channels to average.

perAv%　The number of source frames to use per average.

betAv%　The number of source frames between the first frame for one average and the first frame for the next. If `perAv%` is the same as `betAv%`, then each `perAv%` frames processed make a new average frame. If `betAv%` is less than `perAv%`, then some source frames are used for more than one average; if it is greater than `perAv%` then some source frames will be unused.

width　　The width of the average in x axis units. If omitted the whole frame will be used. The maximum is limited by available memory.

offs　　This sets the offset in x axis units from start of frame to the start of the data to average. If omitted or zero, the data will be taken from the start of the frame.

sum%　　If present and non-zero, each channel in the memory view will hold the sum of the data accumulated. If omitted or zero, the memory view channels will hold the mean of the data accumulated.

xzero% If present and non-zero, this forces the x axis of the memory view to start at zero. If omitted or zero, the start of the x axis will be the same as the start of the data that is averaged.

cntEx% For non state-based averaging (if **mode%** is zero): if present and non-zero, excluded frames will count as if they had been added, so Signal will not use extra frames to form each average and thereby remain in step with (for example) the sampling protocol. For state-based averaging (if **mode%** is 1): if present and non-zero, an average will not be generated for frames with state zero.

doErr% If present and set to 1 then error bar information will be generated.

mode% If present and set to 1 then averaging will use the source state number to determine the destination frame for each source view frame, otherwise the **perAv%** and **betAv%** values are used to calculate the destination frame numbers.

MaxFr% If present and set to more than 0 then this sets the maximum number of frames that will be created in the new memory view during averaging, otherwise no limit will be imposed. Note that when using **By State** mode, the maximum state number for which an average will be created is `MaxFr%-1` as the first average frame is used for state number 0.

Returns  The function returns a handle for the new view, or a negative error code.

**See also:**
```
SetXXX(), SetAverage(), Process(), ProcessAll(), ProcessFrames(), Sweeps(), View(),
MinTime()
```

# SetAverage()

This function creates a memory view to hold a sum or average in each channel when it is processed. `Sweeps()` reports the number of sweeps of waveform data accumulated by processing into the memory view. The current view when `SetAverage()` is called will be the source view for the data to be processed.

```
Func SetAverage(cSpc{, width, offs{, sum%{, xzero%{, doErr%}}}});
```

cSpc    A channel specifier for the channels to average.

width   The width of the average in x axis units. If omitted the whole frame will be used. The maximum is limited by available memory.

offs    This sets the offset in x axis units from start of frame to the start of the data to average. If omitted or zero, the data will be taken from the start of the frame.

sum%    If present and non-zero, each channel in the memory view will hold the sum of the data accumulated. If omitted or zero, the memory view channels will hold the mean of the data accumulated.

xzero%  If present and non-zero, this forces the x axis of the memory view to start at zero. If omitted or zero, the start of the x axis will be the same as the start of the data to average, as defined by offset `offs` from `MinTime()` in the current frame.

doErr%  If present and set to 1 then error bar information will be generated.

Returns  The function returns a handle for the new view, or a negative error code.

**See also:**
```
SetXXX(), SetAutoAv(), Process(), ProcessAll(), ProcessFrames(), Sweeps(), View(),
MinTime()
```

# SetCopy()

This function creates a new memory view with channels selected from and identical to those in the current view. The new view can be empty or contain data copied from the current frame. It is attached to no source view and has no implied `Process()`. Idealised trace channels are not handled.

```
Func SetCopy(cSpc, title$, bcopy%);
```

cSpc    A channel specifier for the channels to copy. Virtual channels, real marker channels and idealised trace channels will all be ignored.

title$  The new window title.

bcopy%  If this is not 0 the data values are copied into the new memory view. If this is 0 the waveform data values in the new view are zero and marker channels are empty.

Returns  A handle for the new view, or a negative error code.

**See also:**
```
SetXXX(), SetMemory(), View()
```

# SetINTH()

This function creates a memory view to hold an interval histogram in each channel when it is processed. Each marker interval processed increments the value returned by `Sweeps()`, even intervals that are too short or too long to contribute to the histogram. The current view when `SetINTH()` is called will be the source view for the data to be processed.

```
Func SetINTH(cSpc, bins%, binsz{, minInt{, cross%{, sTime|sTime${, eTime|
eTime$}}}});
```

cSpc   A channel specifier for the marker channels in the source view that are going to be analysed. Invalid channels are ignored. The resulting channel list must hold at least one valid channel.

bins%   The number of bins in the resulting histogram.

binsz   The width of each bin in seconds, this is converted into the time units in use before the histogram is created.

minInt  The smallest interval to be represented in the histogram, the default is zero.

cross%  Omit or set to zero if you do not want to process intervals crossing frame boundaries, set to 1 if you do want to process these intervals.

sTime   The start time of the data to be included in the analysis. The default is the minimum time in the frame. It is an error to use cross-frame analysis with partial frames so if the `cross%` parameter is non-zero this parameter must not be provided.

sTime$  A string giving the start time of the data to be included in the analysis, e.g. `"Cursor(1)"`. Again this should not be provided if cross-frame analysis is used.

eTime   The end time of the data to be included in the analysis. The default is the maximum time in the frame. It is an error to use cross-frame analysis with partial frames so if the `cross%` parameter is non-zero this parameter must not be provided.

eTime$  A string giving the end time of the data to be included in the analysis e.g. `"Cursor(2)"`. Again this should not be provided if cross-frame analysis is used.

Returns  The function returns a handle for the new memory view, or a negative error code.

**See also:**
```
SetXXX(), Process(), ProcessAll(), ProcessFrames(), Sweeps(), View()
```

# SetLeak()

This function creates a memory view to hold leak subtracted data when it is processed. The current view when `SetLeak()` is called, which must be a file view, will be the source view for the data to be processed.

```
Func SetLeak(mode%, chan%, stim%, base|base$, pulse|pulse$, width, form%,
sub%{, zero%{, cntEx%}});
```

mode%   A value to set the leak subtraction mode: 0 for Basic, 1 for P/N or 2 for States.

chan%   A single waveform channel from the current view, this is the channel that will be leak-subtracted, all other source channels are copied unchanged. Use a channel number (1 to n).

stim%   A single waveform channel from the current view. This is the channel that will be used to measure the stimulus pulse size. Normally this will be a channel on which the stimulus was recorded.

base    A time at which the baseline level can be measured; a time outside the stimulus pulse.

base$   The baseline level time expressed as a string, allowing constructs such as "Cursor(1) - 10".

pulse   A time at which the pulse level can be measured; a time inside the stimulus pulse.

pulse$  The pulse time expressed as a string.

width   The width of the two level measurements. The measurement used is the average of all waveform points within the specified width.

form%   The first frame used to measure the leak in Basic mode, the number of frames to use for the leak in P/N mode and the state code for leak frames in States mode.

sub%    The last frame used to measure the leak in Basic mode and the number of frames to subtract the current leak from in P/N mode. This parameter is unused in States mode.

zero%   If present and non-zero, the baseline level will be maintained constant by the leak subtraction process; otherwise this adjustment is not done.

cntEx%  If present and non-zero, excluded frames will count as if they had been used, so Signal will not continue to search for enough frames to form each leak and so remain in step with the sampling protocol.

Returns  The function returns a handle for the new view, or a negative error code.

**See also:**
SetXXX(), FrameState(), SetPower(), Process(), ProcessAll(), ProcessFrames(), View()

# SetMemory()

This function creates a memory view of user-defined type, attached to no source view and with no implied Process().

```
Func SetMemory(chans%, pts%, binsz, offset, marks%, tmks%, mkBns%, title$,
xU${, yU${, xT${, yT$}}});
```

chans%  The number of waveform channels in the view (1 to 80).

pts%    The number of data points in each waveform channel.

binsz   The x axis increment per point in the waveform channels. This is equivalent to the sample interval for sampled data. This value should be positive and non-zero.

offset  The x axis value at the first point of the waveform channels.

marks%  The number of marker channels in the view, not including text markers (0 to 80).

tmks%   The number of text marker channels in the view. Not implemented yet.

mkBns%  The number of marker items in each marker channel.

title$  The new window title.

xU$     The x axis units.

yU$     Optional, y axis units, blank if omitted.

xT$     Optional, x axis title (otherwise blank).

yT$     Optional, y axis title (otherwise blank).

Returns  The function returns a handle for the new view, or a negative error code.

**See also:**
SetXXX(), SetCopy(), View()

# SetOpCl()

This function creates an idealised trace process. The current view when `SetOpCl()` is called will be the source view for the data to be processed.

```
Func SetOpCl(cSpc, start|start$, end|end$, lev1|lev1$, lev2|lev2$, base|
base${, intrp%{, track%{, flags%}}});
```

cSpc      A channel specifier for the channels to analyse.

start     The start time in seconds of the idealised trace to be created.

start$    The start time as a string, e.g. `"Cursor(1)"`.

end       The end time in seconds of the idealised trace to be created.

end$      The end time as a string, e.g. `"Cursor(2)"`.

lev1      The level which the original data trace must cross in order to change from closed state to open state.

lev1$     The `lev1` value as a string, e.g. `"HCursor(1)"`.

lev2      The level which the original data trace must cross in order to change from open state to closed state.

lev2$     The `lev2` value as a string, e.g. `"HCursor(2)"`.

base      The level at which the data trace is considered to be in the closed state. This is used for calculating additional thresholds for multi-level data.

base$     The base level as a string, e.g. `"HCursor(3)"`.

intrp%    The interpolation method to use for calculating the time of a threshold crossing. Set to 0 for no interpolation and 1 for linear interpolation. The default value is 0.

track%    The number of data points in the closed state to use in tracking the base level, in order to adjust the thresholds to compensate for baseline drift. The default value is 0.

flags%    A set of flags built up by adding together the following values:
   1   Outward current. An opening of a channel leads to a more positive current.
   2   Multiple level data. Normally set if there is more than one channel in the patch.

   This parameter is set to 0 by default.

Returns   The function returns the number of the idealised trace channel.

**See also:**
`SetXXX(), SetOpClAmp(), SetOpClBurst(), SetOpClHist(), SetOpClScan()`

# SetOpClAmp()

This function creates a memory view to hold an open/closed amplitude histogram from an idealised trace. The current view when `SetOpClAmp()` is called will be the source view for the data to be processed.

```
Func SetOpClAmp(chan%, bins%, minA|minA$, maxA|maxA$, incl%, excl%{, flags
%});
```

chan%     The channel number in the source view. This channel must have an idealised trace fitted, for a histogram to be built.

bins%     The number of bins in the resulting histogram.

minA      The smallest amplitude to be represented in the histogram.

minA$     The smallest amplitude to be represented in the histogram, as a string. Strings such as `"HCursor(1)"` can be used.

maxA      The largest amplitude to be represented in the histogram.

maxA$     The largest amplitude to be represented in the histogram, as a string. Strings such as `"HCursor(1)"` can be used.

incl%   A set of flags associated with each open/closed time to include. If an event has any of the flags in the `incl%` set and none of the flags in the `excl%` set, it will be included in the histogram. See `SetOpClHist()` for a list of flag values.

excl%   A set of flags defining those events to be excluded from the histogram. Events having flags in the `excl%` set will be excluded regardless of whether they have flags in the `incl%` sets. See `SetOpClHist()` for a list of flag values.

flags%  Set to 1 if amplitudes are to be measured relative to the baseline. If this is set to 0 or omitted then processing will use absolute amplitudes.

Returns  The function returns a handle for the new view, or a negative error code.

**See also:**
`SetXXX(), SetOpCl(), SetOpClBurst(), SetOpClHist(), SetOpClScan()`

# SetOpClBurst()

This function creates a memory view to hold an open/closed burst time histogram from an idealised trace. The current view when `SetOpClBurst()` is called will be the source view for the data to be processed. A burst duration is from the start time of an included event to the start time of an excluded event having a duration greater than the critical interval.

```
Func SetOpClBurst(chan%, binsz, maxDur, crInt, incl%, excl%);
```
```
Func SetOpClBurst(chan%, minDur, maxDur, crInt, incl%, excl%, nBins%);
```

chan%   The channel number in the source view. This channel must have an idealised trace fitted for a histogram to be built.

binsz   The x increment per bin in the histogram.

minDur  The minimum duration of a burst to be included in the histogram. This is used for log-binning.

maxDur  The maximum duration of a burst to be included in the histogram.

crInt   The critical interval.

incl%   A set of flags associated with each open/closed time to include. If an event has any of the flags in the `incl%` set and none of the flags in the `excl%` set, it will be included in the histogram. See `SetOpClHist()` for a list of flag values.

excl%   A set of flags defining those events to be excluded from the histogram. Events having flags in the `excl%` set will be excluded regardless of whether they have flags in the `incl%` sets. See `SetOpClHist()` for a list of flag values.

nBins%  The number of bins in the histogram of log-binned data. Omit or set to 0 for all bins to be of width `binsz`.

Returns  The function returns a handle for the new view, or a negative error code.

**See also:**
`SetXXX(), SetOpCl(), SetOpClAmp(), SetOpClHist(), SetOpClScan()`

# SetOpClHist()

This function creates a memory view to hold an open/closed time histogram from an idealised trace. The current view when `SetOpClHist()` is called will be the source view for the data to be processed.

```
Func SetOpClHist(chan%, binsz, maxDur, incl%, excl%);
```
```
Func SetOpClHist(chan%, minDur, maxDur, incl%, excl%, nBins%);
```

chan%   The channel number in the source view. This channel must have an idealised trace fitted for a histogram to be built.

binsz   The x increment per bin in the histogram.

minDur  The minimum duration of an open/closed time to be included in the histogram. This is used for log-binning.

maxDur The maximum duration of an open/closed time to be included in the histogram.

incl% A set of flags associated with each open/closed time to include. If an event has any of the flags in the incl% set and none of the flags in the excl% set it will be included in the histogram. A set of flags is built up by adding together the following values:

| | | |
|---|---|---|
| 0x0001 | 1 | Level 1. Closed times and the first open level will have this set. |
| 0x0002 | 2 | Bad data: Events marked as bad in the idealised trace editor. |
| 0x0004 | 4 | Assumed amplitude: Events whose amplitude has not been calculated from the raw data. |
| 0x0008 | 8 | Reserved |
| 0x0010 | 16 | First latency: The period from the start of the idealised trace to the first transition. |
| 0x0020 | 32 | Truncated: The last event in an idealised trace. |
| 0x0040 | 64 | Closed time. |
| 0x0080 | 128 | Open time. |
| 0x0100 | 256 | Reserved |
| 0x0200 | 512 | Reserved |
| 0x0400 | 1024 | Reserved |
| 0x0800 | 2048 | Level 6 of multi-level data. |
| 0x1000 | 4096 | Level 5 of multi-level data. |
| 0x2000 | 8192 | Level 4 of multi-level data. |
| 0x4000 | 16384 | Level 3 of multi-level data. |
| 0x8000 | 32768 | Level 2 of multi-level data. |

excl% A set of flags defining those events to be excluded from the histogram. Events having flags in both the incl% and excl% sets will be excluded.

nBins% The number of bins in the histogram of log-binned data. Omit or set to 0 for all bins to be of width binsz.

Returns The function returns a handle for the new view, or a negative error code.

**See also:**
SetXXX(), SetOpCl(), SetOpClAmp(), SetOpClBurst(), SetOpClScan()

# SetOpClScan()

This function creates an idealised trace process using the SCAN method. The current view when SetOpClScan() is called will be the source view for the data to be processed. You will need to call OpClNoise before calling this function.

```
Func SetOpClScan(cSpc, start|start$, end|end$, base|base$, thr|thr$, open|
open$, cutoff{, track%{, flags%}});
```

cSpc A channel specifier for the channels to analyse.

start The start time in seconds of the idealised trace to be created.

start$  The start time as a string, e.g. `"Cursor(1)"`.

end    The end time in seconds of the idealised trace to be created.

end$   The end time as a string, e.g. `"Cursor(2)"`.

base   The initial level of the baseline.

base$  The `base` value as a string, e.g. `"HCursor(1)"`.

thr    The threshold level which the original data trace must cross in order to change from closed state to an open state.

thr$   The `thr` value as a string e.g. `"HCursor(2)"`.

open   The approximate initial level at which the channel will be assumed to be fully open.

open$  The `open` value as a string, e.g. `"HCursor(3)"`.

cutoff The –3dB frequency in Hz of the cut-off frequency of the filter used to remove noise from the raw data..

track% The number of data points over which to form a running average of the baseline. The default value is 0.

flags% A set of flags built up by adding together the following values:

  1  Outward current. An opening of a channel leads to a more positive current.
  2  Avoid sub-levels. Normally set if you know there are no sub-levels so multiple transitions will be used instead where possible.

  This parameter is set to 0 by default.

Returns The function returns the number of the idealised trace channel.

**See also:**
`SetXXX(), OpClNoise(), SetOpClAmp(), SetOpClBurst(), SetOpClHist(), SetOpCl()`

# SetPower()

This function creates a memory view to hold a power spectrum in each channel when it is processed. `Sweeps()` reports the number of sweeps accumulated by processing into the memory view. The current view when `SetPower()` is called will be the source view for the data to be processed. In this version of Signal the source view cannot be log-binned data.

```
Func SetPower(cSpc, fftsz%{, offset{, wnd%}});
```

cSpc   A channel specifier for the channels to analyse.

fftsz% The size of the transform used in the FFT. This must be a power of 2 in the range of 16 to 262144, values that are not an integral power of two will be rounded down to the next lower such value. The memory view that is created has half this number of bins. The width of each bin is the sampling rate of the channel divided by `fftsz%`. Each block of `fftsz%` data points processed increments the value for `Sweeps()`.

offset This sets the offset in x axis units from start of frame to the start of the data to analyse. If omitted or zero, the data will be taken from the start of the frame.

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:**
`SetXXX(), SetAverage(), ArrFFT(), Process(), Sweeps(), View()`

# SetTrend()

This function creates an XY view to hold XY points calculated by trend plot processing. The function creates an XY view with a single channel; more channels can be added using `SetTrendChan()`. This command has been replaced by the `MeasureToXY()`, `MeasureX()` and `MeasureY()` commands, which should be used for all new scripts.

```
Func SetTrend(name$, xtyp%, xc%, xp|xp$, xb|xb$, xw, ytyp%, yc%, yp|yp$, yb|
yb$, yw{, pts%{, xFCh%{, yFCh%{, comX%}}})};
```

name$  The name of the channel. The channel name is shown in the XY key area.

xtyp%  The type of measurement to take for the X part of each point. The possible values are:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | Curve area | 6 | Modulus | 11 | Standard deviation | 16 | Standard error |
| 2 | Mean | 7 | Maximum | 12 | Extreme | 17 | RMS error |
| 3 | Slope | 8 | Minimum | 13 | Peak | | |
| 4 | Area | 9 | Peak to peak | 14 | Trough | | |
| 5 | Sum | 10 | RMS amplitude | 15 | Point count | | |

Types from 100 up are special values:

| | | | | | |
|---|---|---|---|---|---|
| 100 | Value at point | 105 | Frame abs time | 110 | Value product |
| 101 | Value difference | 106 | Frame state value | 111 | Value above baseline |
| 102 | Time at point | 107 | 0-based fit coefficient | | |
| 103 | Time difference | 108 | User entered value | | |
| 104 | Frame number | 109 | Value ratio | | |

*These values have changed from earlier versions of Signal – you will need to adjust any scripts that use them.*

xc%  A single waveform channel from the current view, this is the channel that will be used to take the X measurement. Use a channel number (1 to n). For `xtyp%` = 107 this is the coefficient index.

xp  The time for single-point X measurements and difference measurements. For measurements between points, this is the end time.

xp$  The time for single-point X measurements and difference measurements expressed as a string. This allows constructs such as `"Cursor(1) - 10"` to be used.

xb  The reference time for difference measurements; for measurements between points, this is the start time.

xb$  The reference time for difference measurements expressed as a string.

xw  The width used for some measurements, particularly value at point and value difference.

ytyp%  The type of measurement to take for the Y part of each point. The possible values are the same as for `xtyp%`.

yc%  A single waveform channel from the current view; this is the channel that will be used to take the Y measurement. Use a channel number (1 to n). For `ytyp%` = 107 this is the coefficient index.

yp  The time for single-point Y measurements and difference measurements. For measurements between points, this is the end time.

yp$  The time for single-point Y measurements and difference measurements expressed as a string. This allows constructs such as `"Cursor(1) - 10"` to be used.

yb  The reference time for difference measurements; for measurements between points, this is the start time.

yb$  The reference time for difference measurements expressed as a string.

yw  The width used for some measurements, particularly value at point and value difference.

pts%  The number of points for this channel before they are recycled. If this is omitted or set to zero, all points are simply added.

xFCh%  The channel containing the fit to take fit coefficient values from if `xtyp%` = 107. If this is set to 0 (the default) Signal will scan all channels to find one containing a fit with sufficient coefficients to use.

yFCh%  The channel containing the fit to take fit coefficient values from from if `ytyp%` = 107. If this is set to 0 (the default) Signal will scan all channels to find one containing a fit with sufficient coefficients to use.

comX%  Set to 1 for all channels in the plot to share the same x-values. 0 (default) sets channels to have independent x-values.

Returns  The function returns a handle for the new view, or a negative error code.

**See also:**
```
MeasureToXY(), FrameState(), FrameAbsStart(), ChanMeasure(), SetTrendChan(),
Process(), ProcessAll(), ProcessFrames()
```

# SetTrendChan()

This function adds another channel to an XY view created using `SetTrend()`. The current view must be the XY view to be modified. The `SetTrendChan()` function can be used to create XY views with up to 32 channels. This command has been replaced with `MeasureChan()`, which should be used for all new scripts.

```
Func SetTrendChan(name$, xtyp%, xc%, xp|xp$, xb|xb$, xw, ytyp%, yc%, yp|yp$,
yb|yb$, yw{, pts%{, xFitCh%{, yFitCh%}}});
```

All of the parameters to `SetTrendChan()` are exactly the same as for `SetTrend()`.

Returns  The function returns zero or a negative error code.

**See also:**
```
MeasureChan(), FrameState(), ChanMeasure(), SetTrend(), Process(), ProcessAll(),
ProcessFrames()
```

# ShowBuffer()

This function gets or sets the show frame buffer flag from the current view.

```
Func ShowBuffer({yes%});
```

yes%  If this is non-zero the frame buffer is shown otherwise the current frame data is shown. If this is omitted, no change is made.

Returns  The buffer show flag at the time of the call.

**See also:**
```
BuffXXX(), Frame()
```

# ShowFunc()

This function draws a function over a data channel. This function is included for compatibility. It has been replaced with `ChanFitShow()` in version 3 or later.

```
Func ShowFunc(func%, chan%{, start, coefs[]});
```

func%  The type of the function to show:

    0   Don't show a function
    1   Single exponential
    2   Double exponential
    3   Single gaussian
    4   Double gaussian

chan%  The channel on which to show the function.

start  The time to start drawing from.

coefs  The coefficients to use in drawing the function.

Returns  The function returns zero or a negative error code.

**See also:**
```
FitExp(), FitGauss(), ChanFitShow()
```

# Sin()

This calculates the sine of an angle in radians, or converts an array of angles into sines.

```
Func Sin(x|x[]{[]...});
```

x        The angle, expressed in radians, or a real array of angles. The best accuracy of the result is obtained when the angle is in the range -2pi to 2pi.

Returns  When the argument is an array, the function replaces the array with the sines and returns 0 or a negative error code. When the argument is not an array the function returns the sine of the angle.

**See also:**
```
Abs(), ATan(), Cos(), Cosh(), Exp(), Frac(), Ln(), Log(), Max(), Min(), Pow(),
Rand(), Round(), Sqrt(), Sinh(), Tan(), Tanh()
```

# Sinh()

This calculates the hyperbolic sine of a value or of an array of values.

```
Func Sinh(x|x[]{[]...});
```

x        The value, or an array of real values.

Returns  When the argument is an array, the function replaces the array elements with their hyperbolic sines and returns 0. When the argument is not an array the function returns the hyperbolic sine of the argument.

**See also:**
```
Abs(), ATan(), Cos(), Cosh(), Exp(), Frac(), Ln(), Log(), Max(), Min(), Pow(),
Rand(), Round(), Sqrt(), Sinh(), Tan(), Tanh()
```

# Sound()

This command has two variants. The first plays a tone of set pitch and duration. The second plays a `.WAV` file or system sound through your sound card if your system has multimedia support.

```
Func Sound(freq%, dur{, midi%});
Func Sound(name${, flags%});
```

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. This uses the system speaker (usually a small and nasty device attached to your computer motherboard), not your sound card.

dur     The sound duration, in seconds. The script stops during output.

midi%   If present and non-zero, `freq%` is interpreted as a MIDI value.

name$   The name of a `.wav` file or of a system sound. You can supply the full path to the file or just a file name and the system will search for the file in the current directory, the Windows directory, the Windows system directory, directories listed in the `PATH` environmental variable and the list of directories mapped in a network. If no file extension is given, `.wav` is assumed. The file must be short enough to fit in available physical memory.

A blank name halts sound output. If `name$` is any of the following (case is important), a standard system sound plays:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| `"S*"` | Asterisk | `"SS"` | Start | `"SE"` | Exit | `"SH"` | Hand |
| `"SW"` | Welcome | `"S?"` | Query | `"SD"` | Default | `"S!"` | Exclamation |

flags%  This optional argument controls how the data is played. It is the sum of:

| 0x0001 | 1 | Play asynchronously (start output and return). Without this flag, Signal does nothing until replay ends. |
| 0x0002 | 2 | Silence when sound not found. Normally `Sound()` plays the system default sound if the nominated sound cannot be found. |
| 0x0008 | 8 | Loop sound until stopped by another `Sound()` command. You must also supply the asynchronous flag if you use loop mode. |
| 0x0010 | 16 | Don't stop a playing sound. Normally, unless the "No wait" flag is set, each command cancels any playing sound. |
| 0x2000 | 8192 | No wait if sound is already playing. `Sound("",0x2010)` returns 1 if a previous asynchronous sound is finished and 0 if not. |

If you don't supply this argument, the flag value is set to 0x2000.

Returns    The tone output returns 0 or a negative error code. The multimedia output returns non-zero if the function succeeded and zero if it failed.

**See also:**
`Speak()`

# Speak()

If your system supports text to speech, this command allows you to convert a text string into speech. We do not provide facilities to setup voices or to route the sound output; you must do this from the Speech applet in the control panel. There are two command variants. The first outputs text as speech; the second is for control of the output and provides status information.

```
Func Speak(text${, opt%});
Func Speak({what%{, val}});
```

text$    A string holding the text to output, for example `"Sampling has started"`.

opt%    This optional argument (default value 1) controls the text conversion and output method. It is the sum of the following flags:

1    Speak asynchronously. Without this flag the command waits until speech output is over before returning.

2    Cancel any pending speech output.

4    Speak punctuation marks in the text.

8    Process embedded SAPI XML; the www.microsoft.com/speech web site has more information on this advanced topic. For example:
`Speak("Emphasis on <EMPH>this</EMPH> word",8);`

16    Reset to the standard voice settings before speaking.

what%    An optional variable, taken as 0 if it is omitted:

0    Returns 1 if speech is playing and 0 if it is not.

1    Wait for up to val seconds (default value 3.0) for output to end. The return value is 0 if playing is finished, 1 if it continues after val seconds.

2    Returns the current speech speed in the range -10 to 10; 0 is the standard speed. If `val` is present, it sets the new speed.

3    Returns the current speech volume in the range 0 to 100, 100 is the standard volume. If `val` is present, it sets the new volume.

val    An optional argument used when `what%` is greater than 0.

Returns    If there is no speech support available, or a system error occurs, the command returns -1. Otherwise the first command variant returns 0 if all is well and the second variant returns the values listed for `what%`.

To use TTS (text to speech), you need a suitable sound card and the Microsoft SAPI software support. Windows XP and later has this software included with the operating system. You can get text to speech support as a download for other versions of Windows. In August 2006, the speech support was available as `SpeechSDK51MSM.exe` from the web page `www.microsoft.com/speech/download/sdk51/` but this location may change.

**See also:**
`Sound()`

# Spline2D()

Given a set of $(x_i, y_i)$ or $(x_i, y_i, z_i)$ positions and the values $v_i$ at each position, this command allows you to generate interpolated values at arbitrary (but local to the existing points) positions by interpolation. The interpolation is based on the distance of each interpolated point from the known points, so it can work in any number of dimensions, but we have implemented it in 2D $(x,y)$ and 3D $(x,y,z)$. We also provide functions to help in generating bitmaps of the result.

For example, in EEG work, you might have a set of measurements from electrodes attached to a head and you want to map the power in a particular frequency band (which is known at each electrode) across the entire head. In the example image the small circles are electrode positions at which the power in a particular frequency band is known and the colours represent power density with red as high and blue as low.
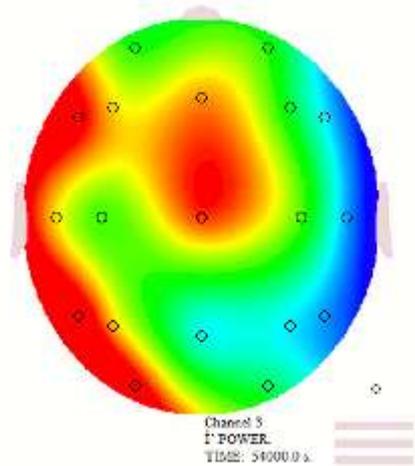
The `Spline2D()` function is useful for a reasonable number of $(x, y)$ positions (say a maximum of a hundred or so). The example picture uses 19.

The function uses two basic methods to map the data: Inverse distance weighting and Radial basis functions. Both methods use the distances $r_i$ of a general position $(x, y)$ or $(x, y, z)$ from each of the $i$ positions, which means that $x$ and $y$ (and $z$) must be measured in the same units. Put another way, this is not a suitable method to use if your $y$ position is distance and your $x$ position is time.

If you have data points on a rectangular grid (such as might occur when looking at the time course of the signals from a linear array), cubic splining treating the x and y axes as independent may be more appropriate.

The command has several variants, some of which are used to set up the problem, some to provide the data and some to obtain the interpolations. As this command will often be used to create bitmaps, we provide support for efficient mapping of levels to colours. The command are:

```
Func Spline2D(const p[][]);
Func Spline2D(const v[]{, mode%{, flags%{, v1}}});
Func Spline2D(x, y{, z});
Func Spline2D(out[], const pos[][]);
Func Spline2D(const map%[], mLo, mHi);
Func Spline2D(map$, mLo, mHi{, nMap%});
Func Spline2D(xy%[][], xLo, yLo, xHi, yHi);
Func Spline2D(xy[][], xLo, yLo, xHi, yHi);
Func Spline2D(xy%[][], const pos[][][]);
```

### Set (x, y) positions
This command variant must be called first and resets any remembered information including resetting the interpolation both `mode%` and `flags%` to 0. It sets the $(x, y)$ positions for use by the other command variants.

```
Func Spline2D(const p[][]);
```

p[][]   A matrix of `x,y` or `x,y,z` positions. That is the matrix is either `p[2][n]` for 2D interpolation or `p[3][n]` for 3D interpolation. `p[0][]` is a vector of x values, `p[1][]` is a vector of y values, and it it exists, `p[3][]` is a vector of z values.

Returns   It returns the number of positions that will be used or -1 if points are not distinct (different).

### Set the values and the interpolation method
This command variant is called after setting the positions. It sets the values at each of the positions and optionally sets the interpolation method and any required settings. It sets up the command so that it can compute the interpolations efficiently.

```
Func Spline2D(const v[]{, mode%{, flags%{, v1}}});
```

v[]    A vector of values. This vector must be at least as long as the number of positions. If there are n positions, the first n elements of v[] are used.

mode%  This determines the method used for interpolation. If omitted, the last set `mode%` is used. The reset value is 0. There is more information about the methods, below. The values currently implemented are:

  0  Inverse distance weighting. `v1` sets the inverse power to use. Positive values greater than 0 and less than 20 or so can be used. Large values may cause arithmetic underflow or overflow.

  1  Polyharmonic spline using $r$.

  2  Polyharmonic spline using $r^2 \ln(r)$. The distance $r$ is scaled by dividing by `v1`, which is taken as 1 if omitted.

  3  Polyharmonic spline using $r^3$.

  4  Polyharmonic spline using $r^4 \ln(r)$. The distance $r$ is scaled by dividing by `v1`, which is taken as 1 if omitted.

  5  Radial basis function using $\exp(-r^2/v1^2)$. The distance $r$ is scaled by dividing by `v1`, which is taken as 1 if omitted.

flags%  Flag settings control aspects of the interpolation (there is only one flag at present). If omitted, the flags take the last set value. The reset value is 0. The value is the sum of:

  1  Subtract the mean of the `z[]` values from the data before use, then add the mean back to the interpolated values. This affects how the results behave outside the locality of the supplied data. If the spline method tends to zero at large distances, this causes it to tend to the mean value.

v1  Optional parameter. If omitted it takes the last set value. The reset value is 1. Its use depends on `mode%`.

Returns  -1 if an error, otherwise the `mode%` value that will be used.

### Interpolate single value

This command variant returns the interpolated value at a designated point. It is an error to call this method without having previously set the positions and set the values and interpolation method.

```
Func Spline2D(x, y{, z});
```

x,y,z  The position to interpolate the result. The *z* value should not be present for a 2D interpolation and is required for a 3D interpolation. We do not restrict the range of positions, but you should be aware that using a position well away from the locality of the positions that define the spline may not generate useful results.

Returns  The interpolated value.

### Interpolate a list of values

This command variant fills a vector of values with interpolated values based on a list of positions passed in. This works with 2D or 3D positions. You could use this to interpolate values over a 3D surface mapped onto a plane. It is an error to call this without having set the positions and the interpolation values and method.

```
Func Spline2D(out[], const pos[][]);
```

out[]  A vector that is filled with interpolated values for each position in `pos`. The number of values returned is the shorter of this vector and the second dimension of pos. That is, `min(len(out, len(pos[0][]))`.

pos  A matrix that is either `2 x n` or `3 x n`, for 2D or 3D interpolation. `pos[0][]` is a vector of x values, `pos[1][]` is a vector of y values and for 3D interpolation, `pos[2][]` is a vector of z values.

Returns  The number of values written to out.

### Set colour mapping to create rectangular bitmap

These command variants fill a rectangular integer matrix with values that map from the interpolation. The expectation is that the mapping array will be filled with RGB colour values and that the `xy%[][]` array will be written as a bitmap.

The first two variants determine how interpolated values map to colours (or colour indices in a palette). You must call one of them before the calls that generate a rectangular bitmap.

```
Func Spline2D(const map%[], mLo, mHi);
Func Spline2D(map$, mLo, nHi{, nMap%));
```

map%[]   An integer vector of values that are to be used in the bitmap to represent interpolated values in a range determined by `mLo` and `mHi`. The formula used to determine the element of `map%` to use for an interpolated value *v* is:

```
mIndex% := (v - mLo) * (Len(map%)-1)/(mHi - mLo);
```

Values outside the range `mLo` to `mHi` are mapped to the nearer end of the map. You can use this mechanism to implement a variety of bitmap formats. For example, a simple approach is to use the 32-bit per pixel format with bits 0-7 for blue, 8-15 for green and bits 16-23 for red (with the option of using bits 24-31 for an alpha channel). Alternatively, if you were using a palette format, the `map%` array might only hold the integer values 0-255 for an 8-bit palette.

map$   Either empty, or the name of a colour scale. If empty, or if the name is not found, the colour scale identified by `mapInd%` (the index of the scale) is used.

nMap%   This is used as the item number of the colour scale if `map$` does not identify a colour map. Valid item numbers are currently in the range -2 to 20 (but not all numbers are valid). If this does not identify a map, a default monochrome map is used.

mLo   The interpolated value to map to the first element of the colour map. It is a fatal error for `mLo` to be the same as `mHi`.

mHi   The interpolated value to map to the last element of the colour map.

Returns   The variant that looks up a colour map returns 1 if a nominated map was found. Otherwise 0 is returned.

## Map into rectangular bitmap
The next variant is used in the 2D case to map a rectangular region of the (*x*, *y*) plane into a matrix such that each element of the output represents the colour of a pixel in a bitmap. You must have already set up the mapping of interpolated values into integers (set a map and `mLo` and `mHi`) and have called the routines to set up the interpolation.

**Func Spline2D(xy%[][], xLo, yLo, xHi, yHi);**

xy%   This is a matrix `xy%[nx%][ny%]` that is filled in by interpolating values, then mapping these values to an integer, where `nx%` is the number of x pixels from left to right and `ny%` is the number of y pixels. We only change bits 31-0 of the 64 bit values held in `xy%`. Further, if any of the bits 63-32 are set, we DO NOT CHANGE the value (or calculate it). This allows you to mask out areas of the output and to fill these areas (if you wish) with graphical elements that never change. If you do not want to be concerned with this, just make sure that the upper 32-bits of each element of `xy%` are zero.

xLo   The x position used to calculate the `xy%[0][]` elements of the matrix.

yLo   The y position used to calculate the `xy%[][0]` elements of the matrix.

xHi   The x position of the end of the range spanned by the matrix. The *x* increment per bin in the *x* direction is `(xHi - xLo)/nx%`.

yHi   The y position of the end of the range spanned by the matrix. The y increment per bin in the *y* direction is `(yHi - yLo)/ny%`.

Returns   The number of elements that were outside the range `mLo` to `mHi`.

## Map 3D or 2D to a rectangular bitmap
The next variant maps a 3D (or 2D) surface onto a 2D bitmap (for example to map interpolations of values on the surface of a human head onto a flat plane). You must have already set up the mapping of interpolated values into integers (`map%`, `mLo` and `mHi`) and have called the routines to set up the interpolation.

**Func Spline2D(xy%[][], const pos[][][]);**

xy%   This is a matrix `xy%[nx%][ny%]` that is filled in by interpolating values, then mapping these values to an integer, where `nx%` is the number of x pixels from left to right and `ny%` is the number of y pixels. The same comments about masking out values applies as for the rectangular map, above.

pos   This is either `pos[2][nx%][ny%]` for a 2D mapping or `pos[3][nx%][ny%]` for a 3D mapping. The first dimension holds the x, y (and z) co-ordinates in the `pos[0][][]`, `pos[1][][]` (and `pos[2][][]`) elements. The numbers of elements in each dimension must be large enough to match the `xy%` matrix and to match the 2D or 3D mapping.

Returns  The number of elements that were outside the range `mLo` to `mHi`.

### Interpolate to rectangular grid

The next variant is used in the 2D case to map a rectangular region of the (*x*, *y*) plane into a matrix such that each element of the output represents the interpolated value at that grid position. You must have called the routines to set the positions, values and interpolation method. This is equivalent to using the interpolate single value code over a rectangular grid.

```
Func Spline2D(xy[][], xLo, yLo, xHi, yHi);
```

xy%      This is a matrix `xy[nx%][ny%]` that is filled in by interpolating values, where `nx%` is the number of x pixels from left to right and `ny%` is the number of y pixels.

xLo      The x position used to calculate the `xy%[0][]` elements of the matrix.

yLo      The y position used to calculate the `xy%[][0]` elements of the matrix.

xHi      The x position of the end of the range spanned by the matrix. The *x* increment per bin in the *x* direction is `(xHi - xLo)/nx%`.

yHi      The y position of the end of the range spanned by the matrix. The y increment per bin in the *y* direction is `(yHi - yLo)/ny%`.

Returns  0.

# Inverse distance weighting

This is a relatively simple-minded form of interpolation, which weights the contribution of each known point to the result at any arbitrary point by some negative power of the distance to the known point. The basic idea is that if $d_i$ is the distance to the *i*th known value $v_i$, the interpolated value is:

$$\sum v_i/d_i^p \quad / \quad \sum 1/d_i^p$$

The larger the value of p, the more the effect of local points dominates the result (*p* need not be integral). High values (16, for example), tend to generate hard edges in the interpolation along the lines joining the known points. You can see examples of this here.

This formula appears to break down as you approach any of the known points due to division by zero, but you can see that as you get very close to any point *n* you can ignore the contribution of all other points (as $d_n << d_i$ for all other points), so the formula becomes $v_n/d_n^p \quad / \quad 1/d_n^p$ which simplifies to just $v_n$.

# Radial basis functions

The idea here is that given our fixed points, to place values $w_i$ (to be calculated) at each of the fixed points such that the interpolated value at a point that is a distance $r_i$ from each of the fixed points is given by:

$$\sum w_i f(r_i)$$

The $w_i$ are calculated such that the interpolation passes through each of the fixed points.

The *f(r)* are known as radial basis functions, and many are possible. We implement a few that are likely to be useful. The first 4 in the list are also known as Polyharmonic Splines (*r^n* for *n* odd, *r^n ln(r)* for *n* even).

$r$           This is a linear interpolation between the calculated points.

$r^2\ln(r)$     This is a Polyharmonic spline that is also known as the Thin plate spline. This has several nice properties that make it a good choice for smooth splining.

$r^3$          Polyharmonic spline.

$r^4\ln(r)$    Polyharmonic spline.

$\exp(-r^2/\sigma^2)$     Gaussian. Setting a sensible σ value is important. Too small a value results in spikes at each fixed data point.

The generation of the wi values involves inverting a matrix (which is an $O(n^3)$ process, with n the number of fixed points, another reason to keep the number of fixed points within reason). If your input data points all have the same sign and are all a long way from zero, you may be able to improve the results by setting the `flags%` value to remove the mean value from the data before interpolation, then add it back after.

# Spline2D example

The following example code shows how you can generate a bitmap from scattered data points. In this case, we have 4 data points (the black squares) with known values of 1, -1, 1 and -1 (clockwise from the top left). The data is interpolated using the "thin plate spline" mechanism using red to represent positive values and blue to represent negative values.



The image to the right shows the result. To interpret this, the darker the red, the more the interpolating surface is raised. The darker the blue, the more the interpolating surface is lowered. The white cross represents the places where the interpolating surface passes through 0.

The code required to set up the problem is:

```
const p[][] := {{-0.5, 0.5}, {0.5, 0.5}, {0.5, -0.5}, {-0.5, -0.5}};
Spline2D(p);                  'set the initial point positions (square)
const v[] := {1, -1, 1, -1};'the data values for each of the positions
Spline2D(v, 2);              'set point values in "thin plate spline mode"
```
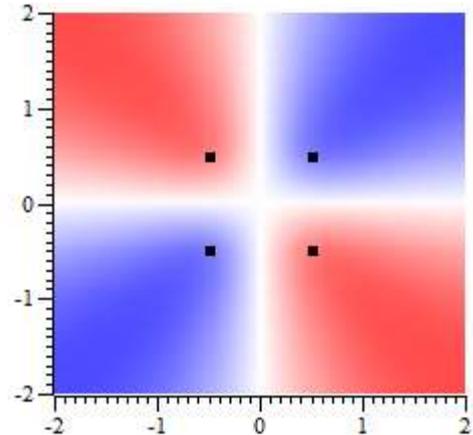
We could now use the `Spline2D(x, y)` calls to obtain the splined values as any desired *(x, y)* position. However, we want to generate a bitmap, so the next step is to generate a map from data values to colours. To do this, we generate a list of integer values, each value representing a colour in RGB format and we set the colours to span a range of values. Colours are coded from 0 (off) to 255 (`0xff`, full on) as:

```
blue% + green%*0x100 + red%*0x10000
```

Hexadecimal notation is convenient as it uses 2 digits for each colour. Black is `0x000000`, bright blue is `0x0000ff`. Bright green is `0x00ff00` and bright red is `0xff0000`. White is `0xffffff`. Setting the same proportion of red, green and blue generates a grey scale. The following code sets up a map from bright blue through white to bright red. We set this to map the range -2 to +2.

```
var col%[511], i%;
for i% := 0 to 255 do col%[i%] := 0xff + i%*0x10100 next;
for i% := 256 to 510 do col%[i%] := 0xff0000 + (510-i%)*0x101 next;
Spline2D(col%, -2, 2);      'colour map blue - white - red for -2 to +2
```

The next task is to map this into a bitmap and then save it in a file. The data points form a square with side 1 centred at 0,0 and we choose to display the mapping from (-2,2) to (2,2) using a 200 x 200 pixel bitmap. We save the bitmap to test.bmp in the folder My Documents\Signal7. You can find the code for `WriteBmp()` in the `BWriteSize()` script command example (so we don't repeat it here). We know that the `bmp%` matrix will be initialised to 0, so we do not need to worry about having some of the data masked out.

```
const xLo := -2, xHi := 2, yLo := -2, yHi := 2;
var bmp%[200][200];           'bitmap space 200 x 200 pixels
Spline2D(bmp%, xLo, yLo, xHi, yHi); 'make a bitmap
var path$ := FilePath$(-4) + "test.bmp"; 'Where to save a bitmap
WriteBmp(path$, bmp%, 0);    'See BWriteSize example for code
```

Finally, we create an XY view and set the newly created bitmap as the channel image. We then scale the image so that it spans the same x and y axis range as the bitmap and add data points for the positions where the data values are known.

```
var xy% := FileNew(12, 1);    ' New XY view
XRange(xLo, xHi);             ' Set axis range to span
YRange(1, yLo, yHi);          ' the data
ChanImage(1, path$);          ' Link bitmap to the channel and set...
ChanImage(1, 2, 1.00, xLo, yLo, xHi, yHi); '...to span the data range
XYAddData(1, p[0][], p[1][]); 'Mark data positions
```

The mapping operation is quite fast. On my computer it took around 8 milliseconds, so it is entirely possible to use this technique to generate an animated display. The time is proportional to the number of pixels and to the number of known data points. If you had 20 known points (a more reasonable number), it would likely take 5 times longer than this simple example with 4 known points.

# Sqrt()

Forms the square root of a real number or an array of real numbers. Negative numbers halt the script with an error when `x` is not an array. With an array, negative numbers are set to 0 and an error is returned.

```
Func Sqrt(x|x[]{[]...});
```

x        A real number or a real array to replace with an array of square roots.

Returns  With an array, this returns 0 if all was well, or a negative error code. With an expression, it returns the square root of the expression.

**See also:**
```
Abs(), ATan(), Cos(), Exp(), Frac(), Ln(), Log(), Max(), Min(), Pow(), Rand(),
Round(), Sin(), Tan(), Trunc()
```

# Str$()

This converts a number to a string.

```
Func Str$(x{, width%{, sigd%}});
```

x        A number to be converted.

width%   Optional minimum field width. The number is right justified in this width.

sigd%    Optional number of significant figures in the result (default is 6) or set a negative number to set the number of decimal places.

Returns  A string holding a representation of the number.

**See also:**
```
Asc(), Chr$(), DelStr$(), InStr(), LCase$(), Left$(), Len(), Mid$(), Print$(),
Print(), Right$(), UCase$(), Val()
```

# StrToChanY()

This function lets you evaluate a text string in exactly the same way that a dialog expression evaluates a channel expression. It is here to allow us to test dialog expressions from a script, but it could occasionally be useful from a script.

```
Func StrToChanY(expr$, chan%, &value);
```

expr$    The channel dialog expression to evaluate.

chan%    The channel to evaluate it for or 0 if no channel is needed. Expressions involving only horizontal cursors do not need a channel as each horizontal cursor is attached to a channel.

value    A real variable returned holding the result of a successful evaluation.

Return   1 for success, 0 if the expression could not be evaluated.

**See also:**
```
StrToViewX()
```

# StrToViewX()

This function lets you evaluate a text string in exactly the same way that a dialog expression evaluates a view expression. It is here to allow us to test dialog expressions from a script, but it could occasionally be useful from a script.

```
Func StrToViewX(expr$, &value);
```

expr$  A view x axis expression to be evaluated.

value  A real variable returned holding the result of a successful evaluation.

Return  1 for success, 0 if unable to evaluate the expression.

**See also:**
StrToChanY()

# Sweeps()

This function returns the number of sweeps accumulated into the frame data and optionally sets it to a new value. If the memory view is saved and reloaded as a file view, the sweeps value is preserved. What each item or sweep is, depends on the type of the analysis.

```
Func Sweeps({new});
```

new  If present, sets the sweeps value for the frame to a new value. Note that, while setting the sweep count for a frame is necessary in some circumstances, if used incautiously this mechanism will corrupt the sweep count of analysed data.

Returns  The number of sweeps accumulated to produce the frame data.

**See also:**
SetAverage(), SetPower(), View()

# System()

This function returns the operating system version as a number and gets information about desktop screens and about the time consumed by the GUI thread. Use the App() command to get the version number of Signal.

```
Func System({get%{, scr%{, sz%[]}}});
Func System(3, &user, &kernel);
```

get%  A value that determines which information to return. If omitted, the value is taken as 0. Values 2 and 3 were added at Signal version 7.05.

0  The return value is the operating system revision times 100: 351=NT 3.51, 400=95 and NT 4, 410=98, 490=Me, 500=NT 2000, 501=XP, 502=Windows Server 2003, 600=Vista, Windows Server 2008, 601=Windows Server 2008 R2, Windows 7, 602=Windows 8, Windows Server 2012, 1000=Windows 10. Signal version 7 requires Windows 7 or later (it may work in Vista with the correct libraries installed).

1  The function returns information about installed desktop monitors (see scr% and sz%[])

2  Set the base time for collecting the time used by the current GUI thread. There must only be one argument. This is for CED diagnostic use. If you do not use this, the value read back by setting get% to 3 will be relative to the start of Signal (not to the start of the current script).

3  Read back the time used in user and kernel mode of the GUI thread in seconds. There must be three arguments. This is for CED diagnostic use.

scr%  Set 0 to return the number of desktop monitors; sz%[] gets the pixel co-ordinates of the desktop. Set to n (>0) to get the pixel co-ordinates of screen n; returns 1 for the primary monitor, 0 if not and -1 if it does not exist. Add 1000 to scr% to get the pixel co-ordinates of screen n with any areas reserved by the system (for example the task bar) removed.

sz%[]  Optional array of at least 4 elements to return pixel positions. Elements 0 and 1 hold the top left x and y, 2 and 3 hold bottom right x and y.

user  Returned holding the time in seconds that the GUI thread has executed for in user mode since the last call to System(2). The result is pretty much meaningless if you do not call System(2) first.

kernel  Returned holding the time in seconds that the GUI thread has executed for in kernel mode since the last call to System(2).

Returns  A value that depends on `get%` as described above.

**See also:**
```
App(), System$(), Window(), WindowVisible()
```

# System$()

This returns the operating system name and accesses Signal environment variables and the program command line. You can also read all the strings into a string array.

### Get operating system name
To get information about the operating system use the command with no arguments:

```
Func System$()
```

Returns  With no arguments, it returns: `"Windows SS build n"` where `SS` is the operating system and `n` is the build number.

### Get and optionally set an environment variable
Each process has its own environment. The environment holds a list of strings of the form `"name=value"`. You can get or set the `value` associated with `name`. A process started with `ProgRun()` inherits a copy of the Signal environment, so you can pass it information. However, you cannot see environment changes made by the new process.

```
Func System$({var$ {,value$}});
```

`var$`    If present, this is the name of an environment variable (case insensitive).

`value$`  If present, the new value. An empty string deletes the environment variable.

Returns  The value of the environment variable identified by `var$` or an empty string if this variable does not exist.

### Get a list of environment variables
This command variant reads the environment strings into a string array.

```
Func System$(list$[] {,&n%});
```

`list$`   An array of strings to fill with environment strings of the form `"name=value"`.

`n%`      An optional integer that is returned holding the number of elements copied.

Returns  An empty string.

### Get elements of the command line that started Signal
Signal can be started with a command line and it can be useful to find out what this held. In particular, Signal ignores command line switches (items starting with a - or / character) that it does not recognise, so a script run using the command line can then interpret other command line switches as defined by the user.

```
Func System$(cmd%);
```

`cmd%`    Used to return the elements that make up the program command line used to start Signal. The first element is 0, the second 1 and so on. If a command element is a flag (introduced by - or /), it starts with a /. The return value is an empty string if the element does not exist. This command line variant was added at Signal version 7.05.

Returns  The element of the command line indexed by `cmd%` or an empty string if there is no such element.

**Examples of use**
```
var list$[200], value$, n%, i%;
PrintLog("%s\n", System$());  'Print OS name
System$("fred","good");       'Assign the value "good" to fred
PrintLog("%s\n", System$("fred")); 'get value of fred
System$("fred","");           'Delete fred from the environment
System$(list$[], n%);         'Print all environment strings
for i%:=0 to n%-1 do PrintLog("%s\n",list$[i%]) next;
```

**See also:**
ProgRun(), System()

# T

# TabSettings()

This sets and gets the tab settings for a text view. Any changes you make apply to the current view only. If you want to change the tab settings for all views, open the Display page of the Preferences dialog and click the appropriate button in the Text view settings. It is possible to do this using a script with the Profile() command.

**Func TabSettings({size%{, flags%}});**

size%    Tab sizes are set in units of the width of the space character in the Default style set for the view (style 32). Values in the range 1 to 100 set the tab size. If size% is 0 or omitted, no change is made. If size% is -1, the return value is the current flags% value for the text view.

flags%   If omitted, no change is made to the flags. Otherwise, this is the sum of flag values: 1=Keep tab characters (the alternative is replace tabs with spaces), 2=show indents.

Returns   If size% is positive, the return value is the tab size at the time of the call. If size% is -1, the return value is the flags% value at the time of the call.

**See also:**
FontGet(), FontSet(), Profile()

# Tan()

This calculates the tangent of an angle in radians or converts an array of angles into tangents. Tangents of odd multiples of pi/2 are infinite, so cause computational overflow. There are 2pi radians in 360degrees. pi is approximately 3.14159265359 (4.0*ATan(1)).

**Func Tan(x|x[]{[]...});**

x        The angle, expressed in radians, or a real array of angles. The best accuracy of the result is obtained when the angle is in the range -2pi to 2pi.

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:**
Abs(), ATan(), Cos(), Exp(), Frac(), Ln(), Log(), Max(), Min(), Pow(), Rand(), Round(), Sin(), Sqrt(), Trunc()

# Tanh()

This calculates the hyperbolic tangent of a value or an array of values.

**Func Tanh(x|x[]{[]...});**

x        The value or an array of real values.

Returns   For an array, it returns 0. Otherwise it returns the hyperbolic tangent of x.

**See also:**

```
Abs(), ATan(), Cos(), Cosh(), Exp(), Frac(), Ln(), Log(), Max(), Min(), Pow(),
Rand(), Round(), Sin(), Sinh(), Sqrt(), Trunc()
```

# Time$()

This function returns the current system time of day as a string. If no arguments are supplied, the returned string shows hours, minutes and seconds in a format determined by the operating system settings. To obtain the time as numbers, use `TimeDate()`. To obtain relative time and fractions of a second, use `Seconds()`.

```
Func Time$({tBase%{, show%{, amPm%{, sep$}}}});
```

`tBase%`  Specifies the time base to show the time in. You can choose between 24 hour or 12 hour clock mode. If this argument is omitted, 0 is used.

0  Operating system settings 2  12 hour format
1  24 hour format

`show%`  Specifies the time fields to show. Add the values of the required options together and use that as the argument. If this argument is omitted or a value of 0 is used, 7 (1+2+4) is used for 24 hour format and 15 (1+2+4+8) for 12 hour format.

1 Show hours    4 Show seconds
2 Show minutes   8 Remove leading zeros from hours

`amPm%`  This sets the position of the "AM" or "PM" string in 12 hour format and has no effect in 24 hour format. If omitted, a value of zero is used. The string that gets printed ("AM" or "PM") is specified by the operating system.

0 Operating system settings 2 Show to the left of the time
1 Show to the right of the time 3 Hide the "AM" or "PM" string

`sep$`  This string appears between adjacent time fields. If `sep$` = ":" then the time will appear as `12:04:45`. If an empty string is entered or `sep$` is omitted, the operating system settings are used.

**See also:**
```
Date$(), Seconds(), TimeDate()
```

# TimeDate()

This returns the time and date in seconds, minutes, hours, days, months, and years plus the day of the week. You can use separate variables for each field or an integer array. To get the data or time as a string, use `Date$()` or `Time$()`. To measure relative times, or times to a fraction of a second, see the `Seconds()` command. To get the current sampling time, see `MaxTime()`.

```
Proc TimeDate(&s%{, &m%{, &h%{, &d%{, &mon%{, &y%{, &wDay%}}}}}});
Proc TimeDate(now%[])
```

`s%`  If `s%` is the only argument, it is set to the number of seconds since midnight. Otherwise it is set to the number of seconds since the start of the current minute.

`m%`  If this is the last argument, it is set to the number of minutes since midnight. Otherwise it is set to the number of full minutes since the start of the hour.

`h%`  If present, the number of hours since Midnight is returned in this variable.

`d%`  If present, the day of the month is returned as an integer in the range 1 to 31.

`mon%`  If present, the month number is returned as an integer in the range 1 to 12.

`y%`  If present, the year number is returned here. It will be an integer such as 2008.

`wDay%`  If present, the day of the week will be returned here as 0=Monday to 6=Sunday.

`now%[]`  If an array is the first and only argument, the first seven elements are filled with time and date data. The array can be less than seven elements long. Element 0 is set to the seconds, 1 to the minutes, 2 to the hours, and so on.

**See also:**

```
Date$(), MaxTime(), Seconds(), Time$()
```

# TimeRatio()

This function returns the ratio between the current view X axis units and seconds; for example in milliseconds mode it returns 1000. Use of this value allows script output to use the preferred time units, as the script functions always see time values in seconds, regardless of the time units preferred.

`Func TimeRatio();`

Returns  The current time ratio.

**See also:**
`TimeUnits$()`

# TimeUnits$()

This function returns the current view time units, for example in milliseconds mode it returns `"ms"`. Use of this allows script output to show the preferred time units.

`Func TimeUnits$();`

Returns  The current time units.

**See also:**
`TimeRatio()`

# The toolbar

The toolbar is at the top of the screen, below the menu. The bar has a message area and can hold buttons that are used in the `Interact()` and `Toolbar()` commands.

It is possible to link user-defined functions and procedures to the toolbar buttons. This is done through a set of functions that define buttons and optionally link the buttons to the toolbar. You can define up to 40 buttons in your toolbar, but you will probably be limited by the available space to a maximum of around 10. Buttons are numbered from 1 to 40. There is an invisible button, numbered 0, that is used to set a function that is called when the toolbar is waiting for a button to be pressed.

When you start a script, the toolbar is invisible and contains no buttons. When a script stops running, the toolbar becomes invisible (if it was visible).

### Toolbar building
There is an example script `Toolmake.sgs` which automates the writing of toolbar commands to generate your desired toolbar.

**See also:**
`Interact(), Toolbar(), ToolbarClear(), ToolbarEnable(), ToolbarSet(), ToolbarText(), ToolbarVisible()`

# Toolbar()

This function displays the toolbar and waits for the user to click on a button. If button 0 has been defined with an associated function, that function is called repeatedly while no button is pressed. If no buttons are defined or enabled, or if all buttons become undefined or disabled, the toolbar is in an illegal state and an error is returned. If the toolbar was not visible, it becomes visible when this command is given.

If the user presses the "escape" key (Esc) with the toolbar active, the script will stop unless an "escape button" has been set by `ToolbarSet()`, in which case the action associated with that button is performed.

`Func Toolbar(text$, allow%{, help%|help$});`

text$  A message to display in the message area of the toolbar. The area available for messages competes with the area for buttons. If there are too many buttons, the message may not be visible.

allow% A code that defines what the user can do (apart from pressing toolbar buttons). The code is the sum of possible activities:

| | | |
|---|---|---|
| 1 | 0x0001 | User may swap to other applications |
| 2 | 0x0002 | User may change the current window |
| 4 | 0x0004 | User may move and resize windows |
| 8 | 0x0008 | User may use File menu |
| 16 | 0x0010 | User may use Edit menu |
| 32 | 0x0020 | User may use View menu |
| 64 | 0x0040 | User may use Analysis menu |
| 128 | 0x0080 | User may use Cursor menu and add cursors |
| 256 | 0x0100 | User may use Window menu |
| 512 | 0x0200 | User may use Sample menu |
| 1024 | 0x0400 | User may not double click y axis |
| 2048 | 0x0800 | User may not double click the x axis or scroll it |
| 4096 | 0x1000 | User may not change channel of horizontal cursors |
| 8192 | 0x2000 | User may not change to another frame |

A value of 0 would restrict the user to the current view in a fixed window and, in a data view, the user would be able to scroll data and switch frames. A value of 9216 is the same as 0 but without being able to change y axes or frame. The `allow%` value used remains in force all the time the toolbar remains active.

help This is either a numeric code or a string that defines the help to be presented if the user asks for it while using the toolbar. A code of 0 means use standard help.

Returns The function returns the number of the button that was pressed to leave the toolbar, or a negative code returned by an associated function.

The buttons are displayed in order of their item number. Undefined items leave a gap between the buttons. This effect can be used to group related buttons together.

**See also:**
`Interact(), ToolbarClear(), ToolbarEnable(), ToolbarSet(), ToolbarText(), ToolbarVisible()`

# ToolbarClear()

This function removes some or all of the toolbar buttons. If you delete all buttons, the `Toolbar()` function inserts an OK button, so you can get out of the `Toolbar()` function. Use `ToolbarText("")` to clear the toolbar message.

```
Proc ToolbarClear({item%});
```

item% If present, this is the button to clear. Buttons are numbered from 0. If omitted, all buttons are cleared. If the toolbar is visible, changes are shown immediately.

**See also:**
`Interact(), Toolbar(), 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(but%{, state%});
Func ToolbarEnable(const but%[], state%);  (new at 6.03)
```

but% The number of the button or -1 for all buttons. You must enable and disable button 0 with `ToolbarSet()` and `ToolbarClear()`.

but%[] An array of button numbers. This allows you to set the state of several buttons with a single call. If you use this array variant of the command, you must provide the `state%` argument.

state% If present this sets the button state. 0 disables a button, 1 enables it, -1 leaves the state unchanged.

Returns The function returns the state of the button (or the first button in the list in the array variant of the command) prior to the call, as 0 for disabled and 1 for enabled. If `but%` or `but%[0]` was -1, the function returns 0 before version 7.06 and now returns the count of enabled buttons. If an undefined button is selected, the function returns -1. Before version 7.06 if button 0 was specified this always returned 0, now it returns -1 if no Idle function is set and 1 if one is.

**See also:**
`Interact()`, `Toolbar()`, `ToolbarClear()`, `ToolbarSet()`, `ToolbarText()`, `ToolbarVisible()`

# ToolbarMouse()

This command gives you access to the mouse positions and left button mouse clicks in Data and XY views when the mouse is over a data channel while a toolbar is active. There is an example, here. The description of this command applies also to DlgMouse() when used with 5 or more arguments.

```
Proc ToolbarMouse(vh%, ch%, mask%, want%, Func Down%{, Func Up%{, Func Move
%}});
```

vh% This is either the view handle of the view that you want to get mouse information from, or 0, meaning that you will accept mouse information from any suitable view.

ch% This is either a channel number in the view that you want mouse information for when you are over it or 0 to accept input from any channel. In an XY view, the display area is treated as belonging to channel 1, so setting 1 or 0 will work. If you set a channel number, once you have clicked on that channel, all values passed to you will be for that channel, even if you drag the mouse over a different channel. If you want to be able to click on a channel, then drag to another and be told about the other channel you must set `ch%` to 0. With `ch%` set to 0, if you drag to a place where there is no channel, you will be returned the last position that was over a channel.

mask% This, and the next argument (`want%`) are used when the left mouse button is clicked to decide if the script should be told about the mouse click. When the mouse button is clicked, and the conditions set here are met, the mouse becomes owned by the script and all mouse input will be given to the script until the mouse is released (or another application grabs the mouse). The mouse is said to be captured. The conditions set here are also used to decide if the script should be informed of mouse movements when the mouse is not owned by the script. Both `mask%` and `want%` are the sum of a set of values:

1   The left-hand mouse button is down.

2   The right-hand mouse button is down (releasing this button will normally display a context menu)

4   The Shift key is down

8   The Ctrl key is down

16   The mouse middle button is down

32   Extra button 1 is down (this is the left-hand side button on my mouse)

64   Extra button 2 is down (this is the right-hand side button on my mouse)

128  The Alt key is down

256  There was a left mouse button double-click

The `mask%` value determines which of these items we care about. For example if you cared about the state of the `Shift` and `Ctrl` keys, you would set the value to 12.

want% This argument sets the desired state of the items that you have identified with `mask%`. For example, if you only want to be told when the `Shift` key is down and the `Ctrl` key is not down, set `mask%` to 12 and `want%` to 4. Another use would be to stop the script being told when the mouse was just being moved around but had not been clicked in an area we wanted. In this case you would set `mask%` to 1 and `want%` to 1 (only tell me when the left-hand mouse button is down).

Down% This is the name of a user-defined function that is called when the mouse left-hand button is clicked and the conditions implied by `vh%`, `ch%`, `mask%` and `want%` are satisfied. The arguments and return value of this function are described below.

Up%    This is the name of an optional user-defined function that is called when the mouse button is released after it has been captured. You will always get a `Down%` function call before you get an `Up%` call. If another application (rudely) takes over the mouse by popping up a window, you will also get a call to the `Up%` function. This function is described below.

Move%   This is the name of an optional user-defined function that is called when the mouse is moved after being captured. This function is also called when the mouse is moved when not captured by a mouse click and the conditions set by `vh%`, `ch%`, `mask%` and `want%` are satisfied. If you only want to be called during a drag operation, make sure you include the value for the left-hand mouse button in both `mask%` and `want%`.

### The user-defined mouse functions

All three functions have exactly the same arguments. The function names do not have to be `Down%`, `Up%` and `Move%`, you can choose any names that are suitable. Ideally your mouse functions (especially mouse move) should not take a long time to run; if they do, the mouse movement will feel uncomfortable and jerky. If you must trigger a more time-consuming operation, set a flag and then service it in an idle routine. The mouse pointer and any screen drawing related to the mouse is not acted upon until after your function returns, so you may get odd screen effects if you have a time-consuming function that causes screen repainting. The return value has different uses in all three cases. The functions are:

```
Func Down%(vh%, ch%, x, y, flags%);
Func Move%(vh%, ch%, x, y, flags%);
Func Up%(vh%, ch%, x, y, flags%);
```

vh%     The view handle of the view that the mouse is over. If you set `vh%` to a view handle value in the `ToolbarMouse()` call, then this will be that value. Your function must not close this view (the view is required to handle the return value from this function).

ch%     The channel number that the values of x and y relate to. If you specified a channel number in the `ToolbarMouse()` call, then this will be that value.

x       The x-axis value in x-axis units. If you click and drag you can get values that are outside the visible range of the x axis. If you want to scroll the view in response to this, you can do so.

y       The y-axis value in y axis units for the channel identified by `ch%`. If there is no y axis, the value will be 0.

flags%  This holds the same information as held by `mask%` and `want%`. It gives you the state of the mouse buttons and `Shift`, `Ctrl` and `Alt` keys.

### Return value

The return values have different uses for the three functions:

**Func Down%()**

If you decide that you do not want to do anything with the mouse, for example the click was not over anything interesting, then return 0 and Signal will decide what to do with the click. You will never get a mouse down call when the mouse is over the XY view key, or over a vertical or horizontal cursor. However, you do get priority over Signal for all other clicks in a view (for sizing, for instance). Return values greater than 0 select the mouse pointer to display (this is covered below) and mean that you want to capture the mouse for script use.

You can also choose to display an indication of a selection size or area by adding a value to the return value (only 1 value can be added). If you add any of the following values and you drag beyond the left or right edges of the data area, the area will scroll (if it is allowed to).

Value   Result

256     Display a selection rectangle, as you would for zooming in and out. If you return 256, the appropriate zoom cursor (16 or 17) is selected based on the state of the Ctrl key.

512     Display a measurement of the distance between the start of the drag and the current position. If you return 512, the measurement cursor (19) is selected.

1024    Display a line from the selection start to the current position. If you return 1024, the measurement cursor is selected.

**Func Move%()**

The return value from this sets the mouse pointer to be used (see below). A return value of 0 makes no change to the mouse pointer, which means that the pointer set by `Func Down%()` will be used for a drag operation and the cross-hair cursor (#1 in the list below) used when not dragging.

**Func Up%()**

The return value from this determines if the toolbar (or user-defined dialog for DlgMouse()) closes or not when the mouse button is released. Return a negative value or 0 to close the toolbar or dialog, 1 to keep the toolbar or dialog running. When used with a dialog a negative value acts as Cancel, while 0 acts as OK. When this function closes the toolbar or dialog, the result of Toolbar() or DlgShow() will be the value returned by Up% for a negative return value, or a value greater than the highest valid button number if zero is returned.

### Mouse pointers set by return values

You have access to many of the mouse pointers that are available in Signal, so you can use these to indicate that you are over an item or to show that you are dragging something. The values for the preset mouse pointers are:
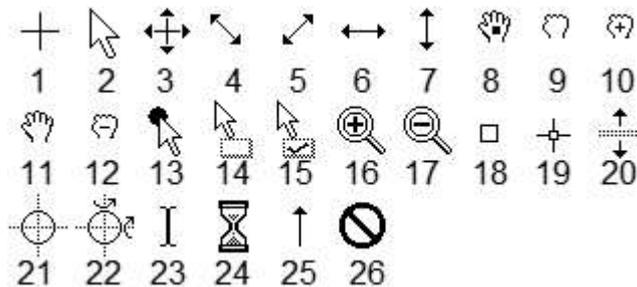


The first seven and the last four of these are system mouse pointers and may have a different appearance if you have chosen a custom set of mouse pointers in the Windows system Control Panel in the Mouse section. You should be aware that 3D and animated mouse pointers can be a lot slower on some systems than simple monochrome pointers. You can also define your own mouse pointers with the `MousePointer()` command. These are assigned numbers above the range of the built-in cursors.

### Mouse double-click behaviour

If you double-click, you will get a mouse down, possibly followed by one or more mouse moves, followed by a mouse up for the first click. The second click generates another mouse down, but this time the double-click flag will be set. It is up to you to decide if you want to undo anything that happened as a result of the initial mouse down and up. If your mouse down function returns 0, the double click will be handled by Signal.

### Things you should not do in a user-defined mouse function

The code in the mouse functions runs as part of the Windows code that decides which mouse pointer to display. There are a few things that you must not do inside the mouse functions:

1. Calling `FileClose()` on the view that is handling the mouse move or button notification will not close the view. This is because if it did close the view, when the mouse function returned control back to the deleted window, the program would crash.

2. Do not use up a lot of time, especially in a mouse move routine. Time consuming mouse routines will make mouse movement and button clicks feel awkward.

3. Mouse move calls after the mouse has been captured should not display user-defined dialogs or use the `Message()` or `Input()` or `Input$()` commands. If you do use these functions, you will find that the mouse is not available as it belongs to the window where the mouse was clicked. The keyboard can be used to navigate the dialog, but users will be very confused and may think that the program has crashed as mouse clicks in the dialog will have no effect.

### Debugging user-defined mouse functions

You can set a break point in the mouse down, move and up functions. However, if you do, the function will not control the mouse pointer (as it is in use for debugging) and the mouse up function will not terminate the Toolbar() call if it returns 0.

### See also:

DlgMouse(), MousePointer(), Toolbar(), ToolbarMouse() example

# ToolbarMouse() example

This example assumes that there is a data or XY view open. The three mouse routines just print information to the log window. The `ToolbarMouse()` call is setup to accept any suitable window and any channel in that window. The `mask%` and `want%` arguments are set so that the mouse left-hand button must be down. This prevents the mouse move function being called unless the mouse is down.

The mouse down function requests mouse pointer 19 (the measurement pointer) and adds 1024, which causes a line to be drawn between the mouse down position and the current position.

The mouse move function returns 0, which will leave the mouse pointer unchanged. If there was no need to do anything in this routine it could be omitted. We have included it so we can print the mouse information to the log window.

The mouse up function returns 1, so that the toolbar continues to run.

```
func MouseDown%(vh%, chan%, x, y, flags%)
PrintLog("Down: chan=%2d, x=%g, y= %g, flags% = %d\n", chan%, x, y, flags%);
return 19+1024; 'cursor 19 + a line linking start to end
end;

func MouseMove%(vh%, chan%, x, y, flags%)
PrintLog("Move: chan=%2d, x=%g, y= %g, flags% = %d\n", chan%, x, y, flags%);
return 0;        'keep same cursor as for the mouse down
end;

func MouseUp%(vh%, chan%, x, y, flags%)
PrintLog("Up: chan=%2d, x=%g, y= %g, flags% = %d\n", chan%, x, y, flags%);
return 1;        'do not close the toolbar
end;

ToolbarSet(1, "Hello");
ToolbarMouse(-1, -1, 1, 1, MouseDown%, MouseUp%, MouseMove%);
Toolbar("Hello", 511);
```

# 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. You can also use this command while the toolbar is displayed to get the last button pressed (added at version 4.06). There are two command variants:

```
Func ToolbarSet(item%, label${, func ff%()});
Func ToolbarSet();
```

item%   The button number in the range 1 to 40 to add or replace or 0 to set or clear a function that is called repeatedly while the toolbar waits for a button press.

You can set an "escape" key as described in `Toolbar()`, by negating `item%`. For example `ToolbarSet(-2,"Quit");` sets button 2 as the escape key.

label$  The button label plus optional key code and tooltip as `"Label|code|tip"`. Labels compete for space with each other; use tooltips for lengthy explanations. The label is ignored for button 0. Tooltips can be up to 79 characters long. To use a tooltip with no code use `"Label||A tooltip with no code field"`.

To link a key to a button, place `&` before a character in the label or add a vertical bar and a key code in hexadecimal (e.g. 0x30), octal (e.g. 060) or decimal (e.g. 48) to the end of the label. Characters set by `&` are case insensitive. For example `"a&Maze"` generates the label aMaze and responds to `m` or `M`; the label `"F2:Go|0x71"` generates the label F2:Go and responds to the `F2` key. Useful key codes include (nk = numeric keypad):

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0x08 | Backspace | 0x22 | Page down | 0x28 | Down arrow | 0x6b | nk + |
| 0x09 | Tab | 0x23 | End | 0x2e | Del | 0x6c | nk separator |
| 0x0d | Enter | 0x24 | Home | 0x30-39 | 0-9 | 0x6d | nk - |
| 0x1b | Escape | 0x25 | Left arrow | 0x41-5a | A-Z | 0x62 | nk . |
| 0x20 | Spacebar | 0x26 | Up arrow | 0x60-69 | nk 0-9 | 0x6f | nk / |
| 0x21 | Page up | 0x27 | Right arrow | 0x6a | nk * | 0x70-87 | F1-F24 |

Use of other keys codes or use of `&` before characters other than a-z, A-Z or 0-9 may cause unpredictable and undesirable effects.

Beware: When the toolbar is active, it owns all keys linked to it. If `A` is linked, you cannot type `a` or `A` into a text window with the toolbar active.

`ff%()` This is the name of a function with no arguments. The name with no brackets is given, for example `ToolbarSet(1,"Go",DoIt%);` where `Func DoIt%()` is defined somewhere in the script. When the `Toolbar()` function is used and the user clicks on the button, the linked function runs. If the `item%` 0 function is set, that function runs while no button is pressed. The function return value controls the action of `Toolbar()` after a button is pressed.

If it returns 0, the `Toolbar()` function returns to the caller, passing back the button number. If it returns a negative number, the `Toolbar()` call returns the negative number. If it returns a number greater than 0, the `Toolbar()` function does not return, but waits for the next button. An item 0 function must return a value greater than 0, otherwise `Toolbar()` will return immediately.

If this argument is omitted, there is no function linked to the button. When the user clicks on the button, the `Toolbar()` function returns the button number.

Returns The call with no arguments returns the item number of the last button pressed. This call can be used to service multiple buttons with a single user-defined function. The call that creates a button returns 0.

**See also:**
```
Asc(), DlgButton(), Interact(), Toolbar(), ToolbarClear(), ToolbarEnable(),
ToolbarText(), ToolbarVisible()
```

# ToolbarText()

This replaces any message in the toolbar, and makes the toolbar visible if it is invisible. It can be used to give a progress report on the state of a script that takes a while to run.

```
Proc ToolbarText(msg$);
```

`msg$`    A string to be displayed in the message area of the toolbar.

**See also:**
```
Interact(), Toolbar(), ToolbarClear(), ToolbarEnable(), ToolbarSet(),
ToolbarVisible()
```

# ToolbarVisible()

This function reports on the visibility of the toolbar, and can also show and hide it. You cannot hide the toolbar if the `Toolbar()` function is in use.

```
Func ToolbarVisible({show%});
```

`show%`    If present and non-zero, the toolbar is made visible. If this is zero, and the `Toolbar()` function is not active, the toolbar is made invisible.

Returns The state of the toolbar at the time of the call. The state is returned as 2 if the toolbar is active, 1 if it is visible but inactive and 0 if it is invisible.

**See also:**
```
Interact(), Toolbar(), ToolbarClear(), ToolbarEnable(), ToolbarSet(), ToolbarText()
```

# Trim()

This function removes leading and trailing white space (spaces, tabs and end of line characters) or user-defined characters from a string variable. This function was added in Signal version 5.00.

```
Proc Trim(&text${, chars$};
```

`text$`    The string variable to remove characters from.

`chars$`  An optional list of characters to remove. If omitted, `" \t\n\r"` is used.

This function and the similar `TrimLeft()` and `TrimRight()` are commonly used to help parse user input that may contain multiple spaces. For example:

Input to `Trim()`    After `Trim()`    After `Trim(text$, "1234 ");`

`"  12AB34  "`    `"12AB34"`    `"AB"`

`"  1234  "`    `"1234"`    `""`

**See also:**
`DelStr$(), InStr(), Left$(), Len(), Mid$(), Right$(), TrimLeft(), TrimRight()`

# TrimLeft()

This function removes leading white space (spaces, tabs and end of line characters) or user-defined characters from a string variable. This function was added in Signal version 5.00.

**Proc TrimLeft(&text${, chars$};**

`text$`   The string variable to remove characters from.

`chars$` An optional list of characters to remove. If omitted, `" \t\n\r"` is used.

This function and the similar `Trim()` and `TrimRight()` are commonly used to help parse user input that may contain multiple spaces. For example:

Input to `TrimLeft()`  After `TrimLeft()`  After `TrimLeft(text$, "1234 ");`

`"  12AB34  "`    `"12AB34  "`    `"AB34  "`

`"  1234  "`    `"1234  "`    `""`

**See also:**
`DelStr$(), InStr(), Left$(), Len(), Mid$(), Right$(), Trim(), TrimRight()`

# TrimRight()

This function removes trailing white space (spaces, tabs and end of line characters) or user-defined characters from a string variable. This function was added in Signal version 5.00.

**Proc TrimRight(&text${, chars$};**

`text$`   The string variable to remove characters from.

`chars$` An optional list of characters to remove. If omitted, `" \t\n\r"` is used.

This function and the similar `TrimLeft()` and `Trim()` are commonly used to help parse user input that may contain multiple spaces. For example:

Input to `TrimRight()` After `TrimRight()`  After `TrimRight(text$, "1234 ");`

`"  12AB34  "`    `"  12AB34"`    `"  12AB"`

`"  1234  "`    `"  1234"`    `""`

**See also:**
`DelStr$(), InStr(), Left$(), Len(), Mid$(), Right$(), TrimLeft(), Trim()`

# Trunc()

Removes the fractional part of a real number or array. To truncate a real number to an integer, assign the real to the integer. `ArrConst()` copies a real array to an integer array.

**Func Trunc(x|x[]{[]...});**

`x`      A real number or a real array.

Returns  0 or a negative error code for an array. For a number it returns the value with the fractional part removed. `Trunc(4.7)` is `4.0`; `Trunc(-4.7)` is `-4.0`.

**See also:**
```
Abs(), ATan(), Ceil(), Cos(), Exp(), Floor(), Frac(), Ln(), Log(), Max(), Min(),
Pow(), Rand(), Round(), Sin(), Sqrt(), Tan(), Trunc()
```

# U

# U1401 commands

The `U1401` commands give you direct access to the CED 1401 interface connected to your computer. You should not use these commands during sampling, as this will probably cause the sampling to fail. You must use the `U1401Open()` command first to take control of the 1401, and you should use the `U1401Close()` command to release the 1401 once you have finished with it. See the 1401 family programming manual for details of using the 1401; this is available from CED as part of the 1401 programming kit and also as a download from the CED web site.

**See also:**
```
U1401Close(), U1401Ld(), U1401Open(), U1401Read(), U1401To1401(), U1401ToHost(),
U1401Write()
```

# U1401Close()

This command closes the link between the script language and a 1401 interface generated by the `U1401Open()` command. If there is no open 1401, the command is ignored.

```
Proc U1401Close();
```

**See also:**
```
U1401Ld(), U1401Open(), U1401Read(), U1401To1401(), U1401ToHost(), U1401Write()
```

# U1401Ld()

This command loads one or more 1401 commands with the option of nominating the folder to load the commands from. If no 1401 is open, the script halts.

```
Func U1401Ld(list${, path$});
```

list$   The list of commands to load separated by commas. Include `KILL` as the first item to clear all commands first. For example: `"KILL,ADCMEM,MEMDAC"`. The file name for a command is the command name plus an extension that depends on the type of 1401. The extension is added automatically.

path$   Optional. The path to the folder to search for the commands.  If omitted, or if the command is not found in this path, the `1401` folder in the Signal source folder is searched, then any path indicated by the `1401DIR` environment variable, and finally, the `\1401` folder on the current drive.

Returns  0 if all commands loaded. Otherwise the bottom 16 bits is a negative error code and the upper 16 bits is the 1-based index to the command in the list that failed to load. If `v%` is the non-zero return value, the command number is `v%/65536` and the error code is `(v% mod 65536 - 65536)`.

**See also:**
```
U1401Close(), U1401Open(), U1401Read(), U1401To1401(), U1401ToHost(), U1401Write()
```

# U1401Open()

This command attempts to open a 1401 for use by other `U1401` commands and returns the type of the opened 1401 or a negative error code. Note that the 1401 opened is the unit that will be (or is) used in sampling; if you attempt to use the 1401 from the script while sampling is in progress the likely result will be a failure of Signal. It is not an error to call `U1401Open()` multiple times with no intervening `U1401Close()`.

```
Func U1401Open({unit%});
```

unit%   Optional 1401 unit number in the range of 1 to 8 or 0 for the first available unit. If a 1401 unit number has been set using the Signal command line, the specified unit is used if `unit%` is omitted, otherwise a default value of zero is used.

Returns  The return value is the type of the 1401 detected: 0=standard 1401, 1=1401*plus*, 2=micro1401, 3=Power1401, 4=Micro1401 mk II, 5=Power1401 mk II, 6=Micro1401-3, 7=Power1401-3/3A, 8=Micro1401-4. Otherwise it is a negative error code that can be decoded by `Error$()`.

**See also:**
`Error$(), U1401Close(), U1401Ld(), U1401Read(), U1401To1401(), U1401ToHost(), U1401Write()`

# U1401Read()

This command reads a text response from a 1401 and optionally converts it into one or more integer values or reports the number of available input lines. If no 1401 is open, the script halts. There are four variants:

| | |
|---|---|
| **Func U1401Read();** | Get count of input lines |
| **Func U1401Read(&text$);** | Real an input line as text |
| **Func U1401Read(&v1%{, &v2, {...}});** | Read a line and convert to integers |
| **Func U1401Read(arr%[]);** | Read a line, convert to integer array |

text$   A text variable returned holding the entire response.

v1%     An integer variable that is returned with the first integer number read.

vn%     Optional integer variables (`v2%` up to `v12%`) returned with following values. Values for which no number is returned are unchanged.

arr%[]  An integer array that is filled (starting at element 0) with converted values.

Returns  The version with no arguments returns the number of available input lines. The other versions return the number of items that were converted from the input text and stored to a script variable or a negative error code. If you use a variant that reads a line and there is no text to read, the command times out after about 3 seconds and returns an error code.

**See also:**
`U1401Close(), U1401Ld(), U1401Open(), U1401To1401(), U1401ToHost(), U1401Write()`

# U1401To1401()

This command transfers the contents of an integer array to memory in the 1401.

**Func U1401To1401(const arr%[]{[]}, addr%{, size%});**

arr%    This is a one or 2 dimensional array to transfer. If you use a 2 dimensional array to interleave 4 channels of data, for example for MEMDAC, set the first dimension to 4 and the second to the number of points per channel.

addr%   The start address of the block of contiguous memory in the 1401 user area to be filled with data.

size%   Optional, the number of bytes in the 1401 that each array element is copied to. Acceptable values are 1, 2 or 4. If `size%` is omitted, 4 is used.

Returns  0 if the data transferred without a problem, or a negative error code.

**See also:**
`U1401Close(), U1401Ld(), U1401Open(), U1401Read(), U1401ToHost(), U1401Write()`

# U1401ToHost()

This command transfers a block of 1401 memory into an integer array.

```
Func U1401ToHost(arr%[]{[]}, addr%{, size%});
```

arr%     This is a one or 2 dimensional array to receive the data. If you use a 2 dimensional array to interleave 8 channels of data, for example for ADCMEM, set the first dimension to 8 and the second to the number of points per channel.

addr%    The start address of the block of contiguous memory in the 1401 user area to copy data from.

size%    Optional, taken as 4 if omitted. The number of bytes of 1401 data used to set each array element. Use 1, 2 or 4 to read 1, 2 or 4 bytes and sign extend to 32-bit integer. Use -1, -2 or -4 to read 1 or 2 bytes and zero extend to 32-bit integer.

Returns  0 if the data transferred without a problem, or a negative error code.

**See also:**
U1401Close(), U1401Ld(), U1401Open(), U1401Read(), U1401To1401(), U1401Write()

# U1401Write()

This command writes a text string to the 1401.

```
Func U1401Write(text$);
```

text$    The text to write to the 1401. Commands to the 1401 are terminated by either a newline "\n" or a semicolon ";".

Returns  0 if the line was added to the 1401 device driver output buffer, or a negative error code.

**See also:**
U1401Close(), U1401Ld(), U1401Open(), U1401Read(), U1401To1401(), U1401ToHost()

# UCase$()

This function converts a string into upper case. The upper-case operation may be system dependent. Some systems may provide localised uppercasing, others may only provide the minimum translation of the ASCII characters a-z to A-Z.

```
Func UCase$(text$);
```

text$    The string to convert.

Returns  An upper case version of the original string.

**See also:**
Asc(), Chr$(), DelStr$(), InStr(), LCase$(), Left$(), Len(), Mid$(), Print$(), ReadStr(), Right$(), Str$(), Val()

# V

# Val()

This converts a string to a number. The converter allows the same number format as the script compiler and leading white space is ignored.

```
Func Val(text${, &nCh%{, flag%}});
```

text$    A string that starts with a floating point number to convert. The conversion stops at the first character that is not part of the number. From version 4.06, the function will also accept a hexadecimal number if flag% is set to 1. In ambiguous cases, the conversion uses the format that uses the most characters of the input string, so "0xa" has the value 10, not 0 and uses all the characters. The string "0x" is converted as 0 and uses 1 character as "0x" is not a valid hexadecimal number. The expected formats are (items in curly brackets are optional, a vertical bar means use one of the characters before or after the bar):

```
{white space}{-|+}{digits}{.digits}{e|E{+|-}digits} or
```

```
            {white space}0x|Xhexadecimaldigits
```

nCh%    If present, it is set to the number of characters used to construct the number.

flag%   If present and set to 1, hexadecimal input is also acceptable.

Returns It returns the extracted number, or zero if no number was present. You should use the nCh% argument to decide if a number was processed.

**See also:**
Asc(), Chr$(), DelStr$(), InStr(), LCase$(), Left$(), Len(), Mid$(), Print$(), ReadStr(), Right$(), Str$(), UCase$()

# View()

The View() function sets the current view and returns the last view handle, or a negative error. A view handle is a positive integer > 0. Changing the current view does not change the focus or bring the view to the front; use FrontView() to do that.

```
Func View({vh%});
```

vh%     An integer argument being:

>0   A valid view handle that is to be made the current view. An invalid view handle will stop the script with a fatal error.

0    (Or omitted) no change of the current view is required.

<0   If the argument is –n, this selects the nth duplicate of the current data view. This is equivalent to Dup(n). Use ViewSource() to get the data view from which a memory or XY view is derived.

Returns 0 if there are no views at all, -1 if the duplicate requested does not exist, otherwise it returns the view handle of the view that was current.

## View(v,c).[ ]

The View(vh%,c).[] construction accesses view data for channel c. The [] refers to the whole array unless it encloses an expression to define a range of array elements. For waveform channels, the array holds the waveform values as expected. Marker channels appear as an array holding the marker times, but this array is read-only and a script error will be caused by attempting to assign to it. Use the MarkTime() function to change the times of markers. Idealised trace channels appear as a read-only array of the event start times.

```
View(vh%{, c%}).[{aExp}]
```

vh%     A view handle of an existing view, 0 for the current view, or -n for the n th duplicate view associated with the current view.

c%      A channel number from the view. If omitted, channel 1 is assumed.

aExp    An optional array indexing expression. If omitted, the whole array is accessed.

Here are three examples, to work on data from bin b% in channel c% of view v%:

```
val:=View(v%,c%).[b%]              'get one data value
sum:=ArrSum(View(v%,c%).[b%:100]); 'sum 100 data values
ArrDiff(View(v%,c%).[]);           'replace data by differences
```

## View().x()

The View().x() construction overrides the current view for the evaluation of the function that follows the dot. It is an error if the selected view does not exist, and the script stops. Don't use this contruct for functions which close the view.

```
View(vh%).x()
```

vh%     A view handle of an existing view, 0 for the current view (a waste of time), or -n for the n th duplicate view associated with the current view.

x()     A function or procedure.

For example, View(vh%).Draw() draws the view indicated by vh%. The equivalent code to View(vh%).x() is:

```
var temp%;
temp% := View(vh%);
x();
View(temp%);
```

This means that `View(vh%).FileClose();` will cause an error if `vh%` is the current view, so you should always use `View(vh%);FileClose();` instead.

**See also:**
`FrontView(), ViewFind(), ViewKind(), ViewSource(), Window(), WindowTitle$(),`
`BinToX(), XToBin()`

# ViewColour()

**Deprecated**. Use `ViewColourGet()` and `ViewColourSet()`. This function gets and sets the colours of data and XY view items, overriding the application-wide colours set by `Colour()`. Currently you can set the background colour.

**Func ViewColour(item%{, col%});**

`item%`   The colour item to get or set; 0=background

`col%`   If present, the new colour index for the item. There are 40 colours in the palette with indices 0 to 39. Use -1 to revert to the application colour for the item.

Returns   The palette colour index at the time of the call or -1 if no view colour is set.

**See also:**
`Colour dialog, ChanColour(), Colour(), PaletteGet(), PaletteSet(), XYColour()`

# ViewColourGet()

This function gets the RGB colour of items in the current data or XY view or in the current information window, indicating if it overrides the application-wide colours set by `ColourSet()`. This command was added at Signal version 5.02.

**Func ViewColourGet(item%{, &r, &g, &b});**

`item%`   The colour item to get. If the current view is a data or XY view the allowed values are 0 = view background, 1 = attached information window text, 2 = attached information window background, here the override information indicates if the application-level colours are overridden for the current data or XY view or for all of the information windows attached to the current data or XY view. If the current view is an information window the allowed values are 0 = information window text, 1 = information window background, here the override information indicates if the parent data or XY view colours are overridden for the current information window.

`r g b`   If present, set to the red, green and blue colour values in the range 0.0 to 1.0.

Returns   1 if the returned colour overrides the application or view colour, 0 if it does not.

**See also:**
`Colour dialog, ChanColourGet(),ColourGet(),ViewColourSet()`

# ViewColourSet()

This function sets the RGB colour of items in the current data or XY view or in the current information window, overriding the application-wide colours set by `ColourSet()`. This command was added at Signal version 5.02.

**Proc ViewColourSet(item%{, r, g, b});**

item%   The colour item to set; If the current view is a data or XY view the allowed values are 0 = view background, 1 = attached information window text, 2 = attached information window background. Changes made to the information window colours for a data or XY view affect all of the attached information windows. If the current view is an information window the allowed values are 0 = information window text, 1 = information window background, any changes made in this case only affect the current information window.

r g b   If present, These set the red, green and blue colour values in the range 0.0 to 1.0. If omitted, the item colour is set to the application default.

**See also:**
Colour dialog, ChanColourSet(),ColourSet(),ViewColourGet()

# ViewFind()

This function searches for a window with a given title and returns its view handle. The search looks for a view whose title starts with the supplied search string and is case-insensitive.

**Func ViewFind(title$);**

title$   A string holding the view title to search for. Note that system settings in Windows Explorer can cause the file extension to be removed from the view title, so you may have to search both with and without the ".cfs".

Returns  The view handle of a view with a title that matches the string, or 0 if no view matches the title.

**See also:**
FileOpen(), Window(), WindowTitle$(), View()

# ViewKind()

This function returns the type of a view or of the current view. Types 5-7 are reserved.

**Func ViewKind({vh%});**

vh%   The handle of the view for which the type is required. If omitted the function returns information about the current view.

Returns  The type of the view. View types are:

| | | |
|---|---|---|
| 0 | File view | A data view showing one frame at a time from a CFS data file. Access to any frame in the file is possible for analysis etc. |
| 1 | Text view | A view holding a text file for editing. |
| 2 | Output sequence | A view holding an output sequence file. |
| 3 | Script view | A view holding a script file. |
| 4 | Memory view | A data view created by a SetXXX() command, similar to a file view but wholly held in memory. |
| 5-7 | Reserved | Reserved for future expansion. |
| 8 | External text file | An invisible view for use with routines Read() and Print(). |
| 9 | External binary file | An invisible view for use by BRead(), BWrite() etc. |
| 10 | Application window | The Signal program window. |
| 11 | Other types | Other views with handles, such as the Status bar, and Toolbar, which can be made visible or invisible. |
| 12 | XY view | An XY view showing sets of XY data points. |
| 17 | Grid view | A grid view showing an array of cells with values |
| 18 | Information window | An information window showing high-visibility data |
| -1 | Unknown type | These include the cursor windows. |
| -2 | Invalid handle | The return value if the vh% parameter value is invalid. |

**See also:**
App(), ChanKind(), FileOpen(), View(), ViewList()

# ViewLineNumbers()

This function is used in a text-based view to display or hide line numbers and to set the number of decimal digits of space to use for a line number.

**Func ViewLineNumbers({show%});**

show%  Set -1 or omit for no change, 0 to hide line numbers, 1 to show them. Use 2-8 to set 2-8 digits of space and greater than 8 for a standard display (5 digits).

Returns  0 if line numbers were hidden at the time of the call, else the number of digits of display space allocated (2-8).

**See also:**
```
Gutter()
```

# ViewLink()

This function returns the view handle of the view that owns the current window. For example, you can use this to get the data view that created a memory or XY view or that owns a cursor window or an information window. This is slightly different from `View(-n)`, which finds the n th duplicate of the data view linked to the current data or XY view. This function was added in Signal version 5.00 and extended to add the second form in Signal 5.02.

```
Func ViewLink();
```

Returns  The handle of the linked view, or 0 if there is no such view.

This command can also be used to iterate through the process operations that are using the current data view as their source. For example, if you create a new sampling document using `FileNew()` with the current sampling configuration, you may also open other memory and XY documents and have additional channels present due to the sampling configuration. In this case, the command is:

```
Func ViewLink(n%{, mask%{, &name$}});
```

n%  This can be 0, meaning count the number of processes and return it, or it can be the process number to report on.

mask%  This value determines the types of processes we wish to count or report on and is the sum of: 1 for result views generated by `SetAverage()` and similar calls, 2 for XY views generated by `MeasureToXY()` and 4 for idealised trace data channels in the current view generated by `SetOpCl()` and similar. If this argument is omitted, it takes the value 3, to report on memory and XY views. Normally this argument will be set to 1, 2, 3 or 4. There is nothing to stop you using the values 5, 6 and 7, but you would need to interpret the returned `name$` argument to decide if the return value was a channel number or a view handle.

name$  If present, and n% is greater than 0, this is set to the name of the command that will process the data.

Returns  If n% is 0, this returns the number of processes that match the mask% argument. If n% is greater than 0, then this returns either the handle of the view or the channel number in the current view that is the target of the process operation. If n% is greater than 0 and there is no corresponding process, the return value is 0.

Note that this form of the command identifies active processes, that is processes for which the `Process()` command could have some effect.

**Example**
This example code lists the processes associated with the current view:

```
var vh%, n%,i%,name$;
n% := ViewLink(0, 3);    'count associated result and XY views
for i% := 1 to n% do
    vh% := ViewLink(i%, 3, name$);
    PrintLog("View Handle %d, name: %s\n", vh%, name$);
next;

n% := ViewLink(0, 4);    'count associated channels
for i% := 1 to n% do
    vh% := ViewLink(i%, 4, name$);
    PrintLog("Channel %d, name: %s\n", vh%, name$);
next;
```

**See also:**
```
App(), SampleHandle(), View(), ViewKind(), ViewList()
```

# ViewList()

This function fills an integer array with a list of view handles. Normally it does not return the view handle of the running script or files included by the running script. Use the special `types%` value for these, or the `App(3)` command to get the handle of the running script.

```
Func ViewList({list%[]{, types%}});
```

`list%`  An integer array that is returned holding view handles. The first element of the array, `list%[0]`, is set to the number of handles returned, and the remaining elements in the array are view handles. If the array is too small to hold the full number, the number that will fit are returned.

`types%`  The types of view to include. This is a code that can be used to filter the view handles. The filter is formed by adding the types from the list below. If this is omitted or if no types are specified for inclusion, all view handles are returned.

| | | | |
|---|---|---|---|
| 1 File views | 8 Script views | 128 Grid views | 1024 Application view |
| 2 Text views | 16 Memory | 256 External text views | 2048 Other view types |
| 4 Sequencer | 64 Information views | 512 External binary | 4096 XY views |

You can also exclude views otherwise included and vice-versa by adding:

| | |
|---|---|
| 8192 | Exclude views not directly related to the current view |
| 16384 | Exclude visible windows |
| 32768 | Exclude hidden windows |
| 65536 | Exclude duplicates |
| 131072 | Include the currently running script and any views holding include files included by the currently running script |

Returns  The total number of windows that satisfy the `types%`. This can be used to find a suitable array size.

The following example prints all the window titles into the log view:

```
var list%[100],i%;        'Assume no more than 99 views
ViewList(list%[]);        'Get a list of all the views
for i%:=1 to list%[0] do  'Iterate round all the views
   PrintLog(view(list%[i%]).WindowTitle$()+"\n");
   next;
```

**See also:**
App(), SampleHandle(), ViewKind()

# ViewMaxLines()

This function gets and optionally sets the maximum number of lines that the current text view will retain if text is added by a script. For the Log view, this value is initially set by the Edit menu Preferences in the Script page of the Preferences dialog, but a script can set or clear the line limit for any text-based view. `ViewStandard()` clears the limit except in a Log view, where it sets the value set in the preferences.

```
Func ViewMaxLines({max%});
```

`max%`  If present, this sets the maximum lines to keep after a script `Print()` or `PrintLog()` statement. If this is omitted or negative, no change is made. Set 0 for no limit and greater than zero to set a limit.

Returns  The value of the limit at the time of the call.

Each time the line limit is exceeded, the number of lines in the file is reduced to 90% of the maximum line count by deleting the lowest numbered lines. We reduce the line count to 90% of the maximum because deleting lines

from the start of the view is a slow operation (in very large views); by allowing the view to grow normally for a while this minimises the time penalty for setting this option. For example, if `max%` is set to 1000, the next time that a line is added and more than 1000 lines are in the file, the line count is reduced to 900. After this 100 lines can be added before the limit is tripped.

**See also:**
Edit Preferences, ViewStandard()

# ViewSource()

This function returns the handle of the source view for a memory or XY view created by processing.

```
Func ViewSource();
```

Returns  The handle of the data view from which this memory or XY view was derived, and from which it obtains its data. Note that memory views created using `SetCopy()` or `SetMemory()` do not have a data view attached and return 0.

**See also:**
View(), SetXXX(), SetAverage(), SetCopy(), SetMemory(), SetPower()

# ViewStandard()

This sets the current data, XY, grid or text 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 Display page of the Preferences dialog. In an XY view the key is hidden and the axes are optimised.

In a text-based view it removes any maximum line limit except in the Log view, where is applies the line limit set in the Display page of the Preferences dialog. It also hides line numbers, displays the gutter and the folding margin (for views that support folding). All the text styles are set to the default values (equivalent to opening the Font dialog and using Reset All) and any zooming is removed.

```
Proc ViewStandard();
```

**See also:**
ChanOrder(), ChanWeight(), DrawMode(), XYDrawMode()

# ViewUseColour()

This function can be used to force the display to use black and white only, or to use the colours set in the colour dialog or by the `Colour()` command.

```
Func ViewUseColour({use%});
```

use%  If present, a value of 0 forces Signal to display all windows in black and white. Any other value allows the use of colour. If omitted, no change is made.

Returns  The current state as 1 if colour is in use, 0 if black and white is used.

**See also:**
Colour()

# ViewZoom()

This function is used in a text-based view to get and optionally set the zoom factor. This is the number of points to add to the nominal size of the text.

```
Func ViewZoom({zoom%});
```

zoom%  Omit for no change or a value in the range -10 to 20 to add to the point size of all the text. The resulting minimum text size is 2 point regardless of `zoom%`.

Returns  The zoom value at the time of the call.

**See also:**

```
ViewStandard(), Zooming
```

# VirtualChan

This command controls virtual channels in the current data view. Virtual channels are defined by an algebraic expression that can include other channels (but not virtual channels). This command has the functionality of the Virtual Channel dialog. The first command variant creates and modifies a virtual channel, the second reports the state of a virtual channel.

```
Func VirtualChan(chan%, expr${, match%{, binsz{, align}}});
Func VirtualChan(chan%, get%{, &expr$});
```

chan%   In the first command variant, set this to 0 to create a new virtual channel and return the channel number. The remaining arguments set the initial channel settings, otherwise default values are used. If not 0, this is the number of an existing virtual channel to modify or from which to read back the settings.

expr$   In the first command variant, this is a string expression defining the output. See the Analysis chapter of the Signal manual for details. In the second command variant, this is an optional string variable that is returned holding the current expression for the channel.

match% This is the number of an existing waveform channel to match for sample interval and alignment. If 0, the sample interval and alignment are set by the `binsz` and `align` arguments. If negative, no change is made.

binsz   If `match%` is 0, this sets the sample interval of the channel. Values of `binsz` less than or equal to zero are ignored.

align   If `match%` is 0, this sets the channel alignment. The first point of the channel will be at time align mod binsz. Values less than 0 are ignored.

get%   In the second command variant, `get%` determines the returned value. 0 = the state of the expression parsing, 1 = `match%`, 2 = `binsz`, 3 = `align`.

Returns   When creating a new channel, the return value is the new channel number or a negative error code. When modifying an existing channel, the return value is a negative code if the `match%`, `binsz` or `align` arguments are illegal, a positive code if the expression contains an error and 0 if there was no error. The second variant returns the requested information or a negative error code. If `get%` is 0, the return value is negative if the channel is not a virtual channel, 0 if the expression is acceptable and a positive code if not.

**See also:**
More about virtual channels

# W

# Window()

This sets the position and size of the current view. Normally, positions are percentages from the top left-hand corner of the application window size. You can also set positions relative to a monitor. This can also be used to position, dock and float dockable toolbars.

```
Func Window(xLow, yLow{, xHigh{, yHigh{, scr%{, rel%}}}});
```

xLow   Position of the left hand edge of the window in percent. When docking a dockable toolbar, the `xLow` and `xHigh` values correspond to the position of the top left corner of the window when dropped with the mouse.

yLow   Position of the top edge of the window in percent.

xHigh   If present, the right hand edge. If omitted the previous width is maintained. If the window is made too small, the minimum allowed size is used. If the current view is a dockable control and `yHigh` is 0, values less than 1 or greater than 4 float the window at (`xLow`, `yLow`), otherwise `xHigh` sets the docking state:

| | | | |
|---|---|---|---|
| 1 | Docked to the left window edge | 3 | Docked to the right edge |
| 2 | Docked to the top window edge | 4 | Docked to the bottom edge |

yHigh   If present, the bottom edge position. If omitted the previous height is maintained. If the window is made too small, the minimum allowed size is used.

If the Window is dockable and `yHigh` is 0, this command sets the docked state of the window (see `xHigh`). Otherwise the window is floated with the nearest allowed width that is no more than `xHigh-xLow`. If `xHigh-xLow` is 0 or negative `YHigh` sets the height of the dockable window.

scr%    Optional screen selector for views, dialogs and the application window. If omitted or -1, positions are relative to the application window. Otherwise, 0 selects the entire desktop rectangle and greater values select a particular screen rectangle (but see `rel%`). See `System()` for more screen information.

rel%    Ignored unless `scr%` is greater than 0. Set 0 or omit for positions relative to the intersection of the selected monitor and the application window, 1 for positions relative to the monitor. If there is no intersection, there is no position change. When positioning the application window, `rel%` is treated as if it were 1.

Returns  1 if the position was returned, or -1 if the rectangle set by scr% and rel% is of zero size.

Examples:

```
view(App()).Window(0,0,100,100,0);  'use all desktop
view(App()).Window(0,0,100,100,2);  'use all second monitor
```

**See also:**
```
App(), System(), WindowDuplicate(), WindowGetPos(), WindowSize(), WindowTitle$(),
WindowVisible()
```

# WindowDuplicate()

This duplicates the current data view, creating a new window that has all the settings of the current view. It does not duplicate channels; these are shared with the existing window. The new window becomes the current view and is created invisibly.

**Func WindowDuplicate();**

Returns  This command returns the view handle of the new window or a negative error code or 0 if no free duplicates (there is a limit of 9 duplicates per data view.)

**See also:**
```
Dup(), Window(), WindowGetPos(), WindowSize(), WindowTitle$(), WindowVisible(),
View()
```

# WindowGetPos()

This gets the window position of the current view with respect to the application window. Positions are measured from the top left-hand corner as a percentage.

**Func WindowGetPos(&xLow, &yLow, &xHigh, &yHigh{, scr%{, rel%}});**

xLow    A real variable that is set to the position of the left hand edge of the window.

yLow    A real variable that is set to the position of the top edge of the window.

xHigh   A real variable that is set to the position of the right hand edge of the window or that returns a docking code for a docked control bar if `yHigh` is returned as 0.

1   Docked to the left window edge     3   Docked to the right edge
2   Docked to the top window edge      4   Docked to the bottom edge

yHigh   A real variable that is set to the position of the bottom edge of the window or to 0 if the window is docked.

scr%    Optional screen selector for views, dialogs and the application window. If omitted or -1, positions are relative to the application window. Otherwise, 0 selects the entire desktop rectangle and greater values select a particular monitor (but see `rel%`). See `System()` for more screen information..

rel%    Ignored unless `scr%` is 0 or greater. Set 0 or omit for positions relative to the intersection of the rectangle set by `scr%` and the application window, 1 for positions relative to the `scr%` rectangle.

Returns  1 if the position was returned, or -1 if the rectangle set by scr% and rel% is of zero size.

**See also:**
```
Window(), WindowDuplicate(), WindowSize(), WindowTitle$(), WindowVisible(), System(),
DlgGetPos()
```

# WindowSize()

This procedure is used to resize the current window without changing the position of the top left-hand corner. Setting a window dimension less than zero leaves the dimension unchanged. Setting a dimension smaller than the minimum allowed sets the minimum value. Setting a size greater than the maximum allowed sets the maximum size. There are no errors from this function. When this is used to size Signal application window the available area is the whole screen, otherwise the available area is Signal application area.

```
Proc WindowSize(width, height);
```

width   The width of the window as a percentage of the available area.

height  The height of the window as a percentage of the available area.

**See also:**
```
App(), Window(), WindowDuplicate(), WindowGetPos(), WindowTitle$(), WindowVisible()
```

# WindowTitle$()

This function gets and sets the title of the current window. There may be windows that are resistant to having their title changed, for these, the routine has no effect. Most windows can return a title. If you change a title, dependent window titles change, for example, cursor windows belonging to data views track the title of the data view. Data files use decorated titles where the basic title has extra information (normally the frame number) appended to it. The title that is returned using this function will incorporate this extra text, but when the command is used to set a new window title the new text only sets the basic title to which any decoration is applied.

```
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. An iconised window can be sent data, sized and so on; the result is not visible. This will dock a dockable window at its last docked position.

3   Maximise, make it as large as possible or float a dockable window.

4   Application window only; extend over all available desktop monitors.

Returns  The window state prior to this call.

**See also:**

```
FrontView(), Window(), WindowDuplicate(), WindowGetPos(), WindowSize(),
WindowTitle$()
```

# X

## XAxis()

This turns on and off the x axis of the current view and returns the state of the x axis.

`Func XAxis({on%});`

on%    Set the axis state. If omitted, no change is made. 0=Hide axis, 1=Show axis.

Returns  The axis state at the time of the call (0 or 1, as above) or a negative error code. It is an error to use this function on a view that has no concept of an x axis.

Changes made by this function do not cause a redraw immediately. The affected view is drawn at the next opportunity.

**See also:**
`XAxisMode(), XAxisStyle(), XHigh(), XLow(), XRange()`

## XAxisAttrib()

This function controls the choice of logarithmic or linear axis and automatic adjustment of axis units at high zoom levels. This command is equivalent to the check boxes at the bottom of the X Axis dialog.

`Func XAxisAttrib({flags%});`

flags% A value that controls how the axis looks and behaves, zero sets a linear axis with no auto-adjust of units for high zoom. Add the following values to change the behaviour:

1    Display a logarithmic axis.
2    Display powers on a logarithmic axis (you must have added 1 as well for this to take effect).
4    Cause a linear axis to auto-adjust its units by factors of 1000 at high zoom around 0.
16   When used along with 4, causes the axis to auto-adjust its units by using an SI prefix rather than factors of 1000.

Omit this argument for no change to the attributes.

Returns  The sum of the current flags set for the x-axis.

**See also:**
`XAxis(), XAxisMode(), XAxisStyle(), XHigh(), XLow(), XRange(), YAxisAttrib()`

## XAxisMode()

This function controls what is drawn in an x axis.

`Func XAxisMode({mode%});`

mode%   Optional argument that controls how the axis is displayed. If omitted, no change is made. Possible values are the sum of the following:

1    Hide all the title information.
2    Hide all the unit information.
4    Hide x axis small ticks. These are also hidden when big ticks are hidden.
8    Hide x axis numbers. These are also hidden when big ticks are hidden.
16   Hide the big ticks and the horizontal line that joins them. This will also hide the small ticks and numbers as there is nowhere to put them.
32   Scale bar axis. If selected add 4 to remove the end caps.

Returns  The x axis mode value at the time of the call or a negative error code.

**See also:**

```
XAxis(), XHigh(), XLow(), XRange(), YAxisMode()
```

# XAxisStyle()

This function controls the major and minor tick spacing for all views that have an x axis. If you set values that would cause illegible or unintelligible axes, they are stored but not used unless the axis range or scaling changes to make the values useful.

```
Func XAxisStyle({style%{, nTick%{, major}}});
```

style%  A value of -1 returns the number of minor divisions set or 0 for automatic. A value of -2 returns the major tick spacing or 0 for automatic spacing. Set `style%` to 0 if setting `nTick%` or `major`.

nTick%  The number of minor tick subdivisions or 0 for automatic spacing. Omit `nTick%` or set it to -1 for no change.

major  If present, values greater than 0 set the major tick spacing. Values less than or equal to 0 select automatic major tick spacing.

Returns  See the description of `style%` for return values.

**See also:**
```
XAxis(), XAxisAttrib(), XAxisMode(), XHigh(), XLow(), XRange(), YAxisStyle()
```

# XHigh()

This function  returns the value of the right end of the X axis of the current data or XY view or the line past the last fully visible line in a text view. To find the frame limit use `Maxtime()`.

```
Func XHigh();
```

Returns  In a data or XY view, the result is the X axis upper limit in X axis units. It is in lines in a text view.

In a text view, the value is the first visible line number plus the number of fully visible lines. At the end of a file, the returned value can be greater than the number of lines in the file.

**See also:**
```
Draw(), XRange(), BinToX(), XToBin(), XLow(), Maxtime()
```

# XLow()

This function returns the value that corresponds to the left end of the X axis of the current data or XY view or the first visible line in a text view. To find the frame limit use `Mintime()`.

```
Func XLow();
```

Returns  The X axis lower limit in X axis units. In a text view, this returns a line number (the first line number in a text view is 1).

**See also:**
```
Draw(), XRange(), BinToX(), XToBin(), XHigh(), Mintime(), MoveBy()
```

# XRange()

This function sets the start and end of the x axis in a data or XY view in x axis units. In a grid view, it sets the first column to display. Unlike `Draw()`, it does not update the view immediately; updates must wait for the next `Draw()`, `DrawAll()`, `Yield()` or some interactive activity.

```
Proc XRange(low{, high});
```

low  The left hand edge of the view in x axis units or the 0-based first column for a grid view.

high  The right hand edge of the view, not used for grid views. If omitted, the view stays the same width.

Values are limited to the axis range. Without `high`, it preserves the width, adjusting `low` if required. If the resulting width is less than the minimum allowed, no change is made.

**See also:**
```
Draw(), XLow(), XHigh(), Yield()
```

# 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 function gets the title of the x axis. In a data or XY view you can also set the title. The window will update with a new title at the next opportunity, but note that the x axis title is not written to CFS data files.

**`Func XTitle$({new$});`**

new$    If present this sets the new x axis title in a data or XY view.

Returns  The x axis title at the time of the call.

**See also:**
```
ChanTitle$(), XUnits$()
```

# XToBin()

This function converts between x axis values and bin numbers for a channel in the current view. For waveform channels a bin is a waveform point, for marker-type channels it is a marker item and for idealised trace channels it is a trace segment.

**`Func XToBin(chan%, x);`**

chan%   A channel number (1 to n).

x       An x axis value. If it exceeds the x axis range it is limited to the nearer end.

Returns  In a data view it returns the bin number that corresponds to x, bin numbers start at zero with the first data point. For waveform channels this will generally 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. For marker-type channels the result will be the index of the marker item at or before the specified x position. For idealised trace channels the result will be the index of the trace segment within which the x position lies, or the index of the segment lying before that x position.

**See also:**
```
BinToX(), BinSize(), BinZero()
```

# XUnits$()

This function gets the units of the x axis in the current view, note that for a data document with a time X axis the units returned will be **"s"** regardless of how time is displayed. In a data or XY view, you can also set the units. The window will update with the new units at the next opportunity and they will become part of the new file if it is saved.

**`Func XUnits$({new$});`**

new$    If present this sets the new x axis units in a new file or memory view.

Returns  The x axis units at the time of the call.

**See also:**
```
ChanUnits$(), XTitle$()
```

# XY view commands

## XYAddData()

This adds data points to an XY view channel. If the axes are set to automatic expanding mode by `XYDrawMode()`, they will change when you add a new data point that is out of the current axis range. If the channel is set to a fixed size (see `XYSize()`), adding new points causes older points to be deleted once the channel is full. The first form of the command allows unrestricted x and y positions. The second form is for data that is equally spaced in the x direction.

```
Func XYAddData(chan%, x|const x[]|x%[], y|const y[]|y%[]);
Func XYAddData(chan%, const y[], xInc{, xOff});
```

chan%   A channel number in the current XY view. The first channel is number 1.

x       The x co-ordinate(s) of the added data point(s). In the first form of the command, both x and y must be either single variables or arrays. If they are arrays, the number of data points added is equal to the size of the smaller array.

y       The y co-ordinate(s) of the added data point(s). In the second form of the command, this is an array of equally spaced data in x.

xInc    Sets the x spacing between the y data points in the second form of the command.

xOff    Sets the x position of the first data point in the second form of the command. If omitted, the first position is set to 0.

Returns  The number of data points which have been added successfully.

**See also:**
XYColour(), XYCount(), XYDelete(), XYDrawMode(), XYGetData(), XYInCircle(),
XYInRect(), XYJoin(), XYKey(), XYRange(), XYSetChan(), XYSize(), XYSort()

## XYColour()

**Deprecated**, use ChanColourSet() and ChanColourGet() instead in new scripts. This function gets or sets a channel line colour index or a channel fill colour index in the current XY view.

```
Func XYColour(chan%{, col%{, item%}});
```

chan%   A channel number in the current XY view. The first channel is number 1.

col%    The index of the colour in the colour palette. There are 40 colours in the palette, numbered from 0 to 39. If omitted or -1, there is no colour change.

item%   Set 1 for the line colour, 2 for the fill colour. Taken as 1 if omitted.

Returns  The colour index in the colour palette at time of call or a negative error code.

**See also:**
Colour dialog, ChanColourSet(), ChanColourGet(), Colour(), XYAddData(), XYCount(),
XYDelete(), XYDrawMode(), XYGetData(), XYInCircle(), XYInRect(), XYJoin(), XYKey(),
XYRange(), XYSetChan(), XYSize(), XYSort()

## XYCount()

This gets the number of data points in a channel in the current XY view. To find the maximum number of data points, see the `XYSize()` command.

```
Func XYCount(chan%);
```

chan%   A channel number in the current XY view. The first channel is number 1.

Returns  The number of data points in the channel or a negative error code.

**See also:**

XYAddData(), XYColour(), XYDelete(), XYJoin(), XYDrawMode(), XYGetData(),
XYInCircle(), XYInRect(), XYKey(), XYRange(), XYSetChan(), XYSize(), XYSort()

# XYDelete()

This command deletes a range of data points or all data points from one channel of the current XY view. Use
ChanDelete() to delete the entire channel.

```
Func XYDelete(chan%{, first%{, last%}});
```

chan%    A channel number in the current XY view. 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(), XYColour(), XYCount(), XYDrawMode(), XYGetData(),
XYInCircle(), XYInRect(), XYJoin(), XYKey(), XYRange(), XYSetChan(), XYSize(),
XYSort()

# 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) | 3 | crosses x | 6 | diamonds |
| 1 | boxes | 4 | circles (not 9x) | 7 | horizontal line |
| 2 | plus signs + | 5 | triangles | 8 | vertical line |

2    Get or set the size of the data points in points (units of approximately 0.353 mm). The sizes allowed
     are 0 to 100; 0 is invisible. The default size is 5.

3    Get or set the line style. If the line thickness is greater than 1 all lines are drawn as style 0. Styles are:

| | | | | |
|---|---|---|---|---|
| 0 | solid (default) | 1 | dotted | 2 | dashed |

4    Get or set the line thickness in points. Thickness values range from 0 (invisible) to 10. The default is
     1.

5    Get or set automatic axis range mode. This applies to the entire view, so the chan% argument is
     ignored. Values are:

0 The axes do not change automatically when new data points are added.
1 When new data points are added that lie outside the current x or y axis range, the data and axes
  screen area update at the next opportunity to display all the data.

new%     New draw mode or axis expanding mode. If omitted, no change is made.

Returns  The value of the relevant channel draw mode or axis expanding mode at the time of the call or a negative
         error code.

**See also:**
XYAddData(), XYColour(), XYCount(), XYDelete(), XYGetData(), XYInCircle(),
XYInRect(), XYJoin(), XYKey(), XYRange(), 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, 0 is used.

last%    The zero-based index of the last data point returned, used when x and y are arrays. If omitted or greater than or equal to the number of data points, the final data point is the last one in the channel. If last% is less than first%, no data points are returned.

Returns  The number of data points copied. If the x or y arrays are not big enough to hold all the data points from first% to last%, the return value is the array size. If x or y are not arrays, if a data point with index first% exists, 1 is returned.

The index number of a data point depends on the current sorting method (see XYSort()).

**See also:**
XYAddData(), XYColour(), XYCount(), XYDelete(), XYDrawMode(), XYInCircle(),
XYInRect(), XYJoin(), XYRange(), XYKey(), XYSetChan(), XYSize(), XYSort()

# XYInChan()

This function returns the number of data points in a channel of the current XY view that lie inside another channel that is treated as a joined up shape. The points in the channel that defines the shape are not sorted (regardless of the sorting mode set for that channel); they are always considered in the order in which the points were added to the channel. You might use this function to let a user draw a shape (using ToolbarMouse()) in an XY view around data points that they want to select for a particular purpose.

For a data point to lie inside the channel, we count the number of times that a line drawn from the data point to infinity in any direction crosses a line of the channel. If it crosses an even number of times it is outside (0 is even). If it crosses an odd number of times, it is inside. This is obvious for simple shapes, but less so for complex ones. In the example picture to the right, the shaded sections are the inside and the non-shaded are the outside. Points that lie exactly on the boundary may be inside or outside; however, if you have a set of shapes that exactly tessellate to fill an area, any point placed in that area
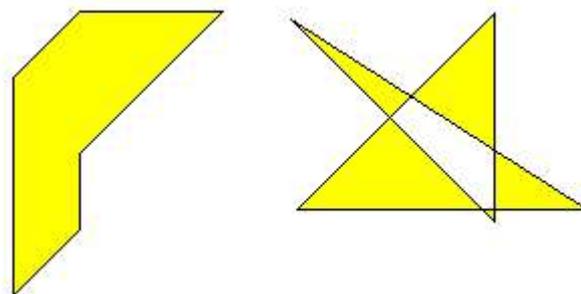


*Inside and outside*

will be in only one of the shapes. If there are n points to check, and the channel that defines the shape has m points, then this algorithm takes a worst case time of O(n*m). This means that it pays to search inside shapes with relatively few points to define them. The algorithm will be relatively quick if you are searching for points inside an area that is small compared to the area that the data points cover.

```
Func XYInChan(chan%, shCh%{, list%[]});
```

chan%    A channel number in the current XY view that defines the points to test against the shape.

shCh%    The channel in the current XY view that defines the shape that the points must be inside.

list%    An optional integer array that is returned holding the indices of the data points that were inside. If the array is too small, the function returns the number that would have been returned had the array been large enough. The index values returned are the index positions in the current sort mode for the channel. You can read the points back with XYGetData().

Returns  The number of data points inside the rectangle or a negative error code.

**See also:**
```
XYAddData(), XYColour(), XYInCircle(), XYInRect(), XYDelete(), XYDrawMode(),
XYGetData(), XYJoin(), XYKey(), XYRange(), XYSetChan(), XYSize(), XYSort()
```

# XYInCircle()

This gets the number of data points inside a circle defined by `xc`, `yc`, and `r` in the current XY view. A general point $(x, y)$ is considered to be inside the circle if:

$$(x-xc)^2 + (y-yc)^2 <= r^2$$

Points lying on the circumference are considered inside, but owing to floating-point rounding effects they may be indeterminate.

```
Func XYInCircle(chan%, xc, yc, r{, list%[]});
```

chan%  A channel number in the current XY view. The first channel is number 1.

xc,yc  These are the x and y co-ordinates of the centre of the circle.

r      This is the radius of the circle. `r` must be $>= 0$.

list%  This is an optional integer array that is filled in with the indices of the points in the channel that were inside the circle. Points are filled from index 0 of the array until the array is full or there are no more points. The index values returned are the index positions in the current sort mode for the channel. You can read the points back with XYGetData().

Returns  The number of data points inside the circle or a negative error code.

**See also:**
```
XYAddData(), XYColour(), XYInChan(), XYCount(), XYDrawMode(), XYGetData(),
XYInRect(), XYJoin(), XYRange(), XYKey(), XYSetChan(), XYSize(), XYSort()
```

# XYInRect()

This function returns the number of data points in a channel of the current XY view that lie inside a rectangle. To be inside, a data point must lie between the low rectangle coordinate up to, but not including the high coordinate. This is so that two rectangles with a common edge will not both count a data point on the boundary.

```
Func XYInRect(chan%, xl, yl, xh, yh{, list%[]});
```

chan%  A channel number in the current XY view.

xl,xh  The `x` co-ordinates of the left and right hand edges of the rectangle. `xh` must be greater than or equal to `xl`.

yl,yh  The `y` co-ordinates of the bottom and top edges of the rectangle. `yh` must be greater than or equal to `yl`.

list%  This is an optional integer array that is filled in with the indices of the points in the channel that were inside the rectangle. Points are filled from index 0 of the array until the array is full or there are no more points. The index values returned are the index positions in the current sort mode for the channel. You can read the points back with `XYGetData()`.

Returns  The number of data points inside the rectangle or a negative error code.

**See also:**
```
XYAddData(), XYColour(), XYInChan(), XYInCircle(), XYDelete(), XYDrawMode(),
XYGetData(), XYJoin(), XYKey(), 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), filled or filled and framed.

```
Func XYJoin(chan%{, join%});
```

chan%   A channel number in the current XY view. The first channel is number 1. -1 is also allowed, meaning all channels.

join%   If present, 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.
3   Not joined by lines, but the channel data is filled with the channel fill colour.
4   Joined by lines and filled with the channel fill colour set by `XYColour()`.
5   Draw as a histogram with a baseline at zero, filled with the channel fill colour.

Returns  The joining method at the time of the call or a negative error code.

### See also:
XYAddData(), XYColour(), XYCount(), XYDelete(), XYDrawMode(), XYGetData(),
XYInCircle(), XYInRect(), XYRange(), XYKey(), XYSetChan(), XYSize(), 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:
View menu Options command, ChanTitle$(), XYAddData(), XYColour(), XYCount(), XYDelete(),
XYDrawMode(), XYGetData(), XYInCircle(), XYInRect(), XYJoin(), XYRange(),
XYSetChan(), XYSize(), XYSort()

# XYOffset()

This command sets and gets an XY channel offset. The offset moves the origin of the x,y co-ordinate system when the channel is drawn. This allows you to generate a grid of histograms or to generate waterfall displays.

```
Proc XYOffset(chan%, x, y{, opt%});
Proc XYOffset(chan%, &x, &y{, opt%});
```

chan%   A channel in the current XY view.

x,y     Used when setting the channel offset. This is the offset in the current *axis units*.

&x,&y   Used when returning the offset. Returns the offset in the current *axis units*.

opt%    Omit or set to 0 to set the offset. Set -1 to return the offset.

### Axis units

In Signal you have the choice of drawing axes in linear, logarithmic or square root mode. In linear mode, the axis units are the same as the user units. In logarithmic mode, the axis units are `Log(user units)`, in square root mode they are `root`(user units).

This command is somewhat experimental (at version 5.06) and more `opt%` values may be added in the future.

**See also:**
`XYRange()`

# XYRange()

This function gets the range of data values of a channel or channels in the current XY view. This is equivalent to the smallest rectangle that encloses the points.

```
Func XYRange(chan%, &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(), XYColour(), XYCount(), XYDelete(), XYDrawMode(), XYGetData(),`
`XYInCircle(), XYInRect(), XYKey(), XYSetChan(), XYSize(), XYSort()`

# XYSetChan()

This function creates a new channel or modifies an existing channel in the current XY view. It is an error to use this function if the current view is not an XY view. This function can modify all properties of an existing channel without calling the `XYSize()`, `XYSort()`, `XYJoin()` and `XYColour()` commands individually.

```
Func XYSetChan(chan%{, size%{, sort%{, join%{, col%}}}});
```

`chan%`    A channel number in the current XY view. If `chan%` is 0, a new channel is created. Each XY view can have maximum of 256 channels, numbered 1 to 256. The first channel is created automatically when you open a new XY view with `FileNew()`. If `chan%` is not 0, it must be the channel number of an existing channel to modify or -1 to modify all channels.

`size%`    This sets the number of data points in the channel and how and if the number of data points can extend. The only limits on the number of data points are the available memory and the time taken to draw the view.

       A value of zero (the default) sets no limit on the number of points and the size of the channel expands as required to hold data added to it.

       If a negative size is given, for example `-n`, this limits the number of points in the channel to `n`. If more than `n` points are added, the oldest points are deleted to make space for the newer points. If you set a negative size for an existing channel that is smaller than the points in the channel, points are deleted.

       If a positive value is set, for example `n`, this allocates storage space for `n` data points, but the storage will grow as required to accommodate further points. Using a positive number rather than 0 can save time if you know in advance the likely number of data points as it costs time to grow the data storage.

`sort%`    This sets the sorting method of data points. The sorting method is only important if the points are joined. If they are not joined, it is much more efficient to leave them unsorted as sorting a large list of points takes time. The sort methods are:

    0    Unsorted (default). Data is drawn and sorted in the order that it was added. The most recently added point has the highest sort index.

1 Sorted by x value. The index runs from points with the most negative x value to points with the most positive x value.

2 Sorted by y value. The index runs from points with the most negative y value to points with the most positive y value.

If this is omitted, the default value 0 is used for a new channel. For an existing channel, there is no change in sorting method.

join% If present, this is the new joining method of the channel. If this is omitted, no change is made to an existing channel; a new channel is given mode 0. The data point joining methods are:

0 Not joined by lines (this is the default joining method).

1 Joined by lines. The line styles are set by XYDrawMode().

2 Joined by lines and also connect the first and last data points to form a loop.

3 Not joined by lines, but the channel data is filled with the channel fill colour.

4 Joined by lines and filled with the channel fill colour set by XYColour().

col% If present, this sets the index of the colour in the colour palette to use for this channel. There are 40 colours with indices 0 to 39. If omitted, the colour of an existing channel is not changed. The default colour for a new channel is the colour that a user has chosen for an XY view channel.

Returns The highest channel number that was affected or a negative error code. When you create a channel, the value returned is the new channel number.

**See also:**
XYAddData(), ChanColourSet(), XYCount(), XYDelete(), XYDrawMode(), XYGetData(), XYInCircle(), XYInRect(), XYJoin(), XYKey(), XYRange(), XYSize(), XYSort()

# XYSetData()

This command is used to modify the data in an XY channel. Use XYAddData() to add new data points. This command was added at Signal version 7.05.

```
Func XYSetData(chan%, x, y, index%);
```

chan% A channel number in the current XY view. The first channel is number 1.

x The new x co-ordinate of the data point.

y The new y co-ordinate of the data point.

index% The index (in the current sort mode) of the data point in the channel to be modified, the first point in the channel is at index zero. If the data is sorted, the modified point will not (in general) have the same index as the original. This can cause problems if you wish to modify multiple points as this may change the indices of all the other points.

Returns 1 if the point was modified, 0 if the index does not exist and a negative error code if the channel does not exist.

**See also:**
XYAddData(), XYCount(), XYDelete(), XYGetData(), XYInCircle(), XYInRect(), XYJoin(), 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 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()

# Y

# 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()

# YAxisAttrib()

This function controls the choice of logarithmic or linear axis and automatic adjustment of axis units at high zoom levels. This command is equivalent to the check boxes at the bottom of the Y Axis dialog.

```
Func YAxisAttrib(cSpc{, flags%});
```

cSpc    A channel specifier or -1 for all, -2 for visible or -3 for selected channels. When multiple channels are specified, returned values are for the first channel.

flags%  A value that controls how the axis looks and behaves, zero sets a linear axis with no auto-adjust of units for high zoom. Add the following values to change the behaviour:

　　1　Display a logarithmic axis.
　　2　Display powers on a logarithmic or square root axis (you must have added 1 or 8 as well for this to take effect).
　　4　Cause a linear or square root axis to auto-adjust its units by factors of 1000 at high zoom around 0.
　　8　Display a square-root axis.
　　16　When used along with 4, causes the axis to auto-adjust its units by using an SI prefix rather than factors of 1000.

　　It is an error to add both 1 and 8 for logarithmic and square root mode at the same time, 2 without 1 or 8 and 16 without 4 do nothing. Omit this argument for no change to the attributes.

Returns  The sum of the current flags set for the y-axis of the first channel in the list.

**See also:**
```
YAxis(), YAxisMode(), YAxisStyle(), YHigh(), YLow(), YRange(), XAxisAttrib()
```

# YAxisLock()

This function locks and unlocks the axes of grouped channels and reports on the locked state of grouped channels. If you lock a group, the grouped channels keep their own axis ranges, but display using the axis of the first channel in the group. The `YRange()`, `YHigh()` and `YLow()` commands operate on the information stored with a channel. To change the displayed range of grouped and locked channels, you must use `YRange()` on the first channel in a group.

```
Func YAxisLock(chan%{, opt%{, vOffs}});
```

chan%   A channel that is in the group that you wish to address.

opt%    If present, values of 1 and 0 set and unset the locked state. A value of -1 returns the visual offset per channel for the group. If omitted, no change is made.

vOffs   If present, this sets the y axis display offset to apply between channels in the group. The `nth` channel has a visual offset of `(n-1)*offs`.

Returns  The current locked state of the group unless `opt%` was -1, when it returns the y axis visual offset per channel for the group.

**See also:**
```
ChanOrder(), YAxisMode(), YHigh(), YLow(), YRange()
```

# YAxisMode()

This function controls what is drawn in a y axis and where the y axis is placed with respect to the data.

```
Func YAxisMode({mode%{, hal%{, hsp%{, hvp%}}}});
```

mode%   Optional argument that controls how the axis is displayed. If omitted or negative, no change is made. Positive values are the sum of :

　　1　Hide all the title information.
　　2　Hide all the unit information.
　　4　Hide y axis small ticks. These are also hidden when big ticks are hidden.
　　8　Hide y axis numbers. These are also hidden when big ticks are hidden.

16    Hide the big ticks and the vertical line that joins them. This will also hide the small ticks and numbers as there is nowhere to put them.

32    Scale bar axis. If selected add 4 to remove the end caps.

4096   Place the y axis on the right of the data.

8192   Horizontal text for the title and units.

hal%    The horizontal label alignment as -1 or omitted for no change, 0 for centred, 1 for aligned to the edge.

hsp%    Set the horizontal label character space in the range 2-17 or -1 or omitted for no change.

hvp%    Set the horizontal label vertical position as 0 for centred, 1 for top, 2 for bottom or -1 or omitted for no change.

Returns   If `mode%` is positive, omitted or -1 it returns the `mode%` value at the time of the call or a negative error code. Set `mode%` to -2 to return the current value of `hal%`, -3 for `hsp%` or -4 for `hvp%`.

**See also:**
`ChanNumbers(), YAxis(), YAxisStyle(), YHigh(), YLow(), YRange()`

# YAxisStyle()

This function controls the y axis major and minor tick spacing. If you set values that would cause illegible or unintelligible axes, they are stored but not used unless the axis range or scaling changes to make the values useful.

**Func YAxisStyle(cSpc, opt%{, major});**

cSpc    A channel specifier  or -1 for all, -2 for visible and -3 for selected channels. When multiple channels are specified, returned values are for the first channel.

opt%    Values greater than 0 set the number of subdivisions between major ticks. 0 sets automatic small tick calculation. Use -1 for no change. Values less than -1 return information, but do not change the axis style

major   If present and `opt%` is greater than -2, values greater than 0 sets the major tick spacing. Values less than or equal to 0 select automatic major tick spacing.

Returns   If `opt%` is -2 this returns the current number of forced subdivisions or 0 if they are not forced. If `opt%` is -3 this returns the current major tick spacing if forced or 0 if not forced. Otherwise the return value is 0 or a negative error code. If multiple channels are specified the return value is for the first channel in the list.

**See also:**
`YAxis(), YAxisMode(), YHigh(), YLow(), YRange(), XAxisStyle()`

# YHigh()

This function returns the current upper limit of the y axis in a data or XY view.

**Func YHigh(chan%);**

chan%    A channel number (1 to n). The channel number is ignored for an XY view.

Returns   The value at the appropriate end of the axis.

**See also:**
`ChanTitle$(), ChanUnits$(), YLow(), YRange(), Optimise()`

# Yield()

This function suspends script operation for a user-defined time and allows the system to idle. During the idle time, invalid screen areas update, you can interact with the program and the system has the opportunity to do housekeeping. If your script runs for long periods without using `Interact()` or `Toolbar()`, adding an occasional `Yield()` can make it feel more responsive and stop the operating system marking Signal as "not responding". However be aware that `Yield()` uses up time and slows down script execution, even when used to give the shortest possible idle time. You should therefore avoid putting calls to `Yield()` inside loops, particularly deep inside nested loops, unless this is definitely necessary.

**Func Yield({wait{, allow%}});**

wait   The minimum time to wait, in seconds. If omitted or 0, Signal gives the system one idle cycle. If greater than 0 and there is still waiting time left after an idle cycle completes, other processes are given the opportunity to run between idle cycles. If set negative, there is no idle cycle but the `allow%` argument is applied.

allow%  This defines what the user can do during the wait period, if this argument is omitted a default value of zero is used. See `Interact()` for the possible `allow%` values. The `allow%` value is cancelled and the allow settings restored to the previous state after the yield finishes unless `wait` is negative. An `allow%` value of 0 would restrict the user to inspecting data and positioning cursors in a single, un-moveable window, but being able to switch frames. An `allow%` value of 8192 is the same but without changing frames.

Returns   The function returns 1. We may add more return codes in future versions.

**See also:**
`Interact(), Seconds(), Toolbar(), YieldSystem()`

# YieldSystem()

To share the system CPU(s) among competing processes, the operating system allocates time slices of around 10 milliseconds based on process priorities and recent process activity. A process can surrender a time slice if it has nothing to do. A typical application spends most of its time waiting for user input, which appears as messages in the application message queue; it will surrender a time slice if the message queue is empty unless it has a task to work on. Signal normally surrenders time slices, but if you run a script, it runs for the full time slice unless it is in `Yield()`, `Interact()`, or you use `ToolBar()` or `DlgShow()` without an idle routine.

The `YieldSystem()` command causes Signal to surrender the current time slice and suspends the user interface and script thread for a user-defined time or until a new message arrives in the Signal input queue. It has no effect on sampling, which runs in a separate thread. Unlike `Yield()`, it does not allow Signal to idle.

**Proc YieldSystem({wait});**

wait   The time to wait, in seconds, before resuming the thread. Values are rounded to the nearest millisecond. Values greater than 10 are treated as 10 seconds; values less than –10 are treated as –10 seconds.

For `wait` values greater than 0, the wait is ended by unserviced messages; keyboard and mouse activity, timers for screen updates and the like cause messages. If `wait` is 0 or omitted, the current time slice is surrendered, but if Signal is the highest priority task it will be re-scheduled immediately. Negative values suspend Signal for `–wait` seconds regardless of messages.

`YieldSystem()` with wait values greater than 0 returns immediately if there are messages in the input queue. Unless you allow Signal to idle, either with a `Yield()` call or with `Toolbar()` or `DlgShow()`, there will always be pending messages, so it will have no effect. If you have a script loop that causes 100% CPU usage, inserting:

```
Yield();YieldSystem(0.001);
```

Will give other processes a chance to run. Increasing the wait time up to 0.05 will further reduce the CPU usage. Larger values have little additional effect due to timer messages ending the wait early. To give as much system time as possible to other tasks without allowing Signal to idle, you can use:

```
YieldSystem(-0.001);
```

In this case, increasing the time to –10.0 will have an effect; Signal will feel completely unresponsive until the time period has elapsed.

**See also:**
`Interact(), DlgShow(), Seconds(), Toolbar(), Yield()`

# YLow()

This function returns the current lower limit of the y axis in a data or XY view.

**Func YLow(chan%);**

chan%   A channel number (1 to n). The channel number is ignored for an XY view.

Returns   The value at the appropriate end of the axis.

**See also:**
```
ChanTitle$(), ChanUnits$(), YHigh(), YRange(), Optimise()
```

# YRange()

This sets the y axis range for a channel or grid or XY view. Attempting to set the range for a display mode that doesn't have a y axis is not an error, but has no effect. If the y range changes, the display is invalidated, but waits for the next `Draw()`.

**`Proc YRange(chan%|chan%[]|chan${, low, high});`**

`chan%`  A channel number (1 to n) or you can also use -1 for all channels, -2 for all visible channels, -3 for all selected channels, -4 for waveform channels or -5 for marker channels, -6 for selected waveform channels or visible if none selected, -7 for visible waveform channels or -8 for selected waveform channels. The channel number is ignored in an XY view as there is only one axis for all channels.

This can also be an integer array. If it is, the first array element holds the number of channels in the list. This is followed by a list of positive channel numbers.

`chan$`  A string to specify channel numbers, such as `"1,3..8,9,11..16"`.

`low`  The value for the bottom of the y axis. If omitted, and the channel type has a known range, Signal sets `low` and `high` to suitable limits. For example, for a normal waveform channel the limits are those set by the 16-bit nature of the data, if there is no underlying 16-bit data the available data range is used.

**`high`**  The value for the top of the y axis. If **`low`** and **`high`** are the same, or stupidly close, the range is not changed.

### Grid view
In a grid view, this is used to set to top line to display in the grid. The channel specification is currently unused and should be set to 0 for future compatibility. Use the `Draw()` command or `XRange()` to set the first column.

**`Proc YRange(0, line);`**

`line`  The top line number to display. The first line is 0.

**See also:**
```
YHigh(), YLow(), ChanScale(), ChanOffset(), Optimise()
```

# Z

# ZeroFind()

Find a zero (root) of a user-defined continuous function using Brent's method. You must supply a range to search that contains a zero. The method is iterative with each iteration reducing the search range. Iterations stop when a zero is found, the search range becomes smaller than a supplied tolerance or an iteration limit is reached. The need to supply a range that brackets a zero means that you must have some understanding of the general shape of the function before usng `ZeroFind()`.

**`Func ZeroFind(&x, f(x), a, b{, tol{, maxIt%}});`**

`x`  A real variable that is returned with a value that is within `tol` of a position for which `f(x)` is zero.

`f`  The name (no brackets or argument) of a user-defined function that takes a single real argument and that returns a real value.

`a,b`  These values define a range to be searched for a zero. `f(a)` and `f(b)` must both be non-zero and must have different signs.

`tol`  This specifies how close the returned x must be to the exact result and must be positive. If omitted or zero, the tolerance is set based on the root position and floating point precision.

`MaxIt%`  The maximum number of iterations of the algorithm in the range 1 to 200. If omitted, a maximum of 100 iterations are set.

Return The number of iterations left after the tolerance is achieved, or 0 if it was not, or -1 if the initial range (a, b) does not include a zero.

**Examples**

The following example calculates the roots of the equation $x^2-2=0$.

```
Func root2(x) return x*x-2 end;

var r1,r2;
ZeroFind(r1, root2, 0, 2);     'Find a root between 0 and 2
ZeroFind(r2, root2, 0, -2);    'Find a root between 0 and -2
printlog("Roots at %g and %g\n", r1, r2);
```

The result of this example could have been obtained by simple algebra; the roots are obviously $\pm\sqrt{2}$. However, in some cases the function may not be analytic. For example, suppose we want to demonstrate the often quoted result that 95% of normally distributed values lie within two standard deviations of the mean. The indefinite integral of a Gaussian with unit standard deviation can be obtained with:

```
Func NormProp(x) return GammaP(0.5, 0.5*x*x); end;
```

This returns the fraction of the data that lies within x standard deviations of the mean of a normal distribution. We want the value of x for which `NormProp(x) = 0.95`. Put another way, we want `NormProp(x)-0.95` to be zero. The following script does this:

```
Func NormProp(x) return GammaP(0.5, 0.5*x*x); end;
Func Root(x) return NormProp(x)-0.95 end;

var sd;
ZeroFind(sd, Root, 0, 5);     'get 95% level
printlog("95%% at %4.2f\n", sd);
```

You can use this function in some circumstances to calculate the inverse of a function when no other method is available. That is, if $y = f(x)$, then $x = f^{-1}(y)$. For example, in the above case we were finding the inverse of the `NormProp(x)` function at the specific value 0.95. To tabulate the function for a range of values we would change the `Root()` function to:

```
var prop;
Func Root(x) return NormProp(x)-prop end;
```

Now, by setting prop to a value in the range between 0 and 1 (but not to 0 and 1), we can map the inverse function:

```
for prop := 0.01 to 0.99 step 0.01 do
    ZeroFind(sd, Root, 0, 100);
    printlog("%4.2f %5.3f\n", prop, sd);
next;
```

We have omitted 0 and 1 from our range because we cannot set a range that meets the criteria for the arguments a and b for these values (the result from `NormProp()` can only lie in the range 0 to 1, and only reaches 1 at infinity).

# Curve fitting

# Introduction

It frequently happens that you have a set of data values $(x_1,y_1)$, $(x_2,y_2)$ … $(x_n,y_n)$ that you wish to test against a theoretical model $y = f(x, \mathbf{a}_0, \mathbf{a}_1, \mathbf{a}_2…)$ where the $\mathbf{a}_i$ are coefficients that are to be set to constant values which give the *best fit* of the model to the data values.

For example, if we were looking at the extension of a spring ($y$) as it is loaded by weights ($x$), we might wish to fit the straight line $y = \mathbf{a}_0 + \mathbf{a}_1 x$ to some measured data points so that we could measure a weight by the extension it caused. A careful experimenter might also wish to know what the probable error was in $\mathbf{a}_0$ and $\mathbf{a}_1$ so that the probable error in any weight deduced from an extension would be known. An even more cautious experimenter might want to know if the straight-line formula was likely to model the measured data.

To avoid repeating definitions throughout the remainder of this information the following will be taken as defined. We apologise to the statisticians who may read the following and shudder.

### mean

Given a set of *n* values $y_j$, the mean is the sum of the *n* values divided by *n*.

### variance

If the mean of a set of *n* data values $y_j$ is $y_m$, then the variance (sigma squared) of this set of values is: the sum of the squares of $y_j - y_m$ divided by n if $y_m$ is known independently of the data values and by (*n* - 1) if $y_m$ is calculated from the data values $y_j$.

For a data set of any reasonable size, the use of *n*-1 or *n* in the denominator should make little difference.

### standard deviation

The standard deviation (sigma) of a data set is the square root of the variance. Both the variance and the standard deviation are used as measures of the width of the distribution.

# Normal distribution

If you measure a data value in any real system, there is always some error in the measurement. Once you have made a (very) large number of measurements, you can form a curve showing the probability of getting any particular value. One would hope that this error distribution would show a large peak at the "correct" value of the measurement and the width of this distribution would show the spread of likely errors.



There is a particular error distribution that often occurs, called the Normal distribution. If a set of measurements is normally distributed, with a mean ym and standard deviation sigma, then the probability of measuring any particular value y is proportional to:

$P(y) \propto \exp(-\frac{1}{2}(y-y_m)^2/sigma^2)$

It is for this distribution of errors that we have the well-known result that 68% of the values lie within one standard deviation of the mean, that 95% lie within two standard deviations and that 99.7% lie within three standard deviations. Of course, **if the error distribution is not normal, these results do not apply**.
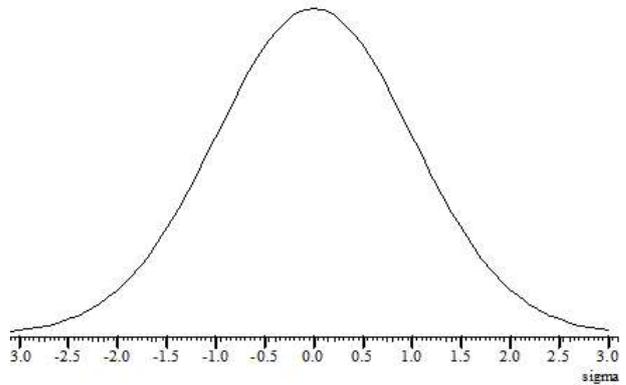
# Chi squared

The fitting routines given here define *best fit* as the values of $\mathbf{a}_i$ (the coefficients) that minimise the chi-squared value defined as the sum over the measured data points of the square of the difference between the measured and predicted values divided by the variance of the data point:

If the sigma of each data point is unknown, then the fitting routines can be used to minimise the sum of the squares of the differences between the actual data values and the values predicted by the fitted function which produces the same result as a chi-squared fit would produce if the variance of the errors at all the data points was the same. This is commonly called least-squares fitting (meaning that the fit minimises the sum of squares of the errors between the fitted function and the data).

Chi-squared fitting is also a maximum likelihood fit if the errors in the data points are normally distributed. This means that as well as minimising the chi-squared value, the fit also selects the most probable set of coefficients that model your data. If your data measurement errors are not normally distributed you can still use this method, but the fit is not maximum likelihood.

If your errors are normally distributed and if you know the variance(s) of the data points, you can form good estimates of the variance of the fitted coefficients, and you can also test if the function you have fitted is likely to model the data.

If your errors are normally distributed but you do not know the variance of the errors at the data points, you can make an estimate of the variance of the errors (based on the assumption that the variance is the same for them all

and that the model does fit the data), by fitting your model and calculating the variance from the errors between the best fit and the data. Having done this, you cannot then use this variance to test if the fit is likely to model the data.

# Residuals

Once your fit is completed, it is a good idea to look at the graph of the errors between your original data and the fitted data (the residuals or residual errors). If your errors are normally distributed and are independent, you would expect this graph to be more or less horizontal with no obvious trends. If this is not the case, you should consider if the correct model function has been selected, or if the fitting function has found the true minimum.

# Linear fit Non linear fit

A linear fit is one in which the theoretical model $y = f(x, \mathbf{a}_0, \mathbf{a}_1, \mathbf{a}_2 \ldots)$ can be expressed as $y = \mathbf{a}_0 f_0(x) + \mathbf{a}_1 f_1(x) + \mathbf{a}_2 f_2(x) \ldots$ for example $y = \mathbf{a}_0 + \mathbf{a}_1 x + \mathbf{a}_2 x^2$. Linear fits are relatively quick as they are done in one step. Usually, the only thing that can cause a problem is if the functions $f_i(x)$ are not linearly independent. The methods we use can usually detect this problem, and can still give a useful result.

A non-linear fit means all other cases, for example, $y = \mathbf{a}_0 \exp(-\mathbf{a}_1 x) + \mathbf{a}_2$. We solve these types of problem by making an initial guess at the coefficients (and ideally providing a range of values that the result is known to lie in) and then improving this guess. This process repeats until some criterion is met. Each repeat is called an *iteration*, so we call this an iterative process.

# Covariance array

Several of the script fitting routines return a covariance array. If you have n coefficients, this array is of size n by n and is diagonally symmetric. If the errors in the original data points are normally distributed, the diagonal elements of this array are the variances in the values of the fitted coefficients. The remaining elements are the co-variances of pairs of the fitting parameters and can be used to estimate errors in derived values that depend on the product of two of the coefficients. If the errors are not normally distributed, the further away from normal the errors are, the less useful is the covariance array as a direct indication of the variances in the fitted coefficients.

For example, in the case of the linear fit:

$$y = \mathbf{a}_0 + \mathbf{a}_1 x + \mathbf{a}_2 x^2$$

you might collect your three coefficients in the array `coef[]`, and the covariance in the array `covar[][]`. In this case, the $\mathbf{a}_0$ value is returned in `coef[0]` and its variance in `covar[0][0]`, the $\mathbf{a}_1$ value is returned in `coef[1]` and its variance in `covar[1][1]`, and the $\mathbf{a}_2$ value is returned in `coef[2]` and its variance in `covar[2][2]`.

Because the array is diagonally symmetric, `covar[i][j]` is equal to `covar[j][i]` and the off-diagonal elements are the expected variance in the product of pairs of the coefficients, so `covar[1][2]` is the variance of $\mathbf{a}_1\mathbf{a}_2$.

If you have not supplied the standard deviations of the errors in the data points, the covariance array is calculated on the assumption that all data points have a standard deviation of 1.0, and the covariance array is incorrectly scaled. In this case, if inspection of the residuals leads you to the conclusion that the function does indeed fit the data and that the errors are more or less the same for all values and not too far from normally distributed, then you can scale the covariance array to the correct values by multiplying all the elements of the array by the sum of squares of the errors between the data and the fitted values divided by the number of data points. If there are `nD%` data points and the sum of squares of the errors is `errSq`, then use `ArrMul(cover[][], errSq/nD%);` to rescale the covariance.

### What does the covariance mean?

Having fitted our data, we would like some idea of how the errors in the original data feed through to uncertainties in the values of the coefficients. The best way to do this is to obtain many sets of (x,y) data and fit our coefficient to each set. Then we can inspect the values of the coefficients and obtain a mean and standard deviation for each coefficient. However, this is very time consuming.

If the errors in the data are normally distributed (or not too far from this ideal case) and known, then the covariance array gives you some useful information. The square root of the covariance for a particular coefficient is the

expected standard deviation in that value (given that the remaining coefficients remain fixed at optimum values). In script language terms, the standard deviation of `coef[i]` is `sqrt(covar[i][i])`.

In this case you would expect the coefficient to be within one standard deviation of the "correct" result 68% of the time, within 2 standard deviations 95% of the time and within 3 standard deviations 99.7% of the time.

# Testing the fit

If the errors in the original data are normally distributed and known (not calculated from the fit), and you know the chi-squared value for the fitted data, you can ask the question, "Given the known errors in the original data, how likely is it that you would get a value of chi-squared at least this large given that the data is correctly modelled by the fitting function plus normally distributed noise?" The answer is (at least in terms of the script language) that the probability is: `GammaQ((nData% - nCoef%)/2.0, chiSq/2.0);` where `nData%` is the number of data points to be fitted, `nCoef%` is the number of coefficients that were fitted and `chiSq` is the chi-squared value for the fit. `GammaQ()` is the incomplete Gamma function.

If you want to follow this result up in a statistical textbook, you should look up *chi-squared distribution for n degrees of freedom*. In our case, we have `nData%-nCoef%` degrees of freedom.

If the fit is reasonable, you should expect a probability value between 0.1 and 1 (but be a bit suspicious if you always get values close to 1.0, as you may have overestimated the errors in the data). If the wrong function has been fitted or if the fit is poor you usually get a very small probability.

Intermediate values (0.0001 to 0.1) may indicate that the errors in the original data were actually larger than you thought, or they may indicate that the model does not explain all the variation in the data.

# Fitting routines

The `ChanFit...()` family of commands give a script the same capability as the Analysis menu Fit Data command. These commands were added to Signal at version 3.

### ChanFit()
This command associates a fit with a data channel in a data or XY view, performs the fit and returns basic fit information.

### ChanFitCoef()
This gets and sets fit coefficients, coefficient limits and the hold flags. It is equivalent to the coefficient page of the Fit Data dialog.

### ChanFitShow()
This controls the display of the fitted data.

### ChanFitValue()
This returns the value at a given x position of the fitted function.

The remaining fitting functions allow you to fit curves to data in arrays:

### FitPoly()
This fits a polynomial of the form $y = \mathbf{a}_0 + \mathbf{a}_1 x + \mathbf{a}_2 x^2 + \mathbf{a}_3 x^3 \ldots$

### FitLine()
This fits a straight line to data in a data view.

### FitLinear()
This fits $y = a_0 f_0(x) + \mathbf{a}_1 f_1(x) + \mathbf{a}_2 f_2(x) \ldots$ where the $f_n(x)$ are user-defined functions of $x$.

### FitExp()

This fits multiple exponentials of the form $y = \mathbf{a}_0 \exp(-x/\mathbf{a}_1) + \mathbf{a}_2 \exp(-x/\mathbf{a}_3)\ldots$

### FitGauss()

This fits Gaussians of the form $y = \mathbf{a}_0 \exp(-\frac{1}{2}(x-\mathbf{a}_1)^2/\mathbf{a}_2^2) + \mathbf{a}_3 \exp(-\frac{1}{2}(x-\mathbf{a}_4)^2/\mathbf{a}_5^2) + \ldots$

### FitSin()

This fits multiple sinusoids of the form $y = \mathbf{a}_0 \sin(\mathbf{a}_1 x + \mathbf{a}_2) + \mathbf{a}_3 \sin(\mathbf{a}_4 x + \mathbf{a}_5) + \ldots$

### FitNLUser()

This fits a user-defined function of the form $y = f(x, \mathbf{a}_0, \mathbf{a}_1, \mathbf{a}_2\ldots)$ to a set of data points where the $\mathbf{a}_i$ are constant coefficients to be determined. You must be able to calculate the differential of the function $f$ with respect to each of the $\mathbf{a}_i$ coefficients. This is the most general fitting routine, and also the slowest and most complex to use.

### GammaP(), GammaQ()

Use these functions to calculate the error function *erf(x)*, the cumulative Poisson probability function and the Chi-squared probability function. These are very useful when considering probabilities of fits associated with the normally distributed data.

### LnGamma()

This calculates the natural logarithm of the Gamma function. It can be used to calculate the log of large factorials and is useful when working with probability distributions.

# Digital filtering

Filtering is used to remove unwanted frequency components from waveforms and can also be used to differentiate a signal. In Signal we provide you with two basic types of filter: Finite Impulse Response (FIR) and Infinite Impulse Response (IIR). Both types of filter have their advantages and disadvantages.

## FIR and IIR filters

Filtering is used to remove unwanted frequency components from waveforms and can also be used to differentiate a signal. In Signal we provide you with two basic types of filter: Finite Impulse Response (FIR) and Infinite Impulse Response (IIR). Both types of filter have their advantages and disadvantages.

### IIR filters

These are similar to analogue filters, and we design them by mapping standard Butterworth, Bessel, Chebyshev filters and resonators into their digital forms. IIR filters have these advantages:



- They can generate much steeper edges and narrower notches than FIR filters for the same computational effort.

- IIR filters are causal; they do not use future data to calculate the output, so there is no pre-ringing due to transients.
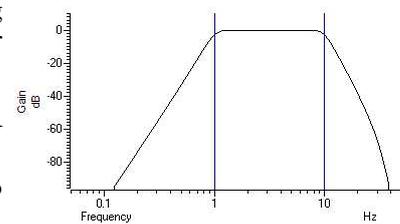
They also have disadvantages:

- IIR filters are prone to stability problems particularly as the filter order increases or when a filter feature becomes very narrow compared to the sample rate.

- IIR filters impose a group delay on the data that varies with frequency. This means that they do not preserve the shape of a waveform, in particular, the positions of peaks and troughs will change.

The output of an IIR filter may take a long time to settle down from the discontinuity at the start (transition from no data to the supplied data).

### FIR filters

We describe FIR filters in terms of frequency bands: *pass bands*, *stop bands* and *transition gaps*. You define a filter by the arrangement of bands and the corner frequencies of each band. FIR filters have these advantages:



- They are unconditionally stable as they do not feedback the filter output to the input.

- There is no phase delay through the filter, so peaks and troughs do not move when data is filtered (this is called *linear phase* in the literature).
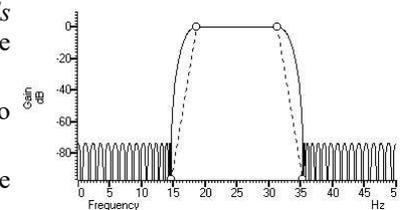
They also have disadvantages:

- They are poor at generating very narrow notches or narrow band pass filters.

- The narrowest frequency band or band gap is limited by the number of coefficients (we allow up to 511 coefficients).

- FIR filters are not causal; they use future as well as past data to generate each output point. A transient in the input causes effects in the output before the transient.

If your FIR filter has n coefficients, the first and last n/2 output points are estimates due to the discontinuity at the start and end of the data.

### So which to choose?

If you want a differentiating filter, you have no choice as we have not implemented differentiators for IIR filters. Unless one of the disadvantages of the FIR filter is a problem, you will likely have fewer unexpected effects with an FIR filter.

However, there are circumstances in which only an IIR filter will do. If you need a high Q notch filter or resonator, then use an IIR filter. If you are interested in small changes just before a large discontinuity, only the IIR filter will

help you. However, make sure that you understand the disadvantages of IIR filters before you depend on their output.

# Digital filter dialog

The Analysis menu Digital filters command is available when you have a data or memory view open. You can choose to apply Finite Impulse Response (FIR) filters or Infinite Impulse Response (IIR) filters. Apart from the dialog title, the initial dialog display is the same for IIR and FIR filters. You can apply one of twelve stored digital filters to a waveform channel, or you can create your own digital filter. You can also load and save additional sets of filters from the dialog.

The dialog shows the original waveform using the overdrawn waveform colour and a filtered version in the colour you have set for waveform data. Whenever you change the filter, the display updates to show the effect of the change.
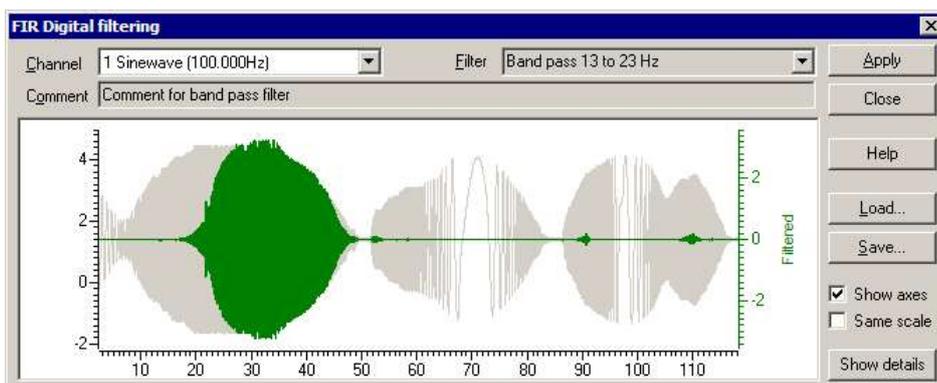


### Show axes

You can choose to Show axes for the original data and for the filtered version. The axis for the original data is always on the left. If a separate axis is required for the filtered data, it is drawn in the filtered data colour on the right.

### Same scale

Initially, the filtered data is drawn at the same scale as the original data. However, sometimes this is inconvenient, for example when high-pass filtering a signal with a significant DC offset or when the result of filtering is very small compared to the original. If you clear the Same scale check box, the filtered data is scaled to fit in the window independently of the scaling of the original waveform.

### Channel

The Channel field allows you to select a waveform channel to filter.

### Filter

The Filter field of the dialog box selects the filter to apply. There are normally 12 filters to choose from. When you first open the dialog, this field is grey, indicating that you cannot edit the filter name. If you display the filter details you can modify the filter name.

### Comment

The Comment field is for any purpose you wish; there is one comment per filter. When you first open the dialog, this field is grey, indicating that you cannot edit the comment. Click the Show details button to edit the comment.

The Filter field of the dialog box selects the filter to apply and the Channel field sets the waveform channel to filter.

The Close button shuts the dialog and will ask if you want to save any changed filter and the Help button opens the on-line Help at the digital filtering topic.

### Close

Click the Close button to shut the filtering dialog. If you have made a change to any of the filters, or loaded a new filter set, you are asked if you want to save the current set of filters as your standard filter set.

### Load and Save

You can choose to save the current set of filters to a `.cfb` file, or to load a new set of filters from a `.cfb` file. If you are working with FIR filters, this only saves or loads FIR filters. If you are working with IIR filters, this only

saves or loads IIR filters. Loading a new filter set does not change your standard filter set, however, you will be asked if you want to change your standard set when you close the dialog.

### Show details

The Show details button increases the dialog size to display a new area in which you can design and edit filters. Click this button again to hide the new dialog area. When you display the filter details, the Filter and Comment fields become editable. If you change a filter or create a new filter or load a new set of filters from a file, you will be prompted to save the filter bank when you close the digital filter dialog. The details are different for FIR and IIR filters.

# Apply

The Apply button opens a new dialog in which you set which channels and frames to filter with the Channels and Frame list fields. The Frame subset field can be used to further specify which frames are to be filtered. The Selected frame state field only appears if you select Frame state = xxx in either the Frame list or the Frame subset field. The channels should all have the same sampling rate as the one you were previewing in the main dialog. Channels of a differing sampling rate will be ignored – this situation should only occur with data imported into Signal.

As applying a filter can be a lengthy process, a progress dialog appears with a Cancel button during the filtering operation.
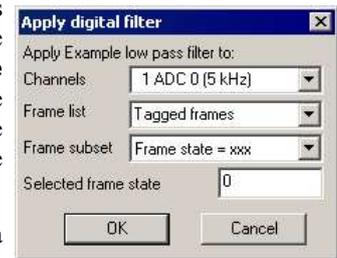
### FIR filter details

If the filter is of length $n$, then $n/2$ points around each input data point are used to produce each output point. When there is no input data available before or after a point, the filter uses a duplicate of the nearest input point as an estimate of the data value. This means that the $n/2$ output points next to either end of the input data should not be used for any critical purpose.

### IIR filter details

IIR filters are applied to all the data in the selected area. IIR filters feed back the results of previous filter steps into the current step. This means that the result of the filter on a data point depend on all the data points up to and including that point. For a stable filter, the effect of a distant point is much less than that of an adjacent one. After a discontinuity (the start of a frame), it will take some time for the filter to settle down. Generally, the sharper the filter, the longer it takes for the result to settle. If a filter is unstable or close to unstable, the result may never settle.

# Filter bank

A digital filter definition is complex and it would be tedious to specify all the properties of a filter each time you wanted to apply one to data. To avoid this, Signal contains a filter bank of 12 FIR and 12 IIR filter definitions. This filter bank is saved to the file `Filtbank.cfb` when you close Signal and reloaded when you open it. When you use the digital filter dialog, you specify which filter you want by the filter name.

Script users identify the filter by its type (FIR or IIR) and an index number in the range 0 to 11. Script users also have access to two additional temporary filters with index number -1 (one for FIR and the other for IIR filters) that they can set and use for channel filtering operations without changing the standard filter set.

# FIR filter details

The graph in the details area displays the ideal and actual frequency response of the filter. The ideal response appears as blue solid lines for each pass band linked by dotted lines that mark transition gaps between the bands. All transition gaps have the same width. The calculated frequency response is drawn as solid black lines and is greyed when the filter specification has changed and the response has not yet been calculated.

The mouse pointer changes to indicate the feature it is over: ⬦ for a pass band or stop band, ⬦ for a transition gap and ⬦ for a band corner. The small circles can be dragged sideways to change the slope of the band edges or you can edit the band edges as numbers in the Frequency panel on the right. You can also drag the bands and the transition gaps sideways.
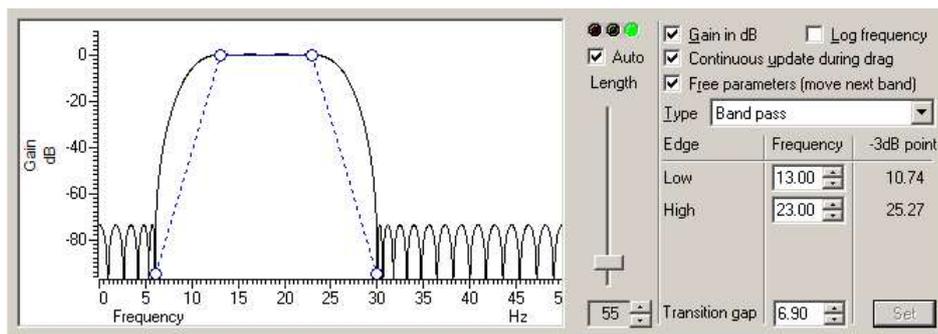
### Set
If you edit the numbers in the Frequency panel, the Set button is enabled so you can force a recalculation of the filter.

### -3dB point
The filters produced by the program are not defined in terms of -3dB corner frequencies and $n$ dB per octave, as is often the case for traditional analogue filters. The 3dB point column is present to help users who are more comfortable describing filter band edges in terms of the 3 dB point.

### Gain in dB
The Gain in dB check box sets the y axis scale to dB if checked, linear if not checked. Using dB is usually the more convenient, except when working on a differentiator.

### Log frequency
You can choose to display the frequency axis as the logarithm of the frequency, which gives you more resolution at low frequencies. However, this can make working with FIR filters more awkward as it removes the visual symmetry of the transition bands. In log mode, the frequency axis extends down to 0.001 of the data sample rate.

### Continuous update
If you check the Continuous update box, the filter is updated while you drag the filter features around. If you have a slow computer and this feels ponderous you can clear the check box, in which case the filter is not recalculated until you stop changing features.

### Free parameters
If you check the Free parameters box, dragged features are not limited by the next band and will push bands along horizontally. If you clear the box, the horizontal motion of a dragged feature is limited by the next moveable object.

### Length, Auto and traffic lights
To the right of the frequency response display is a slider that controls the number of filter coefficients. In general, the more coefficients, the better the filter. However, the more coefficients, the longer it takes to compute them and the longer to filter the data. If you check the Auto box, the program will adjust the number of coefficients for you to produce a useful filter. The "traffic light" display above the slider shows green if the filter is good, amber if the result is usable but not ideal, and red if the result is hopeless.

An FIR filter of length $n$ uses the $n/2$ points before and after each input point to produce each output point. When there is no input data available before or after an input point, the filter uses a duplicate of the nearest point as an estimate of the data value. The $n/2$ output points next to the start or end of a frame of data should not be used for any critical purpose.

# FIR Filter types

The Type field sets the arrangement of filter bands. If you need a filter that is not in this list you can generate it from the script language with the `FIRMake()` command. There are currently 12 different filter types:

### All pass
This has no effect on your signal. This filter type covers the case where you apply a low pass filter designed for a higher sampling rate to a waveform with a much lower sampling rate, so that the pass band extends beyond half the sampling frequency of the new file.

### All stop
This removes any signal; the output is always zero. This filter type is provided to cover the case where you apply a high pass filter designed for a higher sampling rate to a waveform with a much lower sampling rate, so that the stop band extends beyond half the sampling frequency of the new file.

### Low pass
This filter attempts to remove the high frequencies from the input signal. The Frequency field holds one editable number, Low pass, the frequency of the upper edge of the pass band. The stop band starts at this frequency plus the value set by the Transition gap field.
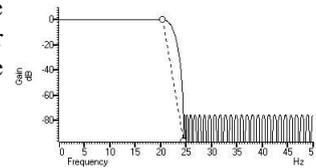


### High pass
A high pass filter removes low frequencies from the input signal. The Frequency field holds one editable number, High pass, the frequency of the lower edge of the pass band. The stop band starts at this frequency less the value set by the Transition gap field.
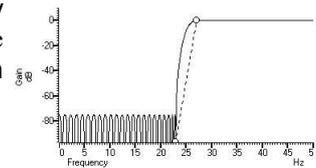


### Band pass
A band pass filter passes a range of frequencies and removes frequencies above and below this range. The Frequency field has two editable numbers, Low and High, which correspond to the two edges of the pass band. The stop band below runs up to Low-Transition gap, and the stop band above from High+Transition gap to one half the sampling rate.
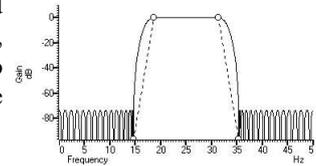


### Band stop
A band stop filter removes a range of frequencies. The Frequency field has two editable numbers, High (the upper edge of the first pass band) and Low (the lower edge of the upper pass band). The stop band below runs from High+Transition gap up to Low-Transition gap.
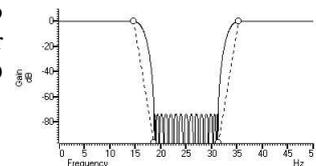


### One and a half low pass
This filter has two pass bands, the first running from zero Hz and the second in the frequency space between the upper edge of the first pass band and one half the sampling rate. The Frequency field has three editable numbers: Band 1 high, Band 2 low and Band 2 high. These numbers correspond to the edges of the pass bands.
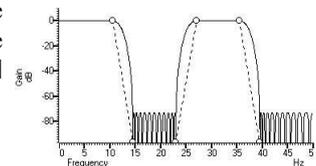


### One and a half high pass
This filter has two pass bands. The second runs up to one half the sampling rate. The first band lies in the frequency space between 0 Hz and the lower edge of the second band. The Frequency field has three editable numbers: Band 1 low, Band 1 high and Band 2 low. These numbers correspond to the edges of the pass bands.
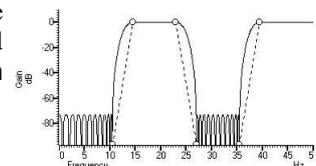
### Two band pass

This filter passes two frequency ranges and rejects the remainder. Both 0 Hz and one half the sampling frequency are rejected. The Frequency field has 4 numeric fields: Band 1 low, Band 1 high, Band 2 low and Band 2 high. These fields correspond to the four edges of the two bands.
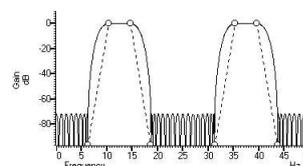
### Two band stop

This filter passes three frequency ranges and rejects the remainder. Both 0 Hz and one half the sampling rate are passed. The Frequency field has 4 numeric fields: Band 1 high, Band 2 low, Band 2 high and Band 3 low. These fields correspond to the four edges of the three bands.
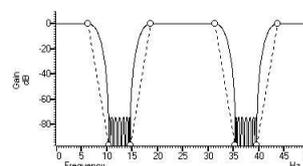
### Low pass differentiator

This filter is a combination of a differentiator (that is the output is proportional to the rate of change of the input) and a low pass filter. The y axis scale is linear, rather than in dB (although you can display it in dB if you wish). There is one editable number in the Frequency field, Low pass, the end of the differential section of the filter.
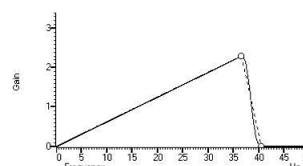
### Differentiator

The output of the filter is proportional to the rate of change of the input. The y axis scale is linear, rather than in dB (although you can display it in dB if you wish). The Frequency field is empty as there is only one band and it extends from 0 Hz to half the sampling rate.
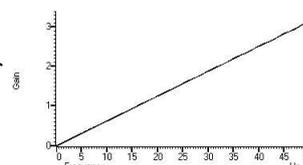
## IIR filter details

We describe IIR filters in terms of a filter type (low pass, high pass, band pass or band stop), the analogue filter model that they are based on (Butterworth, for example), the corner frequencies and the filter order (which determines the steepness of the cut-off outside the desired pass bands). Filters based on Chebyshev designs also require a ripple specification and resonators require a Q factor.
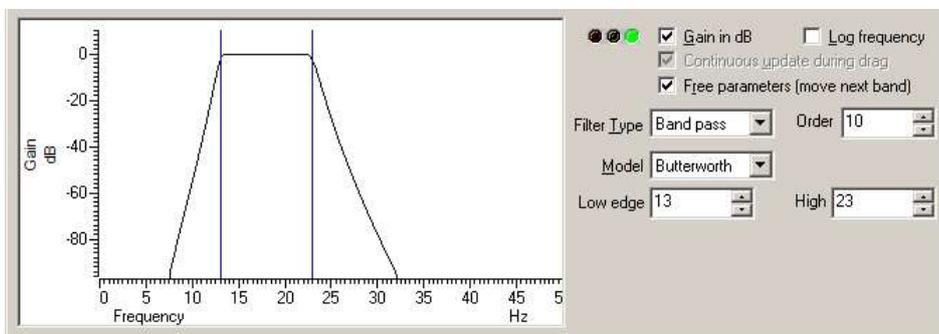
The graph in the details area displays the corner frequencies and the frequency response of the filter. You can adjust the filter by clicking and dragging in this area or by editing the filter parameters as text. The mouse pointer changes to indicate the feature it is over: ⭲⭰ for a corner frequency and ⭥ for an adjustable parameter (ripple or Q factor).

### Gain in dB

The Gain in dB check box sets the y axis scale to dB if checked, linear if not checked. Using dB is usually the more convenient.

### Log frequency

You can choose to display the frequency axis as the logarithm of the frequency, which gives you more resolution at low frequencies. The display extends to 0.0001 of the sample rate. If you need a corner frequency below this you

should consider sampling the signal more slowly in the first place. Alternatively, low pass filter the signal and then use a channel process to down sample it before filtering.

### Continuous update
This is always checked for IIR filters, and is greyed out.

### Free parameters
If you check the Free parameters box, corner frequencies are not limited by the next corner, and will push them along. If you clear the box, corner frequencies cannot be moved past each other.

### Traffic lights and messages
The traffic lights show green if the filter appears to be OK, amber if the filter may be unstable and red if the filter calculation failed, the filter is unstable or if one of the input parameters is illegal. There will usually be an explanatory message in the lower right hand corner of the dialog explaining the problem. In the amber state, the filter may still be usable; you can look at the frequency response and the filtered data to see if the result is acceptable.

### Filter Order
In an analogue filter, the filter order determines the sharpness of the filter, for example Butterworth and Bessel filters tend towards 6n dB per octave, where n is the filter order. In a digital filter, the order determines the number of filter coefficients. The higher the order, the sharper the filter cut-off and also, the more likely the filter is to be unstable due to problems in numerical precision. You can set filter orders of 1 to 10. You should always use the lowest order that meets your filtering criteria. Phase non-linearity gets worse as the filter order increases.

### Filter type
There are four filter types: Low pass, High pass, Band pass and Band stop. However, if you set the filter model to Resonator, there are only two types, being Band pass (this creates a resonator filter) and Band stop (this creates a notch filter). For all filter models except Chebyshev type 2 and Resonator, the frequencies given are the points at which the filter achieves a cut of 3 dB. For Chebyshev type 2 filters, the frequencies are the point at which the filter cut reaches the value set by the Ripple parameter. For Resonators, the frequency is the centre frequency of the resonator.

Low pass    Use this to remove high frequencies and pass low frequencies. This has a single corner frequency.

High pass    Use this to remove low frequencies and pass high frequencies. This has a single corner frequency.

Band pass    This passes a range of frequencies between the Low edge and the high edge. If the filter model is Resonator, then this has a single Centre frequency.

Band stop    This removes a range of frequencies between the Low edge and the high edge. If the filter model is Resonator, then this has a single Centre frequency.

### Filter Model
The IIR filters we provide are digital implementation of standard analogue filter models. The following descriptions use fifth order 1 to 10 Hz band pass filters on 100 Hz data as examples (except for the Resonator filters). The five filter models are:

### Butterworth
This has a maximally flat pass band, but pays for it by not having the steepest possible transition between the pass band and the stop band. This is a good choice for a general-purpose IIR filter, but beware that the group delay can get quite bad near the corners, especially for high-order filters.
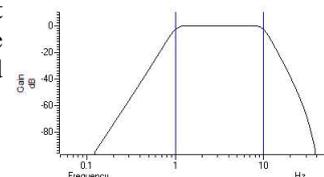
### Bessel

An analogue Bessel filter has the property that the group delay is maximally flat, which means that it tends to preserve the shape of a signal passed through it. This leads to filters with a gentle cut-off. When digitised, the constant group delay property is compromised; the higher the filter order, the worse the group delay.
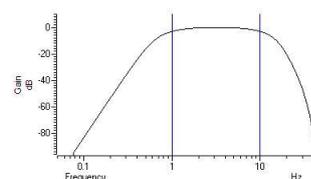
### Chebyshev type 1

Filters of this type are based on Chebyshev polynomials and have the fastest transition between the pass band and the stop band for a given ripple in the pass band and no ripples in the stop band. You can adjust the ripple by dragging the small circles vertically or with the Ripple field to the right of the displayed frequency response.
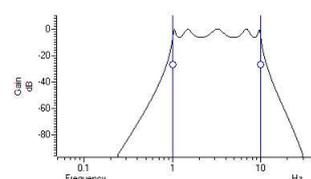
### Chebyshev type 2

Filters of this type are defined by the start of the stop band and the stop band ripple (the minimum cut in the stop band). They have the fastest transition between the pass and stop bands given the stop band ripple and no ripple in the pass band. Drag the small circles to adjust the ripple, or use the Ripple field to the right of the frequency response.
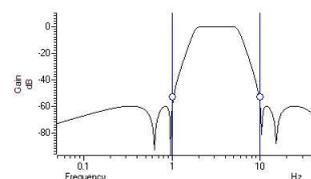
### Resonator

Resonators are defined by a centre frequency and a Q factor. The Q is the centre frequency divided by the width of the resonator at the -3 dB point. You can have a band stop or a band pass resonator. The Q is adjusted by dragging the small circle or with the Q field to the right of the displayed frequency response.

Band stop resonators are also called Notch filters. The higher the Q, the narrower the notch. Notch filters are often used to remove mains hum, but if you do this you will likely need to set notches at the first few odd harmonics of the mains frequency. The example has a very low Q (1.24) to make the filter response visible.
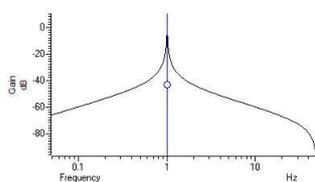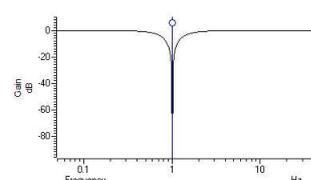
A band pass resonator is the inverse of a notch. Band pass resonators are sometimes used as alternatives to a narrow bandpass filter. The example has a Q of 100.
The higher the Q set for a resonator, the longer it will take for the output to stabilise at the start of the filter output.

# FIR filters technical information

The `FIRMake()`, `FIRQuick()` and `FiltCalc()` script commands and the Analysis menu Digital filters… dialog generate FIR (Finite Impulse Response) filter coefficients suitable for a variety of filtering applications. The generated filters are optimal in the sense that they have the minimum ripple in each defined band. These filter coefficients are used to modify a sampled waveform, usually to remove unwanted frequency components. The algorithmic heart of the filter coefficient generation is based on the well-known FORTRAN program written by Jim McClellan of Rice University in 1973 that implements the *Remez exchange algorithm* to optimise the filter.

The theory of FIR filters is beyond the scope of this document. Readers who are interested in learning more about the subject should consult a suitable text book, for example *Theory and Application of Digital Signal Processing* by Rabiner and Gold published by Prentice-Hall, ISBN 0-13-914101.

# FIR filtering
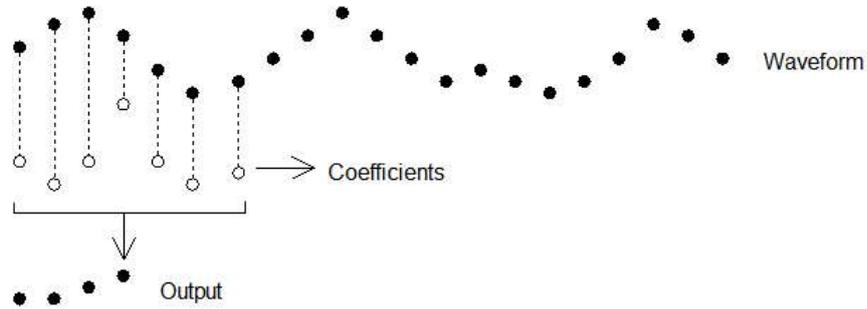


This diagram shows the general principle of the FIR filter. The hollow circles represent the filter coefficients, and the solid circles are the input and output waveforms. Each output point is generated by multiplying the waveform by the coefficients and summing the result. The coefficients are then 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 7 coefficients, there is no time shift caused by the filter. With an even number of coefficients, there is a time shift in the output of half a sample period.

# Frequencies

The Analysis menu Digital filters… command deals with frequencies in Hz as this is comfortable for us to work with. However, if you calculate a FIR filter for one sampling rate, and apply the same coefficients to a waveform sampled at another rate, all the frequency properties of the filter are scaled by the relative sampling rates. That is, the frequency properties of an FIR filter are invariant when expressed as fractions of the sampling rate, not when expressed in Hz.

It is usually more convenient when dealing with real signals to describe filters in terms of Hz, but this means that each time a filter is applied to a waveform the sampling rate must be checked. If the rate is different from the rate for which the filter was last used, the coefficients must be recalculated. Unless you use the `FIRMake()` script command, Signal takes care of all the frequency scaling and recalculation for you. The remainder of this description is to help users of the `FIRMake()` script command, but the general principles apply to all the digital filtering commands in Signal.

Users of the `FIRMake()` script command must specify frequencies in terms of fractions of the sample rate from 0 to 0.5. For example, if you were sampling at 10 kHz and you wanted to refer to a frequency of 500 Hz, you would call this 500/10000 or 0.05.

# Example filter

The heavy lines in the next diagrams show the results obtained by `FIRMake()` when it designed a low pass filter with 80 coefficients with the specification that the frequency band from 0 to 0.08 should have no attenuation, and that the band from 0.16 to 0.5 should be removed. We can specify the relative weight to give to the ripple in each band. In this case, we said that it was 10 times more important that the *stop band* (0.16 to 0.5) should pass no signal than the *pass band* should be completely flat.
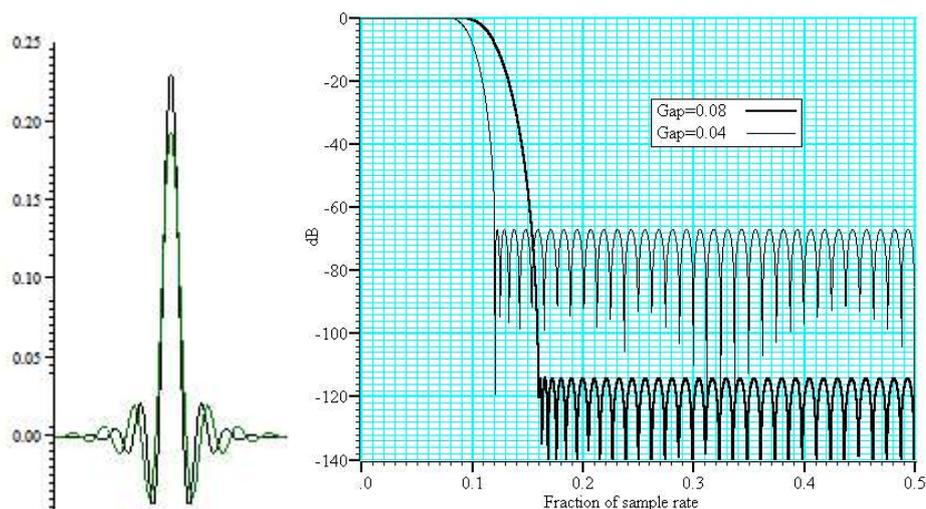
We have shown the coefficients as a waveform for interest as well as the frequency response of the filter. The shape shown below is typical for a band pass filter. One way of understanding the action of the FIR filter is to think of the output as the correlation of the waveform and the filter coefficients.

## Coefficients and frequency response for low pass filters

The frequency response is shown in dB, which is a logarithmic scale. A ratio $r$ is represented by $20 \log_{10}(r)$ dB. A change of 20 dB is a factor of 10 in amplitude, 6 dB is approximately a factor of 2 in amplitude. The graph shows that a frequency in the stop band is attenuated by over 110 dB (a factor of 300,000 in amplitude with respect to the signal before it was filtered).

Because we didn't specify what happened between a frequency of 0.08 and 0.16 of the sampling rate, the optimisation pays no attention to this region. You might ask what happens if we make this transition gap smaller. The lighter line in the graph shows the result of halving the width of the gap by making the stop band run from 0.12 to 0.5. The filter is now much sharper. However, you don't get something for nothing. The attenuation in the stop band is reduced from 110 dB to around 70 dB. Although you cannot see it from the graph, the ripple in the pass band also increases by the same proportion (from 1 part in 30,000 to 1 part in 300).

We can restore the attenuation in the stop band by increasing the number of coefficients to around 120. However, there are limits to the number of coefficients it is worth having (apart from increasing the time it takes to calculate the filter and filter the data). Although the process used to calculate coefficients uses double precision floating point numbers, there are rounding errors and the larger the number of coefficients, the larger the numerical noise in the result.

Because the waveform channels are stored in 16-bit integers, there is no point designing filters that attenuate any more than 96 dB as this is a factor of 32768. Attenuations greater than this would reduce any input to less than 1 bit. If you are targeting data stored in real numbers this restriction may not apply.

It is important that you leave gaps between your bands. The smaller the gap, the larger the ripple in the bands.
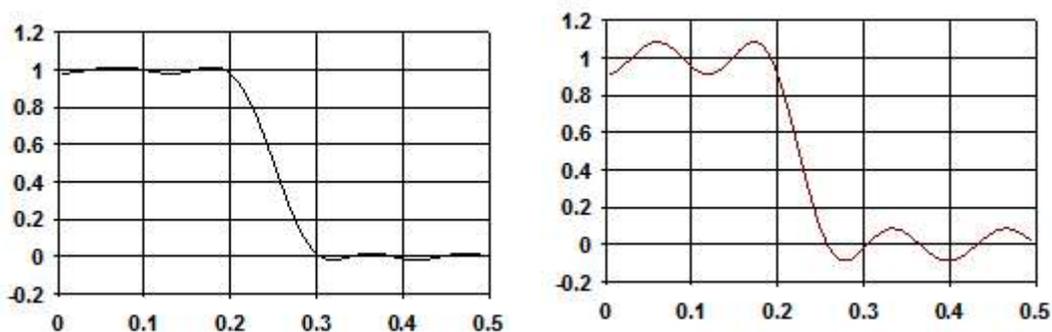


This is illustrated by these two graphs. They show the linear frequency response of two low pass filters, both designed with 18 coefficients (we have used so few coefficients so the ripple is obvious). Both have a pass band of 0 to 0.2, but the first has a gap between the pass band and the stop band of 0.1 and the second has a gap of 0.05.

We have also given equal weighting to both the pass and the stop bands, so you can see that the ripple around the desired value is the same for each band.

As you can see, halving the gap has made a considerable increase in the ripple in both the pass band and the stop band. In the first case, the ripple is 1.76%, in the second it is 8.7%. Halving the transition region width increased the ripple by a factor of 5.

In case you were worrying about the negative amplitudes in the graphs, a negative amplitude means that a sine-wave input at that frequency would be inverted by the filter. The graphs with dB axes consider only the magnitude of the signals, not the sign.

# FIRMake filter types

`FIRMake()` can generate coefficients for four types of filter: Multiband, Differentiators, Hilbert transformer and a variation on multiband with 3 dB per octave frequency cut. The other routines can generate only Multiband filters and Differentiators.

### Multiband filters

The filter required is defined in terms of frequency bands and the desired frequency response in each band (usually 1.0 or 0.0). Bands with a response of 1.0 are called *pass bands*, bands with a response of 0.0 are called *stop bands*. You can also set bands with intermediate responses, but this is unusual. The bands may not overlap, and there are gaps between the defined bands where the frequency response is undefined. You give a weighting to each band to specify how important it is that the band meets the specification. As a rule of thumb, you should make the weight in stop bands about ten times the weight in pass bands.

`FIRMake()` optimises the filter by making the ripple in each band times the weight for the band the same. The ripple is the maximum error between the desired and actual filter response in a band. The ripple is usually expressed in dB relative to the unfiltered signal. Thus the ripple in a stop band is the minimum attenuation found in that band. The ripple in a pass band is the variation of the frequency response from the desired response of unity. In some situations, for example audio filters, quite large ripples in the pass band are tolerable but the same ripple would be unacceptable in a stop band. For example, a ripple of -40 dB in a pass band (1%) is inaudible, but the same ripple in a stop band would allow easily audible signals to pass. By weighting bands you can increase the attenuation in one band at the expense of another to suit your application.

### Differentiators

The output of a differentiator increases linearly with frequency and is zero at a frequency of 0. The differentiator is defined in terms of a frequency band and a slope. The frequency response at frequency *f* is *f * slope*. The slope is usually set so that the frequency response at the highest frequency is no more than 1.

The weight given to each frequency within a band is the weight for that band divided by the frequency. This gives a more accurate frequency response at low frequencies where the resultant amplitude will be the smallest.

Although you can define multiple bands for a differentiator, it is unusual to do so. Almost all differentiators define a single band that starts at 0. Occasionally a differentiator followed by a stop band is needed.

### Hilbert transformers

A Hilbert transformer is a very specialised form of filter that causes a phase shift of -pi/2 in a band, often used to separate a signal from a carrier. The theory and use of this form of filter is way beyond the scope of this document. Unless you know that you need this filter type you can ignore it.

### Multiband with 3dB octave cut

This is a variation on the multiband filter that can be used to filter white noise to produce band limited pink noise. The filter is identical to the band pass filter except that the attenuation increases by 3 dB per octave in the band (each doubling of frequency reduces the amplitude of the signal by a factor of the square root of 2). It is used in exactly the same way as the multiband filter.

# Low pass filter example

A waveform is sampled at 1 kHz and we are interested only in frequencies below 100 Hz. We would like all frequencies above 150 Hz attenuated by at least 70 dB.

A low pass filter has two bands. The first band starts at 0 and ends at 100 Hz, the second band starts at 150 Hz and ends at half the sampling rate. Translated into fractions of the sampling rate, the two bands are 0-0.1 and 0.15 to 0.5. The first band has a gain of 1, the second band has a gain of 0. We will follow our own advice and give the stop band a weight of 10 and the pass band a weight of 1. We will try 40 coefficients to start with, so a possible script is:

```
var prm[5][2];          'Array for parameters
var coef[40];           'Array for the coefficients
'    band start         band end          function          weight
prm[0][0]:=0.00; prm[1][0]:=0.1; prm[2][0]:=1.0; prm[3][0]:= 1.0;
prm[0][1]:=0.15; prm[1][1]:=0.5; prm[2][1]:=0.0; prm[3][1]:=10.0;
FIRMake(1, prm[][], coef[]);
PrintLog("Pass Band ripple=%.1fdB Stop band attenuation=%.1f\n",
         prm[4][0], prm[4][1]);
```

If you run this, the log view output is:

```
Pass Band ripple=-28.8dB Stop band attenuation=-48.8
```

The attenuation in the stop band is only 48 dB, which is not enough. The ripple in the pass band is around 3% of the signal amplitude. We can increase the stop band attenuation in three ways: by increasing the number of coefficients, by giving the stop band more weight, or by making the gap larger between the bands.

We don't want to give the stop band more weight; this would increase the ripple in the pass band. We could probably reduce the width of the pass band a little as the attenuation of the signal tends to start slowly, but we will leave that adjustment to the end. The best way to improve the filter is to increase the number of coefficients. If we increase the size of `coef[]` to 80 coefficients and run again, the output now is:

```
Pass Band ripple=-58.7dB Stop band attenuation=-78.7
```

This is much closer to the filter we wanted. You might wonder if there is a formula that can predict the number of coefficients based on the filter specification. There is no exact relationship, but the following formula, worked out empirically by curve fitting, predicts the number of coefficients required to generate a filter with equal weighting in each of the bands and is usually accurate to within a couple of coefficients. The formula can be applied when there are more than two bands, but becomes less accurate as the number of bands increase.

```
' dB     is the mean ripple/attenuation in dB of the bands
' deltaF is the width of the transition region between the bands
' return An estimate of the number of coefficients
Func NCoefMultiBand(dB, deltaF)
return (dB-23.9*deltaF-5.585)/(14.41*deltaF+0.0723);
end;
```

In our example we wanted at least 70 dB attenuation, and we weighted the stop band by a factor of 10 (20 dB). This causes a 10 dB improvement in the stop band at the expense of a 10 dB degradation of the pass band. Thus to achieve 70 dB in the stop band with the weighting, we need 60 dB without it. If we set these values in the formula (`dB` = 60, `deltaF`=0.05) , it predicts that 67.13 coefficients are needed. If we run our script with 67 coefficients, we get 70.9 dB attenuation, which is close enough!

**A final finesse**

If we look at the frequency response of our filter in the area between the pass band and the stop band, we see that the curve is quite gentle to start with. If you are used to using analogue filters, you will recall that the corner frequency for a low pass analogue filter is usually stated to be the frequency at which the filter response fell by 3 dB which is a factor of root 2]] in amplitude (when the response falls to 0.707 of the unfiltered amplitude).

If we use the analogue filter definition of corner frequency, we see that we have produced a filter that passes from 0 to 0.115 of the sampling rate, and we wanted from 0 to 0.1, so we can move the corner frequency back. This will increase the attenuation in the stop band, and reduce the filter ripple, as it widens the gap between the pass band and the stop band. If we move it back to 0.085, the attenuation in the stop band increases to 84 dB. Alternatively, we could move both edges back, keeping the width of the gap constant. This leaves the stop band attenuation more or less unchanged, but means that the start of the stop band is moved lower in frequency.
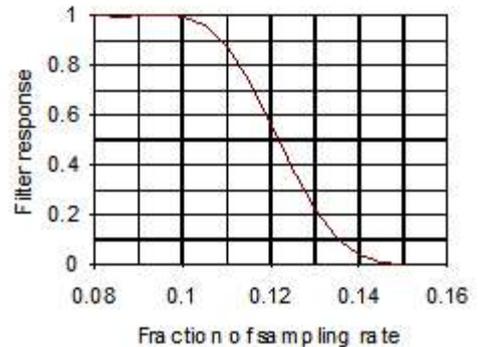
# High pass filter

A high pass filter is the same idea as a low pass except that the first frequency band is a stop band and the second band is a pass band. All the discussion for a low pass filter applies, with the addition that **there must be an odd number of coefficients**. If you try to use an even number your filter will be very poor indeed. The example below shows a script for a high pass filter with the same bands and tolerances as for the low pass filter. We have added a little more code to draw the frequency response in a result view.

```
var prm[5][2];
var coef[69];
'    band start        band end        function          weight
prm[0][0]:=0.00; prm[1][0]:=0.1; prm[2][0]:=0.0; prm[3][0]:=10.0;
prm[0][1]:=0.15; prm[1][1]:=0.5; prm[2][1]:=1.0; prm[3][1]:= 1.0;
FIRMake(1, prm[][], coef[]);
const bins% := 1000;
var fr[bins%];
FIRResponse(fr[], coef[], 0);
SetResult(bins%, 0.5/(bins%-1), 0, "Fr Resp", "Fr","dB");
ArrConst([], fr[]);
Optimise(0);
WindowVisible(1);
```

### Effect of odd and even coefficients
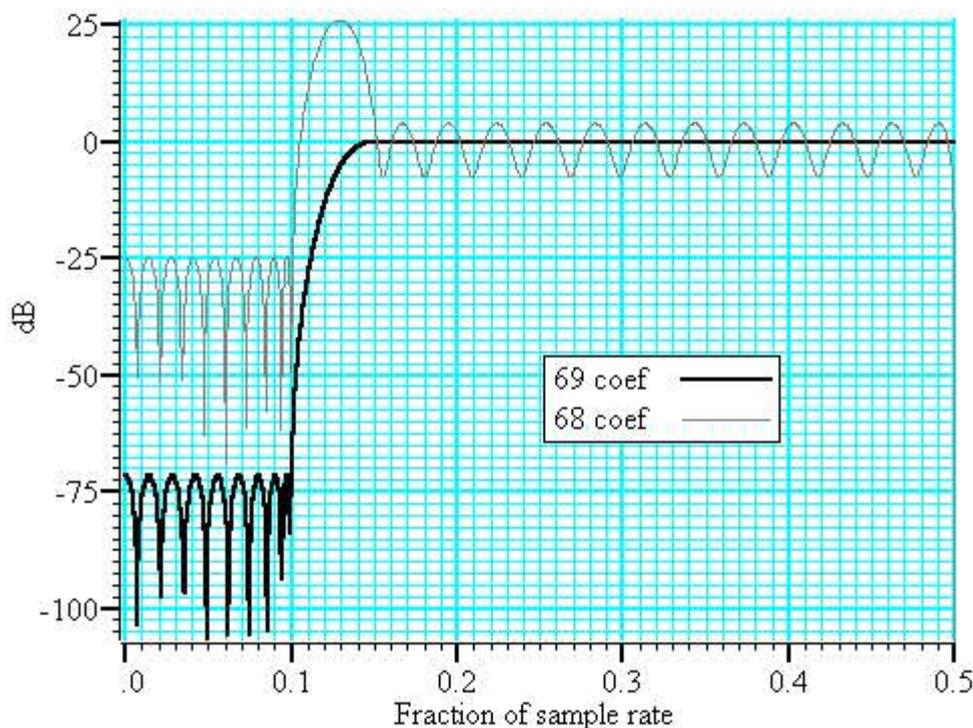


The graph shows the results of this high pass filter design with 69 coefficients, which gives a good result, and with 68 coefficients, which does not. In fact, if we had not given a factor of 10 weight (20 dB) to the stop band, the filter with 68 coefficients would not have achieved any cut in the stop band at all!

The reason for this unexpected result is that we have specified a non-zero response at the Nyquist frequency (half the sampling rate). If you imagine a sine wave with a frequency of half the sample rate, each cycle will contribute two samples. The samples will be 180° out of phase, so if one sample has amplitude *a*, the next will have amplitude -*a*, the next *a* and so on. The filter coefficients are mirror symmetrical about the centre point for a band pass filter, so with an even number of coefficients, the result when the input waveform is *a*, -*a*, *a*, -*a*... is 0. Another way of looking at this is to consider that a filter with an even number of coefficients produces half a sample delay. The output halfway between points that are alternately +*a* and -*a* must be 0.

You can use the formula given for the low pass filter to estimate the number of coefficients required, but you must round the result up to the next odd number.

## General multiband filter

You can define up to 10 bands. However, it is unusual to need more than three. The most common cases with three bands are called band pass and band stop filters. In a band pass filter, you set a range of frequencies in which you want the signal passed unchanged, and set the frequency region below and above the band to pass zero. In a band stop filter you define a range to pass zero, and set the frequency ranges above and below to pass 1.

You must still allow transition bands between the defined bands, exactly as for the low and high pass filters, the only difference is that now you need two transition bands, not one. Also, if you want a non-zero response at the Nyquist frequency, you must have an odd number of coefficients.

For our example we will take the case of a signal sampled at 250 Hz. We want a filter that passes from 20 to 40 Hz (0.08 to 0.16) with transition regions of 7.5 Hz (0.03). If we say it is 10 times more important to have no signal in the stop band than ripple in the pass band, and we want 70 dB cut in the stop band we will get 50 dB ripple in the pass band (because a factor of 10 is 20 dB). To use the formula for the number of coefficients we need the mean attenuation/ripple in dB and the width of the transition region. The two stop bands have an attenuation of 70 dB and the pass band has a ripple of 50 dB, so the mean value is (70+50+70)/3 or 63.33 dB. We have two transition regions (both the same width). In the general case of transition regions of different sizes, use the smallest transition region in the formula. Plugging these values into the formula predicts 113 coefficients, however only 111 are needed to achieve 70 dB.

```
var prm[5][3];                ' 3 bands for band pass
var coef[111];                ' 111 coefficients needed
'    band start        band end        function        weight
prm[0][0]:=0.00; prm[1][0]:=0.05; prm[2][0]:=0.0; prm[3][0]:=10.0;
prm[0][1]:=0.08; prm[1][1]:=0.16; prm[2][1]:=1.0; prm[3][1]:= 1.0;
prm[0][2]:=0.19; prm[1][2]:=0.50; prm[2][2]:=0.0; prm[3][2]:=10.0;
FIRMake(1, prm[][], coef[]);
```

**Band pass filter with 111 coefficients**
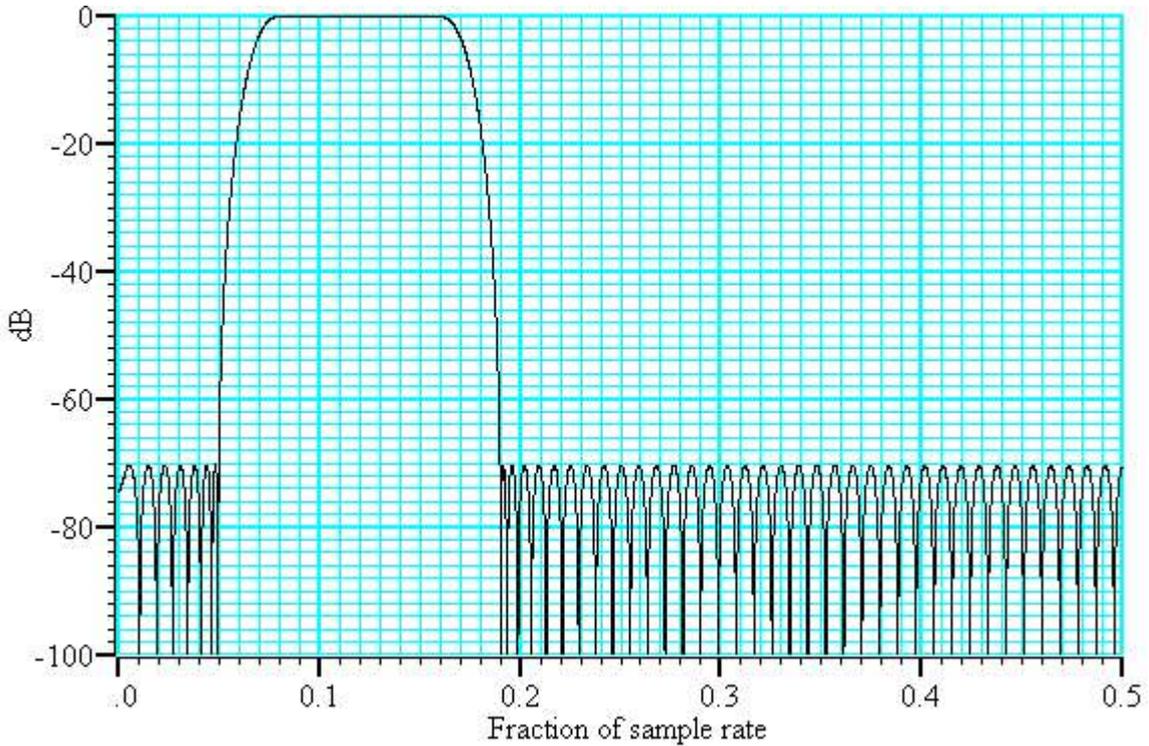


# Differentiators

A differentiator filter has a gain that increases linearly with frequency over the frequency band for which it is defined. There is also a phase change of 90 degrees (pi/2)]] between the input and the output.

### Ideal differentiator with slope of 2.0

You define the differentiator by the number of coefficients, the frequency range of the band to differentiate and the slope. The example above has a slope of 2. Within each band (normally only 1 band is set) the program optimises the filter so that the amplitude of the ripple (error) is proportional to the response amplitude. A differentiator is normally defined to operate over a frequency band from zero up to some frequency f. If f is 0.5, or close to it, you must use an even number of coefficients, or the result is very poor. You can estimate the number of coefficients required with the following function:

```
' dB     the proportional ripple expressed in dB
' f      the highest frequency set in the band
' even% Non-zero if you want an even number of coefficients
func NCoefDiff(dB, f, even%)
if (f<0) or (f>0.5) then return 0 endif;
f := 0.5-f;
var n%;
if (even%) then
   n% := (dB+43.837*f-35.547)/(0.22495+29.312*f);
   n% := (n%+1) band -2;    'next even number
else
   if f=0.0 then return 0 endif;
   n% := dB/(29.33*f);
   n% := n% bor 1;          'next odd number
endif;
return n%;
end
```

For an even number of coefficients this is unreliable when `f` is close to 0.5. For an odd number, no value of `n` works if `f` is close to 0.5.

These equations were obtained by curve fitting and should only be used as a guide. To make a differentiator that uses a small number of coefficients, use an even number of coefficients and don't try to span the entire frequency range. If you cannot tolerate the half point shift produced by using an even number of coefficients and must use an odd number, you must set a band that stops short of the 0.5 point. Remember, that by not specifying the remainder of the band you have no control over the effect of the filter in the unspecified region. However, for an odd number of points, the gain at the 0.5 point will be 0 whatever you specify for the frequency band.

The graph below shows the effect of setting an odd number of coefficients when generating a differentiator that spans the full frequency range. The second curve shows the improvement when the maximum frequency is reduced to 0.45.

**Differentiators with 31 coefficients**
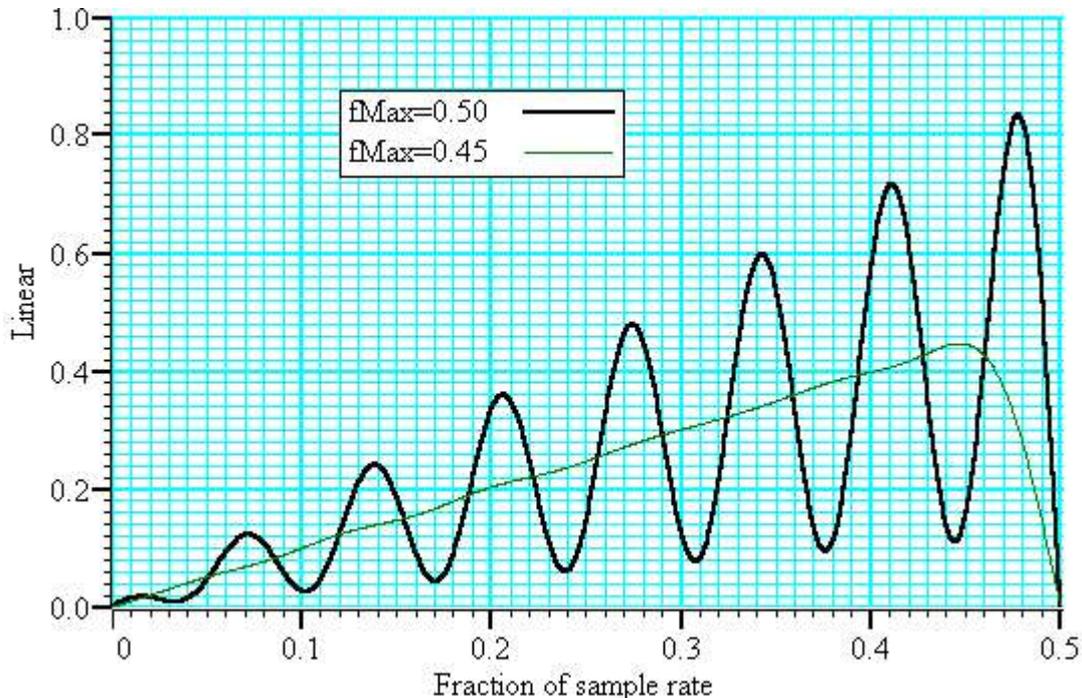


If you must span the full range, use an even number of coefficients. The graph below shows the improvement you get with an even number of coefficients. The ripple for the 0.45 case is about the same with 10 coefficients as for 31.

**Differentiators with 10 coefficients**
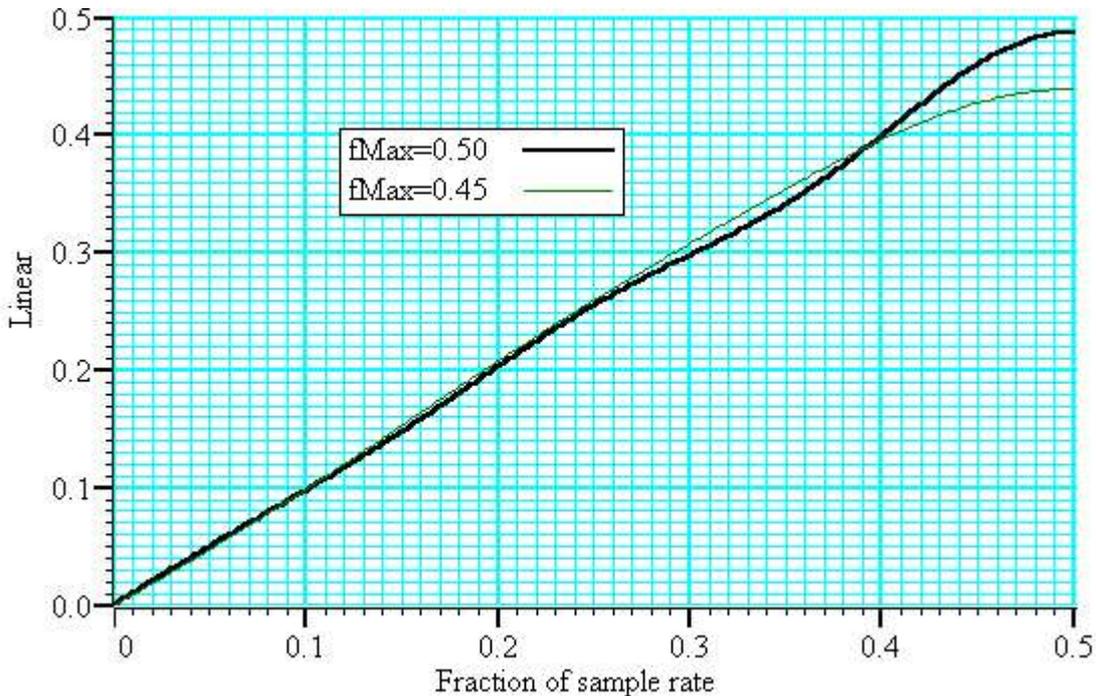


```
var prm[4][1];        ' 1 bands for differentiator
var coef[10];         ' 10 coefficients needed
'    band start         band end           slope           weight
prm[0][0]:=0.00; prm[1][0]:=0.45; prm[2][0]:=1.0; prm[3][0]:=1.0;
FIRMake(2, prm[][], coef[]);
```

# Hilbert transformer

A Hilbert transformer phase shifts a band of frequencies from $F_{low}$ to $F_{high}$ by pi/2. The target magnitude response in the band is to leave the magnitude unchanged. $F_{low}$ must be greater than 0 and for the minimum magnitude overshoot in the undefined regions, $F_{high}$ should be $0.5-F_{low}$. The magnitude response at 0 is 0, and if an odd number of coefficients is set, then the response at 0.5 is also 0. This means that if you want $F_{high}$ to be 0.5 (or near to it), you must use an even number of coefficients.

There is a special case of the transformer where there is an odd number of coefficients and $F_{high} = 0.5-F_{low}$. In this case, every other coefficient is 0. This is no help to Signal, but users who write their own software can use this fact to minimise the number of operations required to make a filter.

It is extremely unlikely that a Hilbert transformer will be of any practical use in the context of Signal, so we do not consider them further. You can find more information about this type of filter in *Theory and Application of Digital Signal Processing* by Rabiner and Gold.

# Programmable Signal conditioners

The Signal conditioner control panel supports the CED 1902 Mk III and IV, the Axon Instruments CyberAmp and the Digitimer D360 and D440 signal conditioners and also the Power1401 gain option. Only one type of signal conditioner can be used with Signal at any given time, the type of signal conditioner to be used (or none) is selected during Signal installation, this choice can be changed using the Conditioner page in the Preferences dialog. Note that the Power1401 gain option is not necessary for general-purpose use of the Power1401; it makes use of the gain control hardware that may optionally be fitted (at the time of manufacture) into Power1401s.

You can open the conditioner control panel from either the sampling configuration port configuration page or from the Sample menu. The 1902 and CyberAmp signal conditioners are controlled through a serial port which is set in the Edit menu Preferences… option.

## What a signal conditioner does

A signal conditioner takes an input signal and amplifies, shifts and filters it so that the data acquisition unit can sample it effectively. Many input signals from experimental equipment are too small, or are masked by high and or low frequency noise, or are not voltages and cannot be connected directly to the 1401.

Signal conditioners may also have specialist functions, for example converting transducer inputs into a useful signal, or providing mains notch filters. The CED 1902 has options for isolated inputs and specialised front ends include ECG with lead selection, magnetic stimulation artefact clamps and EMG rectification and filtering. With the Power1401 with programmable gains option only the gain may be set from within Signal.

You should consult the documentation supplied with your signal conditioner to determine the full range of its capabilities.

## Serial ports

Most signal conditioners use a serial line connection for software control. The serial port, if any, used to communicate with signal conditioners is set in the Conditioner page of the Preferences dialog. In addition to using serial line ports built into your computer you can also make use of serial lines provided by plugging an adaptor into a USB socket. USB serial lines generally work fine but you do need one which always appears as the same COM port, or where you can set the COM port as which it appears. To aid you in getting this correct, the COM port selector will only show ports that actually exist on the PC.

This method of controlling the serial line used does not apply to the Digitimer D360 and D440, where the support software determines the serial line in use independently.

Check the Dump errors to Log Window check box if you are having problems using your conditioners; this will cause log text detailing communications attempts to be added to the Signal log window which will allow the cause of connection problems to be determined. Leave this check box clear if you are not having any problems connecting to signal conditioners – it's intended for troubleshooting only.

# Control panel

The signal conditioner control panel is in two halves. The left-hand side holds the controls that change the conditioner settings, the right-hand side displays data from the conditioner. Signal omits the right-hand side while sampling data or if the 1401 interface is not available for any reason.

If the right-hand side is present, the Volts check box causes the data to be displayed in Volts at the conditioner input in place of user units as defined by the Channel parameters dialog. The number at the bottom right is the mean level of the signal in the area marked above the number.
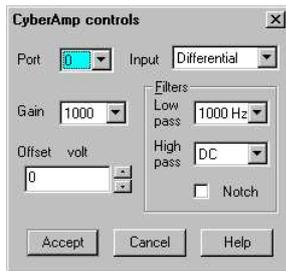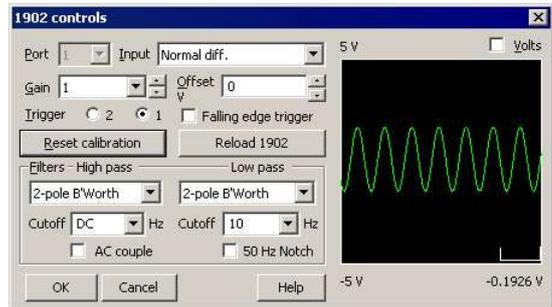
Signal conditioners differ in their capabilities and therefore not all the controls described below will appear for all conditioners. The example on the left is the CyberAmp control panel (with the signal display part omitted). The controls are:

### Port

This is the 1401 ADC port number whose conditioner settings are being adjusted. Only ports for which a conditioner was found are shown.

### Input

If your signal conditioner has a choice of input options, you can select the input to use with this field. The choice of input may also affect the ranges of the other options. Note that with the 1902 amplifier some of the inputs select a grounded input that is useful for testing but will only produce a flat line at zero when used. Signal will warn you if you try to use a grounded 1902 input for sampling, if you get this message you should change the input selection to either one of the other transducer inputs or an isolated input depending upon the input connections that you are using and your experimental requirements.

### Gain

This dialog field sets the gain to apply to the signal selected by the Input field. Signal tracks changes of gain (and offset) and will change the channel scaling factors in the ports configuration to preserve the y axis scale. To make the best use of the accuracy of the 1401 family, you should adjust the gain of your signal so that the maximum input signal does not exceed the limits of the data displayed on the right of the control panel.

### Offset

Some signals are biased away from zero and must be offset back to zero before they can be amplified. If you are not interested in the mean level of your signal, only in the fluctuations, you may find it much simpler to AC couple (1902) or high-pass filter (CyberAmp) the signal and leave the offset at zero.

### Low pass filter

A low-pass filter reduces high-frequency content in your signal. Filters are usually specified in terms of a corner frequency, which is the frequency at which they attenuate the power in the signal by a factor of two and a slope, which is how much they increase the attenuation for each doubling of frequency. Sampling theory tells us that you must sample a signal at a rate that is as least twice the highest frequency component present in the data. If you do not, the result may appear to contain signals at unexpected frequencies due to an effect called aliasing. As the highest frequency present will be above the corner frequency you should sample a channel at several times the filter corner frequency (probably between 3 and 10 times depending on the signal and the application).

You can choose a range of filter corner frequencies, or you can choose to have the data unfiltered (for use when the signal is already filtered due to the source).

If you are using a modern 1902 that uses digital filtering, you can set the low-pass filter type (from 2-pole and 3-pole Butterworth and 2-pole and 3-pole Bessel) as well as set the filter corner frequency.

### High pass filter

A high-pass filter reduces low-frequency components of the input signal. The high-pass filters area is specified in the same way as low-pass filters in terms of a corner frequency and a slope, except that the slope is the attenuation increase for each halving of frequency. If you set a high-pass filter, a change in the mean level of the signal will

cause a temporary change in the output, but the output will return to zero again after a time that depends on the corner frequency of the filter. The lower the corner frequency, the longer it takes for mean level change to decay to zero.

Again, if you are using a modern 1902 that uses digital filtering, you can set the high-pass filter type (from 2-pole and 3-pole Butterworth and 2-pole and 3-pole Bessel) as well as set the filter corner frequency.

### Notch filter
A notch filter is designed to remove a single frequency, usually set to the local mains power supply (50 Hz or 60 Hz, depending on country).

### Reset calibration
The Reset Calibration button resets the port calibration information in Signal to show the input voltage taking into account the current signal conditioner gain and offset, it does not make any changes to the conditioner settings. The units for the calibration will be set to V, mV or uV as appropriate (based on the gain in use) and the port Full and Zero values within Signal are adjusted as appropriate. On the CyberAmp, this option will use the 'native' calibration information and units specified by a SmartProbe, if present.

*The remaining options are available for the CED 1902 only:*

### AC couple
This is present for the 1902 only, and can be thought of as a high-pass filter with a corner frequency of 0.16 Hz. However, it differs from the high-pass filters as it is applied to the signal at the input; the high-pass filters in the 1902 are applied at the output.

### Trigger
The 1902 provides two conditioned trigger inputs, and one output. These controls select which of the trigger inputs is used to generate the output and, in a 1902 mk IV, allow selection of the trigger input polarity.

### Filter type
A 1902 mk IV with digital filters has extra fields that allow you to set the low-pass and high-pass filter types. You can choose between Butterworth or Bessel characteristics and 2 pole or 3 pole filters.

### Reload 1902
The 1902 is controlled by a serial interface and to set the complete state of it would takes a noticeable time. To avoid this, we keep a record of the state that has been commanded and only transmit changes. If you click this button, we transmit all the commands to set every feature of the 1902 to match the control panel. This is provided so that you can recover a 1902 that has been powered down by accident.

## Setting the channel gain and offset

If you change the gain or offset in the control panel, Signal will adjust the port Full and Zero values in the sampling configuration to compensate so as to keep the y axis showing the correct values. This means that if you change the gain, the signals will still be correctly calibrated in the file. However, the first time you calibrate the channel you must tell the system how to scale the signal into y axis units.

For example, to set up the y axis scales in microvolts you do the following:
1. Open the Sampling Configuration dialog.
2. Select the ADC port in the Ports setup page.
3. Press the Conditioner button to open the conditioner control panel.
4. Adjust the gain to give a reasonable signal. Make a note of the gain $G$ you have set.
5. Close the signal conditioner control panel.
6. Set the Units field of the Channel parameters to uV.
7. Set the Full field to $5000000/G$.

You only need do steps 6 and 7 once. Any subsequent change to the conditioner gain will adjust the channel Full value to leave the units in microvolts.

For the more general case imagine you have a transducer that measures some physical quantity (Newtons, for example) and it has an output of 152.5 Newtons per V. If you wanted the y axis scaled in Newtons, you would replace steps 6 and 7 above with:

8. Set the Units field of the Channel parameters dialog to N.
9. Set the Full field to (5 * 152.5)/*G*.

To work this out you must express the transducer calibration in terms of Units per Volt (in this case Newtons per Volt), multiply this by 5 to get the Full value at five volts, then divide it by the gain of the conditioner.

If you have set an offset in the conditioner, and you want to preserve the mean signal level, you should null it out by changing the offset in the Channel parameters dialog.

# Conditioner connections and details

Signal normally expects the first channel of signal conditioning to be connected to 1401 ADC port 0, the second to port 1 and so on. Connect the conditioner output BNC (labelled Amp Out on the 1902, and OUT on the CyberAmp) to the relevant 1401 input. This arrangement can be adjusted to start with another port, but in all cases the conditioner channel number must match the ADC port to which it is connected.

Most signal conditioners connect to the computer with a serial line. You will have received a suitable cable with the unit.

Signal conditioner settings are stored in the Windows registry in location HKEY_CURRENT_USER\Software\CED\CEDCond\XXXX where XXXX is the name of the type of conditioner you are using; it can be one of 1902, CyberAmp, Power1401, D360 or D440. This registry location holds a number of DWORD values each of which has a different name, they are documented here for overall completeness but should be viewed and changed using the controls in the Conditioner page in the Preferences dialog. The registry values are :

### Port

The `Port` value sets the communications port to use for 1902 and CyberAmp signal conditioners. We would recommend you use the Edit menu Preferences… to change the port number or it can be set for 1902s by using Try1902. If this value is missing, COM1 is used.

### Dump

The Dump value enables diagnostic logging. If this value is set to 1 (it's easiest to control it from the Preferences dialog) Signal will generate information in the log window when it initialises the conditioner, this can be used to diagnose communications problems. If you are using 1902 signal conditioners it is recommended that you use the Try1902 application (which can be downloaded from the CED web site) to diagnose communications problems.

As mentioned above, the first signal conditioner is usually connected to ADC port 0, the second to port 1, and so on. Signal searches for signal conditioners based on this assumption. The search starts at signal conditioner channel 0 and continues until a channel does not respond. The search sequence can be changed by two additional items in the registry:

### First & Last

These items (it's easiest to control them from the Preferences dialog) are used to optimise the search for available conditioner channels, both to ensure that the channels wanted are found and to prevent excessive time being spent on the search. If these values are not set, they are both assumed to be 0. Their use depends on the signal conditioner.

### CED 1902

CED 1902s are controlled using a serial line, the COM port used is set by the `Port` registry value. The amplifiers have unit numbers set by an internal switch pack; multiple units usually have the channel number as a label on the front panel. If you have multiple units, they must have different unit numbers. Each 1902 output should be connected to the 1401 ADC input with the same number as the 1902 unit. `First` and `Last` control the range of unit numbers to search, the search starts at `First` and goes at least up to `Last`, it continues beyond `Last` until a unit does not respond. The purpose of `Last` is to force the search to skip over missing conditioners. The purpose of `First` is to skip over missing lower-numbered conditioners to avoid time delays waiting for them to respond.

Normally you will have 1902s with consecutive unit numbers starting at 0, in which case you do not need to set `First` and `Last`. As an example of a more complicated situation, let us suppose you have unit numbers 4, 5, 6, 7, 12, 13, 14 and 15. In this case, you should set `First` to 4 and `Last` to 15.

**CyberAmp**

CyberAmps are controlled using a serial line, the COM port used is set by the `Port` registry value. CyberAmps have a device Address set by a rotary switch on the rear panel, if you have multiple units, they must have different addresses. There are two types of CyberAmp: 8-channel and 2-channel. If you have multiple units, either they must all have the same number of channels or all of the 8-channel units must have lower device addresses than the 2-channel units.

`First` and `Last` set the range of device addresses to search. The search continues beyond `Last` until a device does not respond. The purpose of `Last` is to force the search to skip over missing devices. The purpose of `First` is to skip over missing lower-numbered devices to avoid time delays waiting for them to respond.

The range of ADC channels supported by each CyberAmp is set by the device address (`dev`) and the number of channels in the first device detected (`num`). The first ADC channel supported by the device at address `dev` is `dev*num`. Each CyberAmp support 8 or 2 consecutive ADC channels.

Normally you will have one 8-channel CyberAmp, and you will give it address 0 to support ADC channels 0 to 7. In this case you do not need to set `First` and `Last`. If you want to connect it to channels 8-15, you would set both First and the device address to 1 and then you could connect it to channels 8-15.

Here is a more complex example with three CyberAmps, two with 8-channels and one with 2-channels. The table shows some possible configurations (assuming you have 32 ADC inputs to choose from):

| CEDCOND.INI | | 8-channel unit 1 | | 8-channel unit 2 | | 2-channel unit 3 | |
|---|---|---|---|---|---|---|---|
| First | Last | Switch | ADC | Switch | ADC | Switch | ADC |
| 0 | 2 | 0 | 0-7 | 1 | 8-15 | 2 | 16-17 |
| 1 | 3 | 1 | 8-15 | 2 | 16-23 | 3 | 24-25 |
| 2 | 3 | 2 | 16-23 | 3 | 24-31 | - | - |

**CED Power1401 gain**

The Power1401 data acquisition hardware from CED can optionally be fitted with programmable gains on its ADC ports - note that this option should be ordered at the time you purchase your 1401 - it is expensive or impossible to add programmable gains afterwards. Signal can detect 1401 channels with gain fitted automatically so the `Port`, `First` and `Last` registry values are all unused.

**Digitimer D360**

The Digitimer D360 is also controlled using a serial line but Signal does not know of or control the COM port number that is used - that is managed by the separate D360 support software. Similarly the `First` and `Last` registry values are not used. You should install the D360 support separately and check that it is working correctly using the Digitimer-supplied control software before using it with Signal. We have found that Signal can have problems if you are running older versions of the Digitimer D360 support software, for example it might fail to detect a connected D360 amplifier. If you do encounter problems using the D360 with Signal you should check if your D360 client software is at least version 4.7.0.0 and upgrade to the latest version if it is older than this. Digitimer D360 support is currently only available with the 32-bit build of Signal as the necessary Digitimer software is restricted to 32-bit use.

Detection of a missing D360 takes some time so you may on occasion experience long delays when opening the sampling configuration, opening the conditioner preferences, starting sampling or shutting down Signal if you have installed D360 support but the D360 is not connected or switched on. Detection of connected D360s is only done once, when Signal starts up, so if your amplifier was off at that point you will have to restart Signal in order to use the amplifier.

Multiple D360 amplifiers can be used; each has 8 channels numbered 1 to 8. Channels 1 to 8 of the first D360 are assumed to be connected to ADC inputs number 0 to 7, the second D360 drives ADC ports 8 to 15 and so on. The amplifier numbers are determined by the order in which they are daisy-chained together, they should be connected in serial-number order with the lowest-numbered amplifier connected directly to the PC, the next lowest-numbered amplifier connected to the first amplifier and so forth.

**Digitimer D440**

Install the Digitimer-supplied D440 support software first and verify that it is working by running the Digitimer control panel. Signal should then be able to locate and use the device. The Digitimer software does not (yet) support multiple D440 units.

# Amplifier telegraphs

If you are using an external amplifier to scale your signals before they are sampled by the 1401, you can encounter problems if you change the signal gain while sampling – the amplitude of the sampled data changes and this makes the signal calibrations incorrect. Signal can avoid this problem by using amplifier telegraphs to determine the current amplifier settings. By collecting and interpreting amplifier telegraphs, Signal can automatically adjust channel scalings to compensate for changes in the amplifier settings and maintain signal calibration.

Signal currently supports two types of telegraph signal; a standard gain-only telegraph using one or more analogue signals that are sampled using the 1401 interface and an auxiliary amplifier specific mechanism provided by DLLs installed with Signal. The standard 1401 gain-only telegraphs are always available for use, while the auxiliary telegraph amplifier support is only available if you have chosen to install it when you installed Signal or if you afterwards selected an auxiliary telegraph amplifier using the Clamp page in the Preferences dialog. Only one type of auxiliary telegraph support can be used by Signal at any given time. Currently, three types of auxiliary telegraph amplifier are supported; the Molecular Devices (Axon Instruments) MultiClamp 700 range of amplifiers, the AxoClamp 900A amplifier and the Heka EPC 800 amplifier. If you do not install or select support for an auxiliary telegraph amplifier then only standard 1401 telegraphs will be available.

The support for clamping experiments that is available within Signal is able to make use of the much more complex information available from the auxiliary telegraph systems to automatically configure channels and clamping configurations.

## Standard 1401 telegraphs

A number of different amplifiers, for example the Axoclamp 200, are able to signal the amplifier gain currently in use by generating a 'telegraph' output; a voltage which varied according to the gain currently selected. Originally, a researcher would sample this telegraph output along with all the others and use its level to determine the gain in use and adjust measured values accordingly. When using Signal these amplifier telegraph outputs can be handled automatically using standard 1401 telegraphs.

The standard telegraph support in Signal uses spare 1401 ADC inputs to sample the telegraph outputs; the analogue voltages generated by the amplifier. A lookup table is then used to convert these voltages into the amplifier gain setting which is then used to adjust the signal calibration as necessary. Up to four separate telegraph signals can be defined, each indicating the gain in use for a separate ADC input. Setting up standard 1401 telegraphs consists of defining the ADC ports involved and setting up the table used to convert telegraph voltages to amplifier gains. Because the 1401 is used to sample the telegraph signals in the period between frames, standard 1401 telegraphs cannot be used with any of the fast trigger sampling modes, including gap-free sampling.

## 1401 telegraph configuration

The 1401 telegraph configuration dialog is used to enter the settings for standard 1401 telegraphs. Using the dialog you can create up to four independent telegraph sets, each of which defines an ADC port whose scaling depends upon the amplifier telegraph output, a port from which the telegraph signal is read and a list of telegraph voltages and the corresponding amplifier gains.

To begin setting up amplifier telegraph support, select one of the four telegraph sets using the Telegraph set field.

Then set the ADC ports that are involved. The Scaled item sets the ADC port whose scaling is controlled by a telegraph signal, this is set to None if use of this telegraph set is disabled. The Telegraph item sets the ADC port from which Signal will read the amplifier telegraph output. It should be clear that this is equivalent to telling Signal which 1401 ADC inputs are connected to the two relevant amplifier outputs.
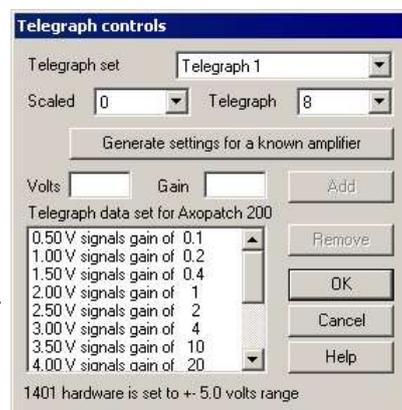
You then have to set up the table of telegraph voltages and corresponding amplifier gains. The button labelled 'Generate settings for a known amplifier', when pressed, will provide a list of common amplifiers which provide telegraph signals. Selecting one of the amplifiers from the list will generate a table of telegraph data

suitable for that amplifier. If your amplifier is not in the list please contact CED and we will try to add it – our thanks to John Dempster for providing much initial amplifier information.

Alternatively, you can use the Volts and Gain fields to directly enter a list of telegraph signal voltages and corresponding amplifier gains. For each amplifier telegraph setting enter the telegraph output voltage value and the corresponding gain value, then press Add to add this pair of values to the table. Note that the Add button will be disabled unless both the Volts and Gain fields are valid - if they are not valid a message indicating what is wrong will be displayed at the bottom of the dialog. To remove a pair of values from the list simply select them by clicking with the mouse and then press Remove.

The 1401 voltage range in use is shown at the bottom of the dialog, the telegraph Volts values entered cannot exceed this range.

## Telegraphs and Ports information
The telegraphs in use are indicated in the port options shown in the Ports page of the Signal sampling configuration dialog. Where a port scaling is being altered or controlled by an amplifier telegraph, the port options will show Tel (or ATel if controlled by an auxiliary telegraph) in red to indicate this fact. If the port is being used by the standard 1401 telegraphs to determine the gain for another port this field shows T>n in red, where n is the number of the port which is affected by this telegraph signal.

The 1401 telegraph system completely ignores the Full and Zero port calibration values defined for the ADC port used to read the amplifier telegraph outputs, it simply converts the ADC data to voltages using the 1401 ADC voltage range that in use.

The scaled port (the port whose scaling is controlled by the telegraph output) should be set up using Full and Zero port calibration values as if the amplifier **gain was set to unity**; the amplifier gains generated by the telegraph system will then be applied to this 'basic' calibration. It is very important that you enter the value for unity amplifier gain, not whatever gain you happen to have in use at the moment. During sampling, the telegraph signals are read at the start and end of each sampling sweep and used to adjust the signal scaling.

## A simple worked example
Let us suppose that we are using a single amplifier to generate our data and we wish to have this data automatically calibrated for us. The scaled output (the output which changes when the gain is changed) of the amplifier is connected to the 1401 ADC port 1, while the telegraph output is connected to 1401 ADC port 3. The amplifier has five gain settings: 10mV/mV, 30mV/mV, 100 mV/mV, 300 mV/mV and 1000mV/mV, the telegraph output voltage for these settings are 0, 1, 2, 3 and 4 volts respectively.

We start by selecting Telegraph set 1 and setting the Scaled port to 1, and the Telegraph port to 3. Now, the gain of an amplifier is the ratio of the output signal to the input signal, so the five amplifier gains are 10, 30, 100, 30 and 1000. To enter the first telegraph data pair we enter 0 in the Volts field and 10 in the Gain field and press Add. We repeat this for the other four pairs of values, at the end the table of telegraph data should show:

0.00 V signals gain of 10
1.00 V signals gain of 30
2.00 V signals gain of 100
3.00 V signals gain of 300
4.00 V signals gain of 1000

Finally, we have to go to the Ports page of the sampling configuration and make sure the calibrations are correct. We don't care about ADC port 3 as its calibration will be ignored but we want to set up the calibration for port 1 as if the gain was 1. So (assuming a 5 Volt 1401) you would set the Zero value for port 1 to 0 and the Full value to 5000 if we wanted to calibrate the scaled signal in millivolts and 5 if we wanted to calibrate in Volts. Note that it does not matter at all that the amplifier cannot give a gain of 1, we simply calibrate the ADC port as if there was a gain of 1 and let the telegraph arithmetic do the rest for us.

And that is all; when sampling, the calibration of the data sampled from ADC port 1 will always be adjusted to match whatever gain was set on the amplifier.

## A slightly less simple example
Suppose now that this amplifier can also measure current and generate an output voltage proportional to the current that flows through an electrode. The 1401 ports used remain the same as above, but the five gain settings are now 1mV/pA, 3mV/pA, 10mV/pA, 30 mV/pA and 100 mV/pA. The telegraph output voltages are also the same but how do we calculate the gains?

Well to be precise, we cannot, as an amplifier gain is voltage out/voltage in, and we do not have a voltage in value, only a current. But we can work round this easily enough by arbitrarily deciding that one of these settings represents a 'gain' of 1 and calculating the other gains relative to that setting. So we make the obvious choice and say that 1mV/pA is a 'gain' of 1, then the second setting of 3mV/pA represents a gain of 3 because we get three times the output for the same input. Similarly the other gains are 10, 30 and 100. So we enter these 'gain' values into the telegraph data and end up with a table like this:

0.00 V signals gain of 1
1.00 V signals gain of 3
2.00 V signals gain of 10
3.00 V signals gain of 30
4.00 V signals gain of 100

We then have to go to the Ports page to enter the signal calibration data. Again, we can ignore ADC port 3 but we have to calibrate port 1 as if the amplifier gain was 1. We have decided that a gain of 1 means 1 mV per pA so with a 5 Volt 1401 the full scale value is 5000 millivolts which corresponds to 5000 pA. So if we want to calibrate our current signal in pA we would enter a Full value of 5000, or if we preferred nA we would enter a Full value of 5. In either case the Zero value should be set to 0.

# MultiClamp 700 telegraphs

The MultiClamp 700 amplifiers supplied by Axon Instruments have two identical sections (referred to as channels) within them, each of which is controllable using a DAC driving the external command input and which has two signal outputs (giving four separate signals that can be sampled per amplifier). The MultiClamp commander software controlling the amplifier generates telegraph information telling interested applications about changes to amplifier gain settings and other settings such as the voltage or current clamp mode and external command sensitivity. Using the MultiClamp 700 telegraph configuration dialog you can configure Signal so that it can receive and make use of this telegraph information from up to four amplifiers, this information is of course only available when the the MultiClamp commander software is in use.

In addition to signalling changes to amplifier gains, output signals and operating modes, the MC700 control software also provides information about other aspects of the amplifier setup. For each channel in the amplifier, Signal can retrieve:

- The primary output's low-pass filter settings
- The secondary output's low pass filter settings
- The membrane capacitance
- The series resistance
- The external command sensitivity

All of these values for each channel in each amplifier are retrieved for every frame of data and placed into extra frame variables in the sampled data file. The frame variable values can be viewed using the File information dialog, plotted in a trend plot or retrieved using the script language.

*Integration with Signal channel scaling*
The basic MC700 support uses the information provided by the MC700 control software to adjust ADC port scaling factors to match the amplifier gains in the same way as for 1401-based telegraphs. The MC700 support is also able to set the initial port scale factors, name and units to match the MC700 settings, ensuring that all settings for ports sampled from MC700 outputs are correct.

*Integration with Signal clamping support*
Extra MC700 support features integrate the Signal support for clamping experiments with the MC700 amplifier setup. This additional mechanism reads back clamping set definitions (pairs of MC700 outputs that are in use and the associated external control DAC), selects between voltage-clamp and current-clamp according to the amplifier controls and in addition sets up the scaling and units of the DAC used for external amplifier control. This ensures that all relevant controls and calibrations are correct at the start of a clamping experiment. A button available in the clamping configuration page can be used to read back all of the clamping, ADC port and DAC scaling information from the MC700 so that they can be inspected.

## Getting started with clamping experiments using the MultiClamp 700 support
Signal is a much more general-purpose program that many of the other applications used for voltage- and current-clamping experiments. While this does have advantages - for example Signal can carry out an extremely wide

range of analyses - it does make getting started with these types of experiment a bit harder, and the complicated nature of the interaction between an amplifier such as the MC700 and such general purpose software also makes things difficult. There is more information about using Signal for clamping experiments in the Sampling with Clamp support chapter, this short guide is aimed at helping you with aspects specifically related to the MC700 amplifier. Nearly all of the information here is also relevant to the interfaces to other amplifiers, specifically the AxoClamp 900A and the EPC800.

### Installing the amplifier interface

The telegraph interface to the MC700 or other telegraph devices is an optional part of Signal. You have to install the support for your amplifier before it will be available for use, if you have not installed the interface software then the Aux Telegraphs button at the bottom of the ports configuration page of the sampling configuration dialog will be greyed-out. To install the interface software if you have not done so, or to change the amplifier telegraph support to a different device, simply re-install Signal into the same directory making sure that your choice of auxiliary telegraph device is correct.

### Configuring the amplifier interface

The first thing you have to do is to use the MultiClamp 700 telegraphs configuration dialog to define the amplifier(s) that you are using and their connections with the 1401. You can set up information for up to four MC700 amplifiers. The configuration dialog does three things really: it tells Signal how to communicate with the MultiClamp commander software, it tells Signal what the connections are between the amplifier and the 1401, and it has options that control how and when Signal uses information from the amplifier.

The amplifier type and serial number at the top should be set to match your amplifier(s) - if they are not correct then communications with the MultiClamp commander software will not work. The Test button at the bottom of the configuration dialog allows you to check that communications with the amplifier are working correctly.

You must then use the middle part of the configuration dialog to set the 1401 ADC and DAC ports that are connected to the relevant amplifier outputs and external command inputs - the wiring that you are using between the amplifier and the 1401. These must be correct if you want Signal to make correct use of information about the amplifier settings. If it does not know what ADC input is connected to MC700 primary output 1, it will not know which port to calibrate using information about primary output 1, similarly with the DAC calibrations. Signal cannot check that you have entered this information correctly so you should take care to make sure it is correct. You will probably find out very quickly if the information is wrong because your data channels are incorrectly named, have the wrong units, or show the wrong signals.

The bottom of the dialog controls what information Signal automatically reads and applies at the start of sampling, normally you should have both of these checkboxes checked so that everything is set up for you (they are intended to be turned-off if you need to override the amplifier information in some way). It also allows you to set your preferred units for the control DAC signals, which allows you to set up your DAC control pulses beforehand while knowing that voltage pulses will use millivolt units, for example. Otherwise the amplifier information could change the units for you, making the pulses 1000 times too large or small...

### Checking the amplifier settings

You can then look at the information read back from the amplifier. Go to the Clamp page of the sampling configuration and (with the amplifier set up and the commander software running, and the amplifiers set in VC mode) and click on the MC700 button at bottom right. Select the top option - Read All. This will cause Signal to read back all the settings from the amplifier and try to use them to set itself up. So you should see the following:

- Up to 8 clamping sets generated using the current amplifier mode and the ADC ports and DACs you defined as being connected to the amplifier(s), note that while you define the amplifier connections using ADC and DAC port numbers (which relate to specific connections on the 1401), the clamping set information refers to channels in the data file that will be created by sampling, so the numbers seen will be different. Obviously, each amplifier can only generate two clamping sets.

- The DACs defined in the clamping sets will have their calibration and units in the Outputs page set up using the information read from the amplifier and the preferred units set in the configuration dialog.

- The ADC ports that are used to sample the channels defined in each clamping set (the ports you said were connected to the amplifier) will have their calibration, title and units in the Ports page set using information read from the amplifier.

- If there is anything about the information read back that Signal does not like, for example bad ADC port units, it will show that information in red in the clamping set information.

So, with the amplifier set in VC mode, you should be able to use the button to read all information and get everything set up as it should be with clamping sets in VC mode. If you see any information in red this will mean that the amplifier outputs set for VC mode are not as Signal expects them to be - it wants a current signal on the primary output and a voltage (membrane potential) signal on the secondary output, but in the amplifier settings in MultiClamp commander you can freely choose which signal is routed to the outputs. So if you see any information in red you should be able to correct things by changing the signals fed to the relevant outputs using MultiClamp commander and pressing the button to re-read all the clamping information.

Then (if you want to carry out current clamping experiments as well), switch the amplifier(s) into IC mode and then repeat the process to get the current clamp information and check that Signal reads it back and finds it satisfactory. Again, you may need to use MultiClamp commander to change the signals routed to the various amplifier outputs so they are as Signal expects. The MultiClamp commander software remembers the signals used in VC and IC modes so once you have them OK for Signal you will not need to change them again.

If you are unable to get the amplifier settings to read back in a way that Signal accepts, contact CED for assistance.

## Getting your sampling configurations correct

The MC700 system can automatically set up a lot of Signal sampling configuration information for you. Other bits you have to do for yourself by reading the relevant pages of this help file, the Sampling data, Sampling with clamp support, Pulse outputs while sampling and Sampling with multiple states chapters will probably all be relevant. At the very least you will need to:

- Use the General page to define the sampling rate that you want to use, the ADC ports you want to sample-from and the duration of each frame of data.
- Use the Outputs page to select pulse outputs. Sequencer-controlled outputs can be used in clamping experiments but hopefully you will be able to start of with the much simpler pulses mode.
- Use (very probably) the General page to enable multiple states sampling and set the multiple states mode to Dynamic outputs.

In addition there is one other thing that you will have to do - generate sets of suitable stimuli using the DACs connected to the amplifier external control inputs. You do this using the pulses configuration dialog available from the Outputs page of the sampling configuration. Multiple states sampling in dynamic outputs mode is very useful here as it allows you to have up to 256 separate sets of output stimuli in one sampling configuration and switch between them in various ways during the experiment.

## Using voltage clamp together with current clamp

It is often the case that users want to use both voltage clamping and current clamping on the same cell. While Signal can easily do this, it cannot do this within the same data file, the amplifier settings for the two modes are too different. So to switch from one mode to another it is necessary to stop sampling and save the new data file, change the amplifier mode, and start sampling again to a different data file. Note that the Automation section of the sampling configuration dialog provides automatic file name generation and data file saving facilities that help to make this quick.

It is generally simplest to have a separate sampling configurations for VC and IC experiments, at least at first, that way you can get all the outputs as needed without having to worry about the other type of experiment. It is possible to have one configuration that covers both modes, using separate sets of output states and pulses for each mode, but this is best left until you are experienced with the system.

So to run a dual-mode experiment you would probably do something like this:

- Set the amp into VC mode
- Load in the VC sampling configuration file
- Run it and collect data
- Stop sampling and save your data
- Set the amp in IC mode
- Load in the IC sampling configuration file
- Run it and collect data

Note that any changes you make to the amplifier gains will automatically be accounted-for, regardless of the information in the sampling configuration, so your signals will always be correctly calibrated.

Also, there is a sample tool-bar in Signal which you can set up with individual buttons for your key sampling configurations. Using the sample bar allows loading in a new sampling configuration file to be a one-click operation, crucial when you are busy trying to collect data quickly.

### Issues with 1401 signal limits

As normally shipped, the 1401 data acquisition interface ADCs and DACs only cover a ± 5 volt range. If you increase the amplifier gain too much, you may well find that they exceed this limit and are clipped, similarly with some external command sensitivities you may find that the 5 volt DAC output range does not allow the stimuli you need. In some circumstances and with some amplifiers these problems can be avoided by using low amplifier gains and high external command sensitivities, but it is normally simplest, with modern 1401s, to change the supported voltage range to ±10 volts. This can be done using the Try1401 program that is installed with Signal, use the File menu 1401 Options command.

# MultiClamp 700 telegraph configuration

Signal can receive telegraph information from up to four MultiClamp 700 amplifiers. The MultiClamp 700 telegraph configuration dialog is used to configure the Signal software so that it can communicate with the MultiClamp Commander software controlling each amplifier and to define how you want information from that amplifier to be used.

At the top of the dialog is a row of radio buttons selecting the amplifier that you want to configure. Below this is a check box used to select the type of amplifier and information used to identify the amplifier. Set this according to the type of amplifier you are using. For the 700A you will have to specify the COM port used to control the amplifier and the Axon bus ID for the amplifier, for the 700B you have to specify the serial number of the amplifier. If you do not want to make use of the MC700 telegraph system in this sampling configuration, set the bus ID for all amplifiers to Do not use (for the 700A) or set the serial number to −1 (for the 700B).
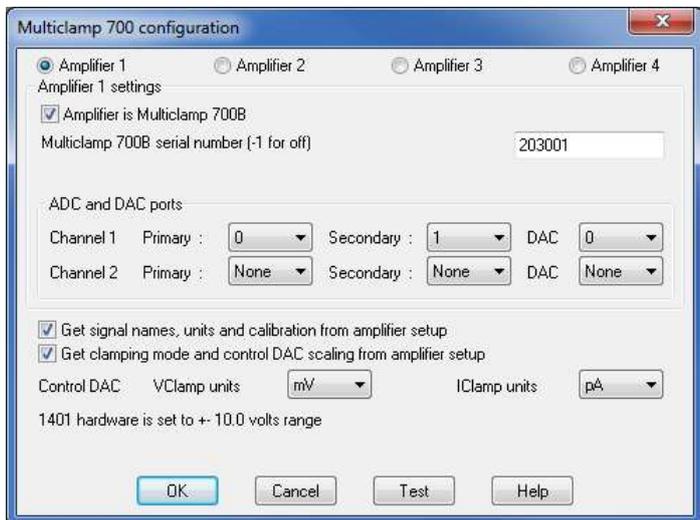
Below the amplifier identification section is an area defining the 1401 ADC ports and 1401 DACs that are connected to the amplifier. Each MC700 amplifier has two channels, each of which has two outputs (called Primary and Secondary for the 700B, Scaled and Raw for the 700A), plus an External command input. Set the ADC ports on the 1401 that are connected to the various amplifier outputs, leaving any amplifier outputs that are not being used set to None. Similarly, set the DAC port number that is connected to the External command input for that channel.

### Automatic signal calibration

The check box labeled 'Get signal names, units and calibration from amplifier setup' is used to allow automatic signal definition & calibration from the amplifier up. When set, ports connected to MC700 outputs are automatically set up using information retrieved from the amplifier. The information retrieved is the port name and units (depending upon what signal you have routed to the relevant MC700 output, current signals are always calibrated in pA) plus both of the port calibration values. This is the easiest way to use MC700 telegraph data - it is generally recommended as it will force channels sampled from MC700 outputs to be correct.

If this check box is left clear, then the name, units and calibration entered in the Signal port setup dialog (from the Port Setup page in the sampling configuration) are used. You can set the channel name and units to anything you wish, the port Zero value should be set to 0 and the Full scale value should be set to the value of a full-scale signal with the MC700 amplifier **gain set to unity**. It is very important that you enter the value for unity amplifier gain, not whatever gain you have in use at the moment. The full scale value is the value corresponding to a +5 volt (or +10 if appropriate) signal on the 1401 input.

### Automatic clamp setup

The check box labeled 'Get clamping mode and control DAC scaling from amplifier setup' is used to allow automatic setup of the Signal clamp system from the amplifier settings. When set (the check box for automatic signal calibration must also be set for this option to be available) Signal will define clamping sets at the start of sampling by using the amplifier settings read from the MC700.

Here, one clamping set corresponds to an MC700 channel. For an MC700 channel to define a clamping set, both the ADC ports and the DAC port must be set to something other than None and the ports must exist on your 1401. The port connected to the Primary channel output (Scaled output for a 700A) will be used for the Response channel in the clamping set. Similarly, the port connected to the Secondary channel output (Raw output for a 700A) will be used for the Stimulus channel, and the DAC number used as the control DAC. The clamping mode is set using information read back from the amplifier, the DAC scaling is set using the external command sensitivity information read from the amplifier, and the DAC units are set as defined using the Control DAC VClamp units and IClamp units fields in the configuration dialog.

With this option enabled you can even set up a single sampling configuration containing control DAC pulse sequences suitable for both voltage clamp and current clamp (knowing which units that the DAC will use in either mode) and can quickly switch between voltage and current clamp experiments by stopping sampling, switching the MC700 amplifier mode, and beginning sampling again. You do have to keep track of which sets of DAC outputs are suitable for which mode! When editing the DAC pulses in a sampling configuration intended to be used in both clamping modes in this manner you should make sure that the full scale value currently set in the DAC calibration is the larger of the two values you expect to use, otherwise the pulses dialog will limit the pulse sizes that you can enter.

The MC700 button in the clamping configuration page of the sampling configuration can be used to retrieve the current MultiClamp settings and use them to set the clamping sets, channels and DACs used for clamping, ADC calibrations, clamping mode and DAC scalings in the sampling configuration. This allows you to see what the settings are without starting sampling and should make it easier to set up an integrated Signal/MultiClamp system.

### Control DAC VClamp units and IClamp units

These two items select the units that will be used for Control DAC calibration in voltage-clamp and current-clamp modes respectively so that the units will not vary with the amplifier settings. The DAC units need to be fixed so that a control pulse 200 pA in size is not spontaneously changed to a 200 nA pulse because on a change to the control amplifier settings - as this would obviously be incorrect behaviour.

### Other controls

The text at the bottom indicates the voltage range in use as set by an item in the sampling preferences.

The button marked 'Test' at the bottom of the dialog is used to test that Signal can communicate successfully with the MultiClamp Commander software for the currently selected amplifier and will provide some diagnostic information if the test fails. Of course, you need to be running MultiClamp Commander (on the same machine) for Signal to be able to communicate with it.

# AxoClamp 900A telegraphs

The AxoClamp 900A amplifier originally supplied by Axon Instruments (now Molecular Devices) has two separate sections each controlling a headstage. There are also two output sections each of which has a scaled output that can carry various signals and a current output that shows the current passing through a headstage, giving up to four signals that can be sampled. The headstages can be configured in a number of modes; depending upon how it is used the amplifier can act as a dual-channel current clamp amplifier or a single channel (one or two-electrode) voltage clamp amplifier. The AxoClamp commander software controlling the amplifier generates telegraph information telling interested applications about changes to amplifier gain settings and other settings such as the headstage mode and external command sensitivity. Using the AxoClamp 900A telegraph configuration dialog you can configure Signal so that it can receive and make use of this telegraph information.

In addition to signalling changes to the amplifier gain, the AxoClamp 900A control software provides information about other aspects of the amplifier setup. For each channel in the amplifier, Signal can retrieve:

- The scaled output's low-pass filter settings
- The external command sensitivity

Both of these values (for both headstages) are retrieved for every frame and placed into extra frame variables in the sampled data file. The frame variable values can be viewed using the File information dialog, plotted in a trend plot or retrieved using the script language.

*Integration with Signal channel scaling*

The basic AxoClamp 900A support uses the information provided by the AxoClamp 900A control software to adjust the port scaling factors to match the channel gains set on the amplifier in the same way as for 1401-based telegraphs. The AxoClamp 900A support is also able to set the initial port scale factors, name and units to match the AxoClamp 900A settings, ensuring that all information for ports sampled from AxoClamp 900A outputs match the amplifier settings.

*Integration with Signal clamping support*

Extra AxoClamp 900A support features integrate the Signal support for clamping experiments with the AxoClamp 900A amplifier setup. This additional mechanism detects suitable pairs of in-use AxoClamp 900A outputs, selects between voltage-clamp and current-clamp according to the headstage mode and in addition sets up the scaling and units of the DAC used for external amplifier control. This ensures that all relevant controls and calibrations are correct at the start of a clamping experiment. A button available in the clamping configuration page can be used to set all the clamping, ADC port and DAC scaling information up to match the AxoClamp 900A settings before you start sampling.

The information in Getting started with clamping experiments using the MultiClamp 700 support contains much that is highly relevant to a user of the 900A.

# AxoClamp 900A telegraph configuration

The AxoClamp 900A telegraph configuration dialog is used to configure the Signal software so that it can communicate with the AxoClamp Commander software controlling the amplifier and to define how you want information from the amplifier to be used.

At the top of the dialog is an item for the serial number of the amplifier which is used to identify it. If you do not want to make use of the AxoClamp 900A telegraph system in this sampling configuration, set the serial number to -1.

Below the serial number is an area used to define the 1401 ADC ports and DACs that are connected to the amplifier. Each 900A amplifier has two headstages, each of which can be configured in various ways, and two output channel sections (almost but not quite corresponding to headstages). Each output channel section has a scaled output which can provide a number of different signals with variable gain, a headstage-specific current output signal and an I-Clamp command input, there is also a separate V-Clamp command input. Set the ADC ports on the 1401 that are connected to the various amplifier outputs, leaving any amplifier outputs that are not being used set to None. Similarly, set the DAC number that is connected to the relevant command input.

## Automatic signal calibration

The check box labeled 'Get signal names, units and calibration from amplifier setup' is used to allow automatic signal definition & calibration from the amplifier up. When set, ports connected to 900A outputs are automatically set up using information retrieved from the amplifier. The information retrieved is the port name and units (depending upon what signal you have routed to the relevant 900A output, current signals are always calibrated in pA) plus both of the port calibration values. This is the easiest way to use 900A telegraph data - it is generally recommended as it will force channels sampled from 900A outputs to be correct.

If this check box is left clear, then the name, units and calibration entered in the Signal port setup dialog (from the Port Setup page in the sampling configuration) are used. You can set the channel name and units to anything you wish, the port Zero value should be set to 0 and the Full scale value should be set to the value of a full-scale signal with the 900A amplifier **gain set to unity**. It is very important that you enter the value for unity amplifier gain, not

whatever gain you have in use at the moment. The full scale value is the value corresponding to a +5 volt (or +10 if appropriate) signal on the 1401 input.

## Automatic clamp setup

The check box labeled 'Get clamping mode and control DAC scaling from amplifier setup' is used to allow automatic setup of the Signal clamping system using the amplifier settings. When set (the check box for automatic signal calibration must also be set for this option to be available) clamping sets within the Signal software will be set using the amplifier settings at the time that sampling starts.

In order for a clamping set to be detected a pair of channel outputs must be found that are in use and provide a pair of signals (current and potential) coming from an active headstage (or, for twin electrode modes, suitable signals from both headstages). The clamping set mode is set using the headstage mode information and the control DAC in use with the headstage is calibrated. The DAC scaling is set using the external command sensitivity information from the amplifier, the DAC units are set using the Control DAC VClamp units or IClamp units fields in the configuration dialog.

With this option enabled you can set up a single sampling configuration containing control DAC pulse sequences suitable for both voltage clamp and current clamp (knowing which units that the DAC will use in either mode) and can quickly switch between voltage and current clamp experiments by stopping sampling, switching the 900A amplifier mode, and beginning sampling again. You do have to keep track of which sets of DAC outputs are suitable for which mode! When editing the DAC pulses in a sampling configuration intended to be used in both clamping modes in this manner you should make sure that the full scale value currently set in the DAC calibration is the larger of the two values you expect to use, otherwise the pulses dialog will limit the pulse sizes that you can enter.

### dSEVC issues

In tests with the AxoClamp commander software (V 1.0.0.62) running in demonstration mode we have found problems when headstage 1 is set up in discontinuous Single Electrode Voltage Clamp (dSEVC) mode. These may be an effect of demonstration mode, or a misunderstanding, but it appears that when dSEVC mode is selected AxoClamp commander automatically switches to using the channel 1 being HS1 membrane potential and channel 2 scaled output being HS1 membrane current. However the information given to Signal always says that channel 1 is showing HS1 membrane current. If you are reading back the clamp settings from the amplifier this will not work correctly unless you set up AxoClamp commander so that channel 1 is showing HS1 membrane current (and channel 2 HS1 membrane potential). As I said I'm not certain if this is a fault somewhere or a misunderstanding, we are in contact with Molecular Devices to try to resolve the issue.

The 900A button in the clamping configuration page of the sampling configuration can be used to retrieve the current AxoClamp 900A settings and use them to set the channels used for clamping, ADC calibrations, clamping mode and DAC scalings in the sampling configuration. This allows you to see what the settings are without starting sampling and should make it easier to set up an integrated Signal/AxoClamp 900A system.

## Control DAC VClamp units and IClamp units

These two items select the units that will be used for Control DAC calibration in voltage-clamp and current-clamp modes respectively so that the units will not vary with the amplifier settings. The DAC units need to be fixed so that a control pulse 200 pA in size is not spontaneously changed to a 200 nA pulse because on a change to the control amplifier settings - as this would obviously be incorrect behaviour.

## Other controls

The bottom of the dialog shows an indicator of the 1401 voltage range in use, this is set by an item in the sampling preferences.

The button marked 'Test' at the bottom of the dialog is used to test that Signal can communicate successfully with the AxoClamp Commander software and will provide some diagnostic information if the test fails. Of course, you need to be running AxoClamp Commander (on the same machine) for Signal to be able to communicate with it.

# EPC 800 telegraphs

The EPC 800 amplifiers supplied by Heka can operate in either voltage or current clamp mode under the control of a DAC driving either the external VC or external CC input and has two signal outputs, one for voltage and one for current. When the EPC 800 is controlled manually but connected to the PC using the built-in USB interface, Signal can interrogate the amplifier to read back the mode and gain settings and use this information to configure itself. Using the EPC 800 telegraph configuration dialog you can configure Signal so that it can receive and make use of this telegraph information from up to eight EPC 800 amplifiers.

In addition to detecting changes to amplifier gains, output signals and operating modes, the EPC 800 telegraph software also provides information about other aspects of the amplifier setup. For each amplifier, Signal can retrieve:

- The low-pass filter settings
- The fast membrane capacitance
- The series resistance
- The external command sensitivity

All of these values for each amplifier are retrieved for every frame of data and placed into extra frame variables in the sampled data file. The frame variable values can be viewed using the File information dialog, plotted in a trend plot or retrieved using the script language.

### Integration with Signal channel scaling
The basic EPC 800 support uses the information read from the EPC 800 amplifier to adjust ADC port scaling factors to match the amplifier gains in the same way as for 1401-based telegraphs. The EPC 800 support is also able to set the initial port scale factors, name and units to match the EPC 800 settings, ensuring that all settings for ports sampled from EPC 800 outputs are correct.

### Integration with Signal clamping support
Extra EPC 800 support features integrate the Signal support for clamping experiments with the EPC 800 amplifier setup. This additional mechanism detects pairs of used EPC 800 outputs, selects between voltage-clamp and current-clamp according to the amplifier controls and in addition sets up the scaling and units of the DACs used for external amplifier control. This ensures that all relevant controls and calibrations are correct at the start of a clamping experiment. A button available in the clamping configuration page can be used to set all the clamping, ADC port and DAC scaling information up to match the EPC 800 settings before sampling.

The information in Getting started with clamping experiments using the MultiClamp 700 support contains much that is highly relevant to a user of the EPC800.

# EPC 800 telegraph configuration

Signal can receive telegraph information from up to eight EPC 800 amplifiers. The EPC 800 telegraph configuration dialog is used to configure the Signal software so that it can communicate with the EPC 800 amplifiers and to define how you want information from that amplifier to be used.

At the top of the dialog is a selector for the amplifier that you want to configure, with the amplifier settings below. At the top of the settings is a field for the serial number of the EPC 800 so it can be identified if you have multiple EPC 800 amplifiers; you can leave this field set to zero if you only have a single amplifier. If you do not want to make use of the EPC 800 telegraph system in this sampling configuration, set the serial numbers of all amplifiers to −1.

Below the amplifier identification section is an area defining the 1401 ADC ports and 1401 DACs that are connected to the amplifier. Each EPC 800 amplifier has two signal outputs, one called Voltage monitor and one called Current monitor, plus a pair of external control inputs, one for voltage clamp and one for current clamp. Set the ADC ports on the 1401 that are connected to the various amplifier signal outputs, leaving any amplifier outputs that are not being used set to None. Similarly, set the DAC numbers that are connected to the external control inputs. It is not an error to use the same DAC to control both voltage clamp and current clamp inputs.
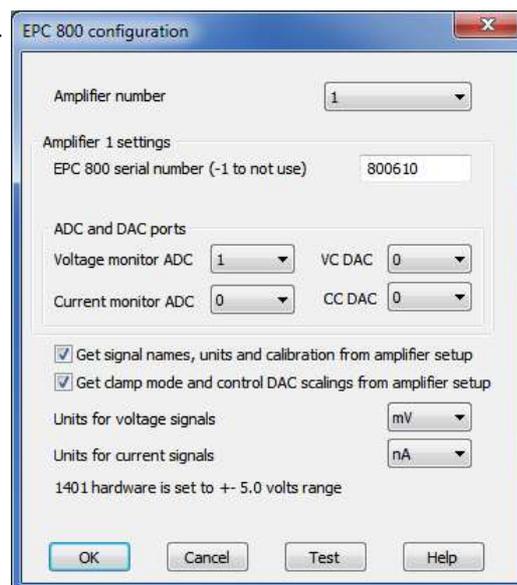
## Automatic signal calibration

The check box labeled 'Get signal names, units and calibration from amplifier setup' is used to allow automatic signal definition & calibration from the amplifier up. When set, ports connected to EPC 800 outputs are automatically set up using information retrieved from the amplifier. The information retrieved is the port name and units (depending upon what signal you have connected) plus the port calibration values. This is the easiest way to use EPC 800 telegraph data - it is generally recommended as it will force channels sampled from EPC 800 outputs to be correct.

If this check box is left clear, then the name, units and calibration entered in the Signal port setup dialog (from the Port Setup page in the sampling configuration) are used. You can set the channel name and units to anything you wish, the port Zero value should be set to 0 and the Full scale value should be set to the value of a full-scale signal with the EPC 800 amplifier **gain set to unity**. It is very important that you enter the value for unity amplifier gain, not whatever gain you have in use at the moment. The full scale value is the value corresponding to a +5 volt (or +10 if appropriate) signal on the 1401 input.

## Automatic clamp setup

The check box labeled 'Get clamping mode and control DAC scalings from amplifier setup' is used to allow automatic setup of the Signal clamp system from the amplifier settings. When set (the check box for automatic signal calibration must also be set for this option to be available) clamping sets within the Signal software that match an EPC 800 amplifier will be set to match the amplifier settings at the start of sampling.

In order for a clamping setup to match a EPC 800 channel and thus be set up using amplifier information, the port connected to current monitor must be used to sample the Response channel defined in the clamping setup in voltage clamp mode (or the Stimulus channel in current clamp mode). Similarly the port connected to the voltage monitor must be used to sample the Stimulus (or Response) depending upon the clamping mode. If these criteria are met then the clamping mode is set to voltage clamp or current clamp as appropriate and the control DAC in use in that clamp setup is calibrated. The DAC scaling is set using the external command sensitivity information from the amplifier, the DAC units are set as defined using the Control DAC VClamp units and IClamp units fields in the configuration dialog.

With this option enabled you can even set up a single sampling configuration containing control DAC pulse sequences suitable for both voltage clamp and current clamp (knowing which units that the DAC will use in either mode) and can quickly switch between voltage and current clamp experiments by stopping sampling, switching the MC700 amplifier mode, and beginning sampling again. You do have to keep track of which sets of DAC outputs are suitable for which mode! When editing the DAC pulses in a sampling configuration intended to be used in both clamping modes in this manner you should make sure that the full scale value currently set in the DAC calibration is the larger of the two values you expect to use, otherwise the pulses dialog will limit the pulse sizes that you can enter.

The EPC 800 button in the clamping configuration page of the sampling configuration can be used to retrieve the current amplifier settings and use them to set the clamping sets themselves, the channels and DACs used for clamping, ADC port calibrations, clamping mode and DAC scalings in the sampling configuration. This allows you to see what the settings are without starting sampling and should make it easier to set up an integrated Signal/EPC 800 system.

## Voltage and current signal units

These two items select the units that will be used for ADC port and external control DAC calibration of voltage and current signals respectively. Thse are provided so that Signal can use the units that you prefer, set correct scalings, and generate the correct control pulses on the DACs.

## Other controls

At the bottom of the dialog is an indicator of the voltage range in use. The 1401 voltage range is set by an item in the sampling preferences and (often) updated with information read from the 1401 itself.

The button marked 'Test' at the bottom of the dialog is used to test that Signal can communicate successfully with the configured EPC 800 amplifiers and will provide diagnostic information if the test fails. Of course, the amplifier needs to be connected (on the same machine) and turned on for Signal to be able to communicate with it.

# Auxiliary states devices

When sampling using multiple frame states, Signal can control external devices such as stimulators in addition to switching the 1401 outputs so that the stimulators are set up differently for each state. This is achieved by using auxiliary states devices; special software modules that control external equipment.

Signal auxiliary states device support provides a mechanism for controlling selected external equipment and setting it up in different ways according to the sampling state in use. In addition to controlling equipment settings by state, Signal is also able to provide device-specific configuration dialogs, to check the equipment health and readiness while sampling, to save the stimulation parameters used in the new data file, to change many of the stimulator settings while data is being sampled and to save and load auxiliary states setup information to and from sampling configurations. You can select which auxiliary states devices you wish to use (or none) using the multiple states page in the Signal sampling configuration dialog. The configuration dialogs of auxiliary states devices that are in use can be accessed during sampling using a popup menu in the main Sample menu or buttons in the auxiliary states control bar. When used online the overall device settings cannot be altered, but the settings for the individual states can be freely adjusted.

Script language functions (`SampleAuxStateNew()` and `SampleAuxStateRemove()`) are provided to control which auxiliary states devices are used in a sampling configuration while others (`SampleAuxStatePar$()`, `SampleAuxStateParX()` and `SampleAuxStateValX()`) are provided to give access to the auxiliary device settings.

Signal currently supports six types of auxiliary states device; the Magstim, MagPro, PowerMAG and DuoMAG transcranial magnetic stimulators and the CED 3304 and Digitimer DS8R constant current stimulators. If you need auxiliary states device support for equipment other than those currently supported, please contact CED.

## Magstim device

The Magstim range of transcranial magnetic stimulators produced by the Magstim company are widely used to provide non-invasive neuronal and muscular stimulation. Signal provides support for all three types of Magstim commonly in use; the $200^2$, BiStim$^2$ and Rapid$^2$, it can also control a pair of Magstim $200^2$ units simultaneously as a single auxiliary states device. Signal uses a serial line for Magstim control so only the modern Magstims that provide serial line control (with the $^2$ suffix to the model name) can be used.

## Magstim safety notice

Transcranial magnetic stimulators are dangerous devices capable of causing serious harm and should only be used by qualified medical practitioners. Before using a Magstim device you should read the user's manual produced by Magstim paying particular attention to the warnings and precautions section. It is your responsibility to ensure that Signal's control of a Magstim is set up in an appropriate and safe fashion for the intended use and to verify that it is operating correctly.

Signal carries out checks to ensure that the Magstim hardware is operating correctly. Firstly, if communications between Signal and the Magstim breaks down then Signal will disarm the Magstim and terminate sampling as unless the communications are working correctly you cannot be sure that the Magstim settings will be those that you have selected. Secondly, after sending new settings to a Magstim Signal always reads back the settings and disarms the Magstim and terminates sampling if they do not match the settings that were sent. Finally, the status information obtained from the Magstim is monitored and if there appears to be any problems Signal will again disarm the Magstim and terminate sampling.

In addition to these checks the Magstim Rapid$^2$ settings are tested in various ways before they are used because the Rapid$^2$ is capable of a very high sustained power output - so high indeed that the equipment might be seriously damaged by extreme settings. Signal attempts to detect situations where it appears that the safe power output levels will to be exceeded and will warn you if it appears that there are problems with the sampling configuration or Magstim settings. But these checks should not be viewed as an absolute guarantee of safety - the lack of warning messages should not be taken as meaning that the configuration is certain to be good; it is always the user's responsibility to ensure that the equipment is used correctly.

We must emphasise that CED is not responsible in any way for problems caused by use of Signal with Magstim equipment.

# Magstim introduction

For Magstim devices, the auxiliary states system sets the power output and (where appropriate) inter-pulse timing and other settings, remote control of the hardware can be disabled for specific states to allow the user to control the stimulation manually. Options are provided to cause Signal to assume independent triggering mode and to enable high resolution interval timing (for the BiStim$^2$) and to cause the Magstim to ignore the coil interlock switch and to enable single pulse mode (for the Rapid$^2$). This documentation only describes the controls available from within Signal; for a complete understanding of the effect these have upon the behaviour of the stimulator you should consult the Magstim user manuals.

In addition to using a serial-line to set the stimulation parameters, standard operation with Signal requires that the stimulation be triggered using one or more TTL pulses generated by the 1401 digital outputs. This allows the pulse timing to be precisely controlled relative to the Signal sampling. A Rapid$^2$ can be configured to either generate a train of pulses in response to a single trigger or to generate one pulse per trigger, allowing complete control of the patterns of stimulation. Manual triggering of the stimulator is also possible, as is triggering from other external hardware, in which case you would use the Magstim trigger to trigger the Signal sampling sweep, however please note that the power level checks performed upon the Rapid$^2$ require that the device be triggered by the 1401 so that the trigger times are known.

The Magstim support monitors the stimulator health; both waiting until the Magstim is ready for use before allowing a sampling sweep to proceed and terminating sampling if a hardware problem develops. To aid data analysis, Signal saves the Magstim stimulation parameters used in the data section variables of the sampled file, note that these values are saved even if the Magstim is manually controlled. The parameters saved are the power level, the pulse interval (BiStim only), the secondary power level (BiStim$^2$ and dual 200$^2$), the pulse frequency (Rapid$^2$ only) and the pulse count (Rapid$^2$ only). These values are placed in extra frame variables in the sampled data file. The saved values can be retrieved using the script language and used as measurements in trend plot generation.

## Getting started with TMS experiments

Signal is a very general-purpose program, much more so than (for example) the data acquisition and analysis software built into the Magstim Rapid. While this does have advantages - for example Signal can carry out an extremely wide range of experiments and analyses - it does make getting started with TMS experiment a bit harder, and the sometimes-complex nature of the interaction between Signal and a Magstim can also make things difficult. There is more information about using Signal data acquisition here, and information about sampling with multiple states here, you will need to have read both of these Signal help chapters to make good use of this. This short guide is aimed at helping you with aspects specifically related to the Magstim system. Nearly all of the information here is also relevant to other auxiliary states devices, in particular the other TMS stimulators such as the MagPro stimulator manufactured by MagVenture.

### Enabling the Magstim software

The auxiliary states interface to the Magstim and other auxiliary states devices are always installed with Signal. You have to add the support for your device to the sampling configuration using the Auxiliary >> button in the States page of the sampling configuration dialog before it will be available for use, if you have not done so then use the menu provided by the Auxiliary >> button to add it.

### Configuring Magstim device and communications

The first thing you have to do is to use the Magstim configuration dialog to define the device(s) that you are using and the serial port that you are using to control it. You can set up information for only one Magstim device (counting the dual 200 as a special device). This part of the configuration dialog does two things really: it tells Signal what sort of hardware you are using and it tells Signal what serial line(s) to use to communicate with the Magstim.

You should start by connecting a serial line on the PC that is running Signal to the serial control port on the Magstim. There are two different sorts of Magstim serial control port; the cable you use must be of the correct sort! CED can supply a suitable cable which also includes a BNC connector to be attached to a 1401 digital output - this digital output connection will be used to trigger the Magstim at the correct time.

In the configuration dialog, the Magstim type and serial port number number(s) should be set to match your hardware and the serial port(s) that you are using. If you are using a Rapid$^2$ device, you may need to enter the device unlock code if your Rapid is using the newest firmware, and you should set the digital output used to trigger the Rapid as this allows Signal to carry out  various checks of the Rapid$^2$ power dissipation and thus avoid

overheating or damaging the Rapid. The Test button at the bottom of the configuration dialog allows you to check that communications with the hardware are working correctly. This button is not a complete test of the communications; we have found that in some circumstances the Test button does not find any problems but communications errors still occur when you try to use the software. However if the Test button shows errors then you can be sure that something is wrong; there is no point in trying to go further until this shows that things are OK.

*Communications problems and how to fix them*
If the Test button shows that there are communications errors then this could be due to:

- The COM port that you have set is incorrect - it does not exist or is not the right one. Fix this by changing the COM port selection or where the serial line cable is connected to the PC.

- The Magstim device that you have set does not match your hardware. Fix this by changing the Magstim device that is set in the dialog.

- The serial line cable is faulty, or not completely plugged in at one end or the other. Check that the cable wiring is as it should be and that the cable is fully plugged-in at both ends.

- You have plugged the serial cable into the wrong port on a BiStim$^2$. The BiStim$^2$ is just two 200$^2$ units connected together with a special adaptor, so there are two serial control ports - one on each 200$^2$. You should be connected to the control port on the top unit.

- You are using a USB serial port which does not handle the Magstim control. Not all USB serial ports are created equal! Some work very well with a Magstim while others simply do not, or work erratically (this is a common reason for the Test button saying all is OK but communications errors happening during experiments). If you have other USB serial ports around, try them - a different one may 'just work'. If you want to buy a USB serial port for use with a Magstim there are two rules to follow: a) don't buy a really cheap device and b) if at all possible buy a device where you know the manufacturer - so no cheap Chinese knockoffs! Most such devices will work well, if you want more guidance I have found that Belkin hardware seems to always work OK, but the most reliable solution of all is to use an actual serial port - either one on the PC motherboard or on a plug-in PCI card.

If none of these suggestions help you, contact CED for assistance.

**Setting up the stimulator**
You then need use the main part of the Magstim configuration dialog to set the stimulator settings for each sampling state. The dialog displays a list describing the settings for each state and below this, the settings for one state where they can be edited. You select the state you want to edit by clicking on the correct line in the list, there are also useful buttons that can be used to copy the settings for the currently edited state to all other states or to all state numbers above the number of the currently edited state. The settings that are available vary according to the device type, there are of course no 'correct' settings as what is wanted depends upon your experiment design and aims. However for the purposes of checking that things are working OK I suggest using power levels of about 30%, moderate BiStim pulse intervals (perhaps 10 ms?) and short Rapid$^2$ pulse trains, perhaps 3 pulses, the idea being to make sure that there are no power level issues for your initial tests. The Signal multiple states system assumes that state number zero is an inert or background state, so you should probably start off with state zero being zero power or manual control - to start off with a tidy arrangement.

There are various other controls available in the configuration dialog according to the type of hardware in use. Generally they can be left alone for initial testing - see the pages on configuring the various device types for details.

You will also need to make sure that the digital output that you will use to trigger the stimulator is enabled in the outputs page of the sampling configuration and that the pulses dialog shows a suitable trigger pulse being generated on that output at the correct point during the sweep. In keeping with the way Signal handles multiple states, no trigger pulse should be generated for state zero.

You should then be able to run the sampling configuration (with sweep triggers off, most probably). While the sampling is happening the stimulator will be under the control of Signal (except for states set as Manual control) and you will see Signal waiting until the stimulator is ready before starting each sweep. You should also be able to hear the stimulator firing at the correct point within each sweep.

Once you have sampling working with this simple configuration, you should alter the configuration so that the Magstim power levels and other settings are similar to what you will want to use in a real experiment and try those.

If you get Magstim communications errors during the sampling this may indicate that your USB serial port cannot cope with use during sampling, even though it was OK with the Test button. You should try a different make of

USB serial port (or use an actual serial port) to see if this cures the problem. If that does not help, contact CED for assistance. After a failure during sampling, the Magstim support will put text in the Signal log window indicating what went wrong - please copy that into any emails set to CED as that may help us work out what is going wrong.

If you get 1401-related errors during the sampling, in particular when the Magstim is firing at high power, then this may indicate that the electromagnetic interference generated by the Magstim (a seriously powerful device) is causing 1401 hardware problems. It may be difficult to find a solution to this problem, the basic idea is to isolate and separate the 1401 and amplifiers side of things from the Magstim. There are a number of things you can try to attempt to improve this isolation/separation:

- Have the Magstim on one side of the room/subject, with the 1401 and amplifiers on the other side.

- Power the Magstim from a different mains supply from the 1401 and amplifiers. You can also try powering the 1401 and amplifiers from a UPS (Uninterruptible Power Supply) for still greater separation.

- Keep all electrode leads & if possible the electrodes themselves as far as possible from the Magstim coils.

### Getting your sampling configuration correct

The Magstim support will work quite well with most Signal sampling configurations as long as suitable trigger pulses are generated and multiple sampling states are in use. To learn about the relevant aspects of Signal sampling, you will need to read relevant pages of this help file, the Sampling data, Pulse outputs while sampling and Sampling with multiple states chapters will probably all be relevant.

One simple mechanism that is used by many TMS experimenters is fixed interval sampling mode. With this mode, Signal sampling sweeps do not happen when a trigger pulse is received but instead they are triggered automatically by a timer. So you could have a 700 millisecond sweep of sampling, then an automatic delay of maybe 10000 milliseconds until the next sampling sweep occurs. This is a very common way of running TMS experiments and works well within Signal. Signal is also able to generate a random variation (of perhaps up to ± 500 milliseconds) in this fixed interval timing. Magstim stimulators can take quite a while to change the power levels, so you may get missed trigger messages at the end of sampling if you set the fixed interval less than about 10 seconds.

There is another way of organising a TMS sampling configuration, where the experimenter wants to control the timing of stimulations manually and have Signal just set the stimulator settings and record the data. This is easily accomplished by:

- Using Peri-triggered sampling mode (with an event trigger) which allows a sweep trigger to occur at a set point within the sampling.

- Modifying the connection between the Magstim and the 1401 so that instead of a digital output going to the Magstim trigger input, the Magstim low-going trigger output is connected to the 1401 trigger input.

- Using a footswitch or equivalent to trigger the Magstim as required.

Though this method is less common than using the 1401 to trigger the Magstim it is equally satisfactory and will work equally well.

### Issues with amplifier overloads

When a Magstim fires it generates a very intense magnetic field which can induce large voltages in electrodes and wiring. These transient voltages can saturate amplifier inputs, which will cause them to take rather long to recover after the stimulation - too long for many experiments. Some amplifiers such as the CED 1902 are able to clamp the amplifier inputs during the stimulation period to prevent this saturation, which is a generally satisfactory solution to this - see your amplifier manuals or manufacturer for more details on this.

# Magstim configuration

To configure a Magstim in Signal, open the sampling configuration dialog and make sure that Multiple states are enabled in the General section. Select the States section and make sure that you are in *Dynamic outputs* or *Static outputs* mode. If Magstim usage has been added to the sampling configuration, clicking on the Auxiliary >> button will show a Magstim item in the lower section of the menu. If this item is not present in the menu, use the Add device item at the top of the menu to add it. If the Auxiliary >> button is present, but disabled, you are in *External Digital* states mode, change the mode to enable the button. Click on any Magstim item in the lower section of the menu to open the Magstim configuration dialog for that device.

You can view and change the device configuration during sampling either by selecting the device in the sub-menu provided by the Sampling menu Auxiliary states devices option, or by pressing the relevant button in the auxiliary states control bar that you can choose to have displayed.

It is meaningful to use the Magstim support without using multiple states, but this is not directly supported because the states system is so closely linked to the Magstim support. If you want to make use of Magstim support without using multiple states (so that the pulse parameters are recorded and Magstim health checks are made, but with complete manual control) you should turn on multiple states, set only one extra state, set state zero to use manual control, and only make use of state zero while sampling. This will allow you access to the Magstim configuration dialog to set the basic Magstim parameters.

### Label

At the top of the configuration dialog there is a field where you can enter a device label used to distinguish this device from others.

### Type of stimulator

Below the label there is a selector used to select the Magstim type. You should set this to the type of stimulator you will be using or to Do not use if you do not want to make use of the Magstim support in this sampling configuration.

### Don't generate settings text for window title

This check box prevents the Magstim support from generating text showing the current level that Signal can display in the title bar of a sampling document. It is often useful to see what the Magstim settings are, but some users find it distracting. A separate check box in the sampling preferences must also be set to allow this text to be shown.

### Com port for communications

This is a selector for the COM port used to control the stimulator, you can select any of the PC COM ports that are detected as available. As many PCs only have one COM port you may have to use a USB serial port expander to provide a spare COM port – we have found that most of these USB serial ports work satisfactorily. When a dual-$200^2$ device is selected, two COM port selectors are shown.

### Stimulator-dependent controls

Below these basic stimulator configuration controls there may be more controls, these will vary according to the type of stimulator selected and will be described in the specific device configuration details (see below).

### Magstim settings

Following these are two more sections holding the actual Magstim settings; first there is a display of settings for all states which is used to select a state for editing (by clicking on the information for that state) and finally an area

where the settings for the selected state may be edited. Again, the information shown here varies according to the type of stimulator selected and will be described in the specific device configuration details (see below).

**Copy to ...**

The button labelled Copy to ... provides a simple dialog that can be used to set up many states quickly. If you press OK it copies the currently selected state to all the other states that are within the selected range. This button is available for all types of Magstim.
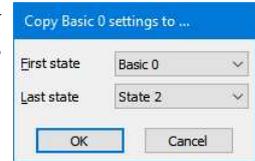
**Test**

The button labelled Test tests if communications with the Magstim can be established. It checks that you have the correct COM port, that the serial line is connected and that the Magstim is communicating correctly with Signal. It also detects if you have set an incorrect Magstim type and if you have the Independent BiStim Trigger check box incorrectly for a BiStim$^2$ device. For a Rapid$^2$ device the power safety checks are carried out on the current parameters and any problems reported. The Test button is available for all types of Magstim.

# 200 device configuration

The 200$^2$ is the simplest type of Magstim unit. When a 200$^2$ device is selected one extra pair of controls governing overall behaviour is shown:

**Powers as % of RMT**

When this check box is set, the power values entered for the individual states are percentages of the subject resting motor threshold (RMT) rather than a percentage of the overall device power (when the check box is clear). The RMT value in use, which is always a percentage of the overall device power, is entered to the right of the check box.

The settings available for the individual states are:

**Manual**

Set this check box for this state to be controlled by the manual controls and not by Signal, clear it to give Signal control.

**Power**

This sets the stimulator power output as a percentage of the maximum available, you can set any value from 0 to 100. If device power is being specified as a percentage of the RMT, you can set any value from 0 to 100/RMT. This field is disabled for manually controlled states.

# BiStim device configuration

The BiStim$^2$ is more complex than the 200$^2$, in essence it is two 200$^2$ units connected together with one unit (the lower one) controlled by the other. The serial line used by Signal to control the BiStim$^2$ should be connected to the upper, master, unit. When a BiStim$^2$ device is selected two controls governing overall behaviour are shown:

### BiStim in independent trigger mode

This sets how Signal will expect the BiStim$^2$ to be configured, rather than how it will be set up by Signal - this is because Signal is unable to put the BiStim$^2$ into or out of IBT mode. Normally, a BiStim$^2$ generates two output pulses, one at the time of the trigger and the second at a preset interval after the trigger. Using the controls on the front of the BiStim$^2$ master unit, you can set the device into independent trigger mode (referred to as IBT mode in the Magstim documentation). When this mode is in use two separate triggers are used (presumably signals from two 1401 digital outputs), one trigger firing the main pulse and the other firing the second (Power B) pulse. Because a BiStim$^2$ behaves rather differently when in IBT mode you have to set this check box to match the BiStim$^2$ setup otherwise errors will be generated when you test the configuration or try to sample.

### Use Hi-res timing mode

When this check box is clear, the interval between the two pulses can be set to any value between 0 and 999 milliseconds with a resolution of 1 millisecond. With Hi-res timing enabled the maximum interval is reduced to 99.9 milliseconds but the timing resolution is improved to 0.1 milliseconds. This control is disabled if independent trigger mode is selected.

### Powers as % of RMT

When this check box is set, the power values entered for the individual states are percentages of the subject resting motor threshold (RMT) rather than a percentage of the overall device power (when the check box is clear). The RMT value in use, which is always a percentage of the overall device power, is entered to the right of the check box.

The BiStim$^2$ settings available for the individual states are:

### Manual

Set this check box for this state to be controlled by the manual controls and not by Signal, clear it to give Signal control.

### Power

This sets the main (master) stimulator power output as a percentage of the maximum available, you can set any value from 0 to 100 percent. If device power is being specified as a percentage of the RMT, you can set any value from 0 to 100/RMT. This field is disabled for manually controlled states.

### Interval

This sets the interval between the two pulses, you can use any value from 0 to 999 milliseconds (or 0 to 99.9 milliseconds for Hi-res timing). Note that the BiStim$^2$ cannot tolerate timing intervals between 0 and 1 millisecond in Hi-res mode, if you enter such a value the interval will be rounded to the nearest of 0 and 1 millisecond. Setting a zero interval will switch the BiStim$^2$ into simultaneous pulse mode, setting a value greater than zero switches it out of simultaneous mode. In simultaneous pulse mode both stimulators must use the same power level, so both levels are set by the main power field. This field is disabled if independent BiStim$^2$ triggers are in use or for manually controlled states.

### Power B

This sets the second (slave) stimulator power output as a percentage of the maximum available, you can set any value from 0 to 100. If device power is being specified as a percentage of the RMT, you can set any value from 0 to 100/RMT. This field is disabled for manually controlled states.

# Rapid device configuration

The Rapid[2] is rather different from the other Magstim devices as it is capable of producing a train of pulses at high rates. The Rapid[2] power supply is very powerful; so powerful in fact that it is capable of generating so much power that the Rapid[2] can be damaged, Signal carries out hardware safety checks when getting ready for sampling to try to prevent this happening. Rather than the simple case-mounted manual controls of the other two units, the Rapid[2] has a separate control system that has to be disconnected to gain access to the serial line control port. This makes manual control unusable and so it is not available in the individual state settings. There are four controls governing overall behaviour:

### Ignore coil interlock switch

This disables checks on the coil interlock switch on the handle of the Magstim coil, so that the unit will fire without a button being held down. Magstim do not recommend disabling these checks as they are a safety feature.

### Rapid[2] v9 unlock code

For additional safety, Magstim have introduced a unlock code, a 16-character text string looking something like "29a5-4827780a-cf", with the version 9 firmware. In order to make use of a Magstim Rapid[2] with this (or later) firmware this unlock code must be entered into the setup dialog, you can find the unlock code sequence for your Rapid[2] by contacting Magstim.

### Use Rapid single-pulse mode

This turns on single-pulse mode, which allows higher power levels - up to 110%. In single-pulse mode only a single pulse is produced per trigger and the pulse train parameters are ignored. The pulse rate can be up to 100 Hz normally (depending upon the power level used – see the safety check details here) but if you select power levels above 100%, the maximum allowed pulse rate is forced to 0.5 Hz. With single pulse mode turned off a train of up to 1000 pulses can be generated at the specified frequency from one trigger and power levels above 100% are not available. Again, the pulse rate can be up to 100 Hz depending upon the power level used.

### Digital output used to trigger Magstim

Signal needs to know the digital output used to trigger the Rapid[2] so that it can find the times of trigger pulses, which are needed for the hardware safety checks. If you do not select a digital output Signal will generate a warning message at the start of sampling, if you select the wrong digital output then the Signal hardware safety checks will not behave correctly.

### Powers as % of RMT

When this check box is set, the power values entered for the individual states are percentages of the subject resting motor threshold (RMT) rather than a percentage of the overall device power (when the check box is clear). The RMT value in use, which is always a percentage of the overall device power, is entered to the right of the check box, you can set any value from 0 to 100, or 110 if single-pulse mode is in use.

The Rapid[2] settings available for the individual states are:

#### ReCh del (ms)

This item sets the delay after the stimulator fires before it begins to recharge again, you can use this to delay charging (which can generate some noise) until after any evoked responses that you may be interested-in have occurred. This field is disabled for manually controlled states.

#### Power

This sets the stimulator power output as a percentage of the maximum available, you can set any value from 0 to 100, or 110 if single-pulse mode is in use. If device power is being specified as a percentage of the RMT, you can set any value from 0 to 100/RMT (or 110/RMT).

#### Pulses

This item is only available if single-pulse mode is not in use, it sets the number of pulses, you can set any value from 1 to 1000. This field is disabled if single-pulse mode is selected.

#### Freq

This item is only available single-pulse mode is not in use, it sets the pulse frequency in Hertz, you can set any value from 0 to 100. This field is disabled if single-pulse mode is selected.

## Dual 200 device configuration

Two Magstim $200^2$s can be controlled by Signal to achieve much the same effect as using a BiStim$^2$. When using this configuration two separate serial lines are used to control the MagStims and two separate trigger pulses are required (as for a BiStim in independent trigger mode). When the dual $200^2$ option is selected a separate selector for the second serial-line port is shown and there is also an extra pair of controls governing overall behaviour:

#### Powers as % of RMT

When this check box is set, the power values entered for the individual states are percentages of the subject resting motor threshold (RMT) rather than a percentage of the overall device power (when the check box is clear). The RMT value in use, which is always a percentage of the overall device power, is entered to the right of the check box.

The settings available for the individual states are:

#### Manual

Set this checkbox for this state to be controlled by the manual controls and not by Signal, clear it to give Signal control.

#### Power

This control sets the main $200^2$ stimulator power output as a percentage of the maximum available, you can set any value from 0 to 100. If device power is being specified as a percentage of the RMT, you can set any value from 0 to 100/RMT. This field is disabled for manually controlled states.

#### Power 2

This sets the second $200^2$ stimulator power output as a percentage of the maximum available, you can set any value from 0 to 100. If device power is being specified as a percentage of the RMT, you can set any value from 0 to 100/RMT. This field is disabled for manually controlled states.

## Magstim connections and cabling

There are two connector arrangements provided by Magstim for serial line control and external triggering of the unit. Older 200 and BiStim units have an arrangement with a 9-way serial line and separate 15-way trigger connector while newer 200 and BiStim units and all Rapids have a high-density 26-way combined serial line and trigger connector. If your PC does not provide a serial line (a common situation with laptops) or no unused serial

line connector is available a USB serial line adapter can be used instead, but these can give problems so we advise using standard serial port hardware if possible.

In both cases two separate connections need to be made; serial line control from a PC and a TTL trigger signal from the 1401. The standard high level trigger input suggested by Magstim works well with Signal and the 1401 and is used in the diagrams shown below, consult the Magstim documentation if you require a different arrangement. In the case of a Rapid stimulator, the 26-way connector is also used by the integrated control system and this will need to be disconnected before the control cable can be connected.

**Cabling for 26-way combined serial and trigger connector**
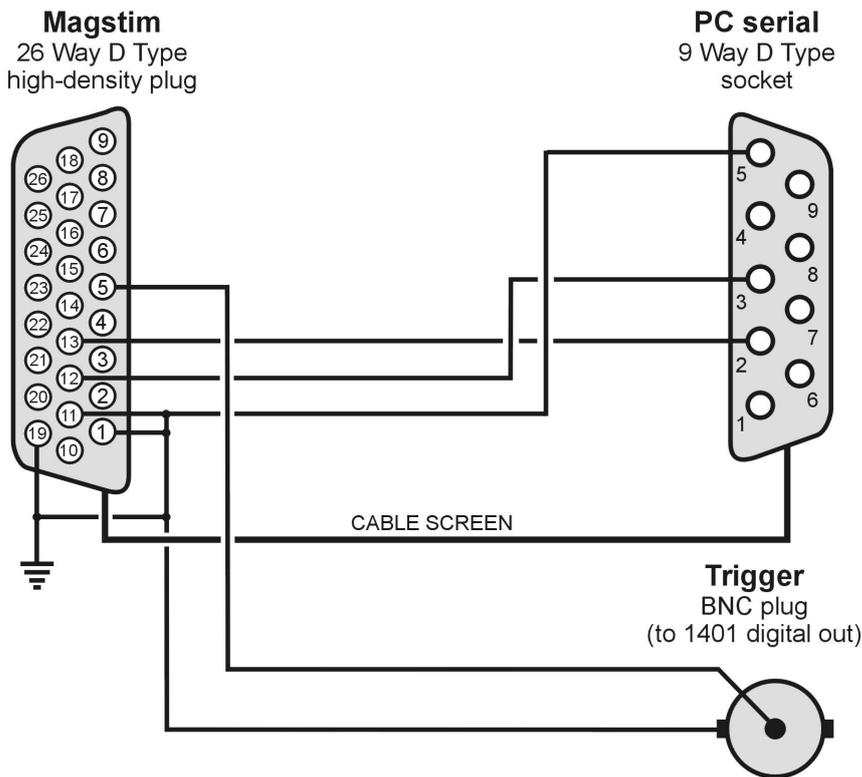
**Cables for separate 9 pin serial and 15-way trigger connectors**



**Magstim**
9 Way D Type
plug

**PC serial**
9 Way D Type
socket

**Serial cable**

CABLE SCREEN

**Magstim**
15 Way D Type
plug

**Trigger**
BNC plug
(to 1401 digital out)

**Trigger cable**
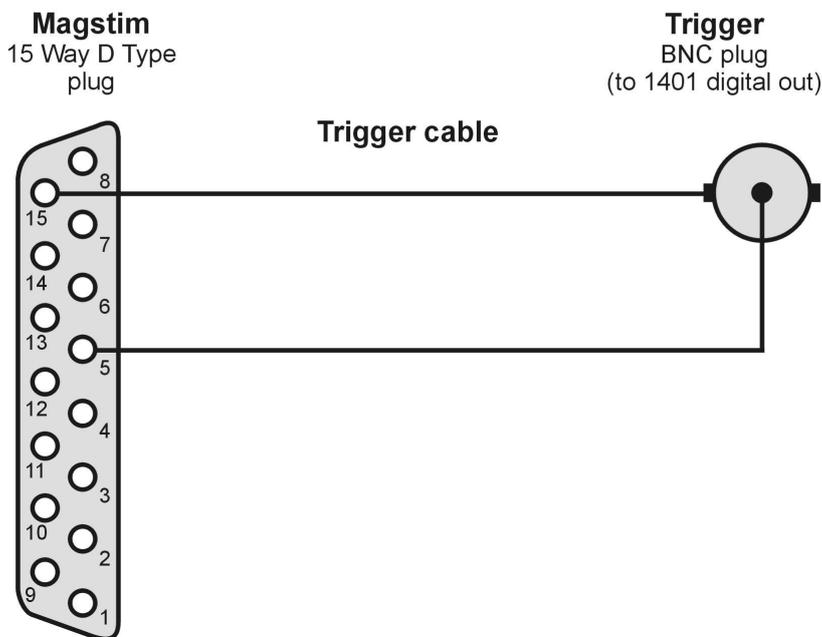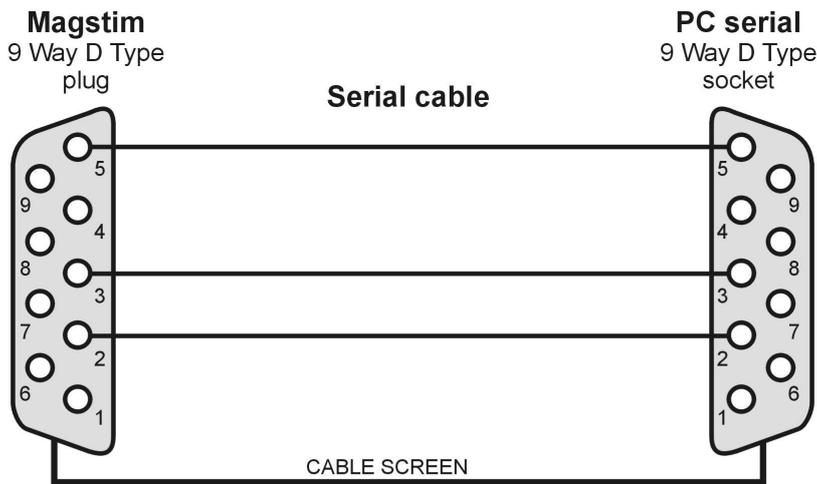
If you are not using the external trigger input to trigger the Magstim from the 1401 you will need a different cabling arrangement, for example to take the Magstim low-going trigger out signal (on pin 8 of the 26-way connector) and use that as an input to the 1401 to trigger the 1401 sweep. Consult your Magstim documentation for more information.

# Notes on Magstim use

A straightforward arrangement to use a Magstim with Signal would be as follows:

1.  Connect the serial port of your computer to the Magstim serial line input using the appropriate serial line cable. Connect the trigger BNC plug to the 1401 digital output port 0 BNC socket found on the front of all modern types of 1401.

2.  In the outputs page of the sampling configuration dialog, make sure that digital output bit zero is enabled for use. Using the pulses configuration dialog set the initial level of digital output bit zero to 0 and place a pulse (which will be high-going) in the outputs at the time when you want the Magstim to fire. This output pulse should be at least 10 microseconds long – 1 millisecond works well.

3.  The sweep mode in the General section of the sampling configuration should be set to Fixed interval mode so that Signal controls the interval between sweeps and that this interval can be set long enough that the stimulator has enough time to recharge. Magstim stimulators can take quite a while to change the power levels, so you may get missed trigger messages at the end of sampling if you set the fixed interval less than about 10 seconds. The pulses configuration dialog is also used to set the sweep interval and optional random variation in the interval - the relevant values are found as part of the control track information.

4.  The Magstim support uses Signal multiple states sampling, which should be set up in dynamic outputs mode. You can use any number of extra states, each extra state providing separate Magstim settings. Each set of pulse outputs should include digital output pulses to trigger the Magstim along with any other outputs required. The Magstim support will work correctly with manual control of the states or with any style of automatic states sequencing including protocols.

When Signal begins sampling with the Magstim support enabled, it checks for a correctly functioning Magstim device as part of the process of initialising for sampling. If a Magstim is found then Signal will carry out the initial configuration of the Magstim and arm the device. While sampling is in progress, Signal will set up the Magstim using the current state data before each sweep, it will then delay the start of each sweep until the Magstim reports that it is armed and ready. At the end of each sweep the Magstim health is checked to make sure it is OK. Note that the checks on Magstim readiness can impose a significant extra inter-sweep delay though steps have been taken to minimise this. When sampling finishes normally, the Magstim is disarmed and remote control disabled.

If the Magstim coil temperature rises too high, Signal will stop sampling but not finish. Once the coil temperature has dropped sufficiently you can press 'More' on the sampling control panel to resume sampling again.

While sampling is in progress, Signal continuously maintains communications to prevent the Magstim from disabling remote control and disarming itself. If Signal ceases to communicate with the Magstim because it has encountered a significant problem ("crashes"), the Magstim will disarm itself automatically within 1 second, but this safety feature only applies if manual control has not been selected by Signal beforehand.

If manual control is selected, the Magstim will disarm itself spontaneously only after 60 seconds have passed without a stimulus trigger, so it is your responsibility to make sure the Magstim is disarmed if manual control is used and Signal encounters a significant problem. The Rapid stimulator does not appear to disarm itself after 60 seconds in this manner and must be disarmed manually if Signal fails during sampling.

Because Signal needs to communicate frequently with a Magstim to stop it disarming, scripts that operate while Signal is sampling need to be correctly designed. If a script carries out a lengthy operation without yielding or using a toolbar or dialog to allow control to pass back to the operating system, this may interfere with Magstim communications and cause the unit to disarm.

# MagPro device

The MagPro range of transcranial magnetic stimulators produced by MagVenture are widely used to provide non-invasive neuronal and muscular stimulation. Signal provides support for the MagPro R30 and X100 devices, both with and without the MagOption extension module that is available for either device. Signal uses a serial line for control of MagPro stimulators.

# MagPro safety notice

Transcranial magnetic stimulators are dangerous devices capable of causing serious harm and should only be used by qualified medical practitioners. Before using a MagPro device you should read the user's manual produced by MagVenture paying particular attention to the warnings and precautions section. It is your responsibility to ensure that Signal's control of a MagPro is set up in an appropriate and safe fashion for the intended use and to verify that it is operating correctly.

Signal carries out checks to ensure that the MagPro hardware is operating correctly. Firstly, if communications between Signal and the MagPro breaks down then Signal will disarm the MagPro and terminate sampling as unless the communications are working correctly you cannot be sure that the MagPro settings will be those that you have selected. Secondly, after sending new settings to a MagPro Signal always reads back the settings and disarms the MagPro and terminates sampling if they do not match the settings that were sent.

We must emphasise that CED is not responsible in any way for problems caused by use of Signal with MagPro equipment.

# MagPro introduction

For MagPro devices, the auxiliary states system sets the mode, waveform, power output and (where appropriate) inter-pulse timing and other settings, remote control of the hardware can be disabled for specific states to allow the user to control the stimulation manually. This documentation only describes the controls available from within Signal; for a complete understanding of the effect these have upon the behaviour of the stimulator you should consult the MagPro user manuals.

In addition to using a serial-line to set the stimulation parameters, standard operation with Signal requires that the stimulation be triggered using one or more TTL pulses generated by the 1401 digital outputs. This allows the pulse timing to be precisely controlled relative to the Signal sampling. Manual triggering of the stimulator is also possible, as is triggering from other external hardware, in which case you would normally use the MagPro trigger to trigger the Signal sampling sweep.

The MagPro support monitors the stimulator health; both waiting until the MagPro is ready for use before allowing a sampling sweep to proceed and terminating sampling if a hardware problem develops. To aid data analysis, Signal saves the MagPro stimulation parameters used in the data section variables of the sampled file, note that these values are saved even if the MagPro is manually controlled. The parameters saved are the device mode, the waveform used, the power levels, the pulse interval and the pulse count. These values are placed in extra frame variables in the sampled data file. The saved values can be retrieved using the script language and used as measurements in trend plot generation.

There is a Getting started with TMS experiments section in the Magstim support documentation; nearly all of this is equally applicable to MagPro devices.

# MagPro configuration

To configure a MagPro in Signal, open the sampling configuration dialog and make sure that Multiple states are enabled in the General section. Select the States section and make sure that you are in *Dynamic outputs* or *Static outputs* mode. If MagPro usage has been added to the sampling configuration, clicking on the Auxiliary >> button will show a MagPro item in the lower section of the menu. If this item is not present in the menu, use the Add device item at the top of the menu to add it. If the Auxiliary >> button is present, but disabled, you are in *External Digital* states mode, change the mode to enable the button. Click on any MagPro item in the lower section of the menu to open the MagPro configuration dialog for that device.

It is meaningful to use the MagPro support without using multiple states, but this is not directly supported because the states system is so closely linked to the MagPro support. If you want to make use of MagPro support without using multiple states (so that the pulse parameters are recorded and MagPro health checks are made, but with complete manual control) you should turn on multiple states, set only one extra state, set state zero to use manual control, and only make use of state zero while sampling. This will allow you access to the MagPro configuration dialog to set the basic MagPro parameters.

### Label
At the top of the configuration dialog there is a field where you can enter a device label used to distinguish the device from others.

### Type of stimulator
Below the label there is a selector used to select the MagPro type. You should set this to the type of stimulator you will be using (the choices are MagPro R30, MagPro R30 + MagOption, MagPro X100, MagPro X100 + MagOption) or to Do not use if you do not want to make use of the MagPro support in this sampling configuration.

### Don't generate settings text for window title

This check box prevents the MagPro support from generating text showing the current stimulus that Signal can display in the title bar of a sampling document. It is often useful to see what the MagPro settings are, but some users find it distracting. A separate check box in the sampling preferences must also be set to allow this text to be shown.

### Assume MagPro software of version 7.1.0 or more

This check box, if checked, allows the configuration dialog to assume that the internal MagPro software is new enough to provide the advanced options available with this release. These options include changing the mode or waveform, and controlling the inter-pulse interval and Power B/A values in Twin mode. The setting is checked against the actual software version when the sampling configuration is used, if you have set this option and the actual software version is not new enough then sampling will be aborted.

### Com port for communications

This is a selector for the COM port used to control the stimulator, you can select any of the PC COM ports that are detected as available. As many PCs only have one COM port you may have to use a USB serial port expander to provide a spare COM port – we have found that most of these USB serial ports work satisfactorily.

### Powers as % of RMT

When this check box is set, the power values entered for the individual states are percentages of the subject resting motor threshold (RMT) rather than a percentage of the overall device power (when the check box is clear). The RMT value in use, which is always a percentage of the overall device power, is entered to the right of the check box.

Following these items are two sections holding the actual MagPro settings; first there is a display of settings for all states which is used to select a state for editing (by clicking on the information for that state) and then an area where the settings for the selected state may be edited. The available settings are:

### Mode

This selects between the various stimulator modes. The available modes for the various types of device are (consult your MagPro documentation for details of these):

| | Standard | Power | Twin | Dual |
|---|---|---|---|---|
| **R30** | Available | | | |
| **R30+MagOption** | Available | | Available | Available |
| **X100** | Available | | | |
| **X100+MagOption** | Available | Available | Available | Available |

### Waveform

This selects between the various types of stimulation waveform that are available. The available waveforms for the various types of device are (consult your MagPro documentation for details of these):

| | Monophasic | Biphasic | Halfsine | Biphasic Burst |
|---|---|---|---|---|
| **R30** | | Available | | |
| **R30+MagOption** | Available | Available | | |
| **X100** | Available | Available | | Available |
| **X100+MagOption** | Available | Available | Available | Available |

### Manual

Set this check box for this state to be controlled by the manual controls and not by Signal, clear it to give Signal control.

**Reverse**

This check box, available for X100 systems only, selects a reversed waveform if checked. This field is disabled for manually controlled states.

**Power A (%)**

This sets the stimulator power output as a percentage of the maximum available, you can set any value from 0 to 100. If device power is being specified as a percentage of the RMT, you can set any value from 0 to 100/RMT. This field is disabled for manually controlled states.

**Interval (ms)**

This field, available in Twin and Dual modes only, sets the interval between the two pulses, you can use any value from 1 to 3000 milliseconds. This field is disabled for manually controlled states.

**Power B (%)**

This item, available in Dual mode only, sets the secondary stimulator power output as a percentage of the maximum available, you can set any value from 0 to 100. If device power is being specified as a percentage of the RMT, you can set any value from 0 to 100/RMT. This field is disabled for manually controlled states.

**Power (B/A)**

This item, available in Twin mode only, sets the secondary stimulator power output as a fraction of the main (A) power, you can set any value from 0.2 to 5. This field is disabled for manually controlled states.

**Charge delay (ms)**

This item, available with all devices and modes, sets the delay after the stimulator fires before it begins to recharge again, you can use this to delay charging (which can generate some noise) until after any evoked responses that you may be interested-in have occurred. This field is disabled for manually controlled states.

**Copy to ...**

The button labelled Copy to ... provides a simple dialog that can be used to set up many states quickly. If you press OK it copies the currently selected state to all the other states that are within the selected range. This button is available for all types of MagPro.
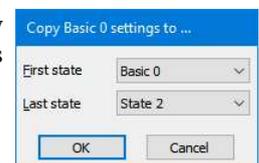
**Test**

The button labelled Test tests if communications with the MagPro can be established. It checks that you have the correct COM port, that the serial line is connected and that the MagPro is communicating correctly with Signal. It also detects if you have set an incorrect MagPro device type. The Test button is available for all types of MagPro.

# MagPro connections and cabling

There are two connectors provided by MagPro for serial line control and external triggering of the unit. The serial line connector is a standard PC-style connector, the COM2 input on the MagPro is the one used for control by Signal, a null modem cable should be used between the MagPro and the computer running Signal as shown below. A separate DSUB 9 pin female connector is used for the trigger connector, connections for this are shown below.

If your PC does not provide a serial line (a common situation with laptops) or no unused serial line connector is available a USB serial line adapter can be used instead, but these can give problems so we advise using standard serial port hardware if this is possible.

Two separate connections need to be made; serial line control from a PC and a TTL trigger signal from the 1401. The standard high level trigger input suggested by MagPro works well with Signal and the 1401 and is used in the diagrams shown below, consult the MagPro documentation if you require a different arrangement.

**Cable for serial line connection**
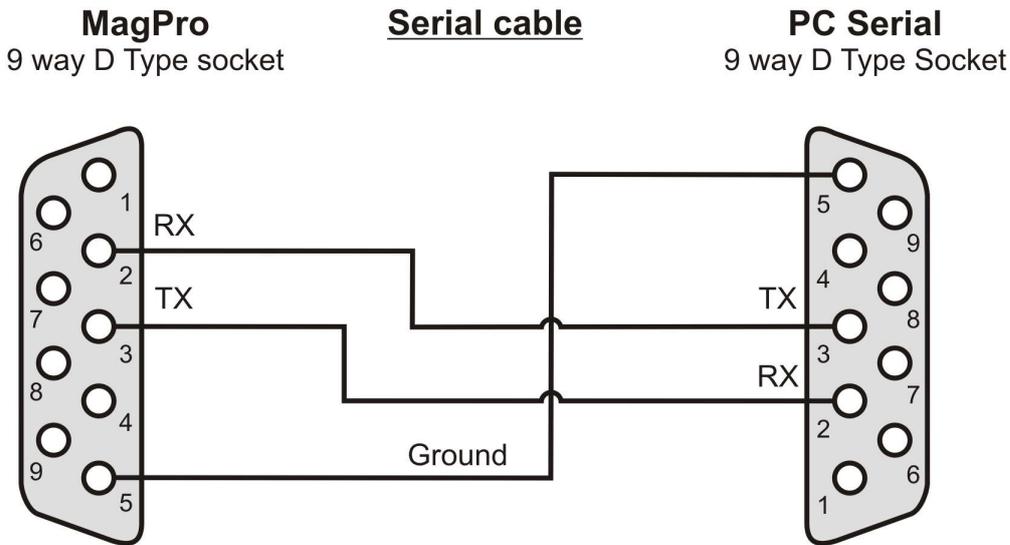


**Cable for DSUB 9 trigger connector**
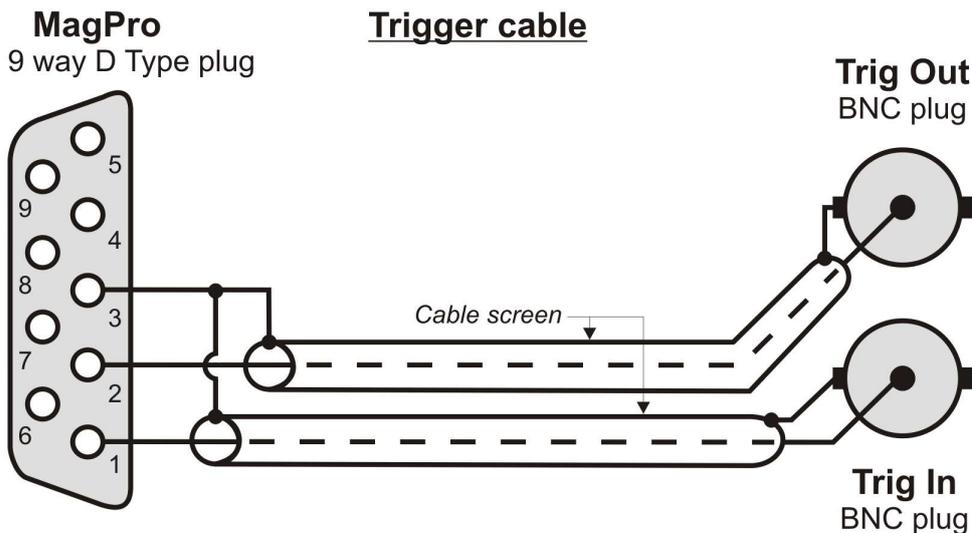


To trigger the MagPro from the 1401 you only need the Trig In part of the cabling; connect the Trig In BNC to the 1401 digital output zero connector and generate trigger pulses on that output. If you are using the MagPro to trigger 1401 data acquisition you will need the Trig Out part of the cabling instead; connect the Trig Out BNC to the 1401 trigger input (we recommend configuring the MagPro trigger out as a low-going signal but either is usable with suitable 1401 trigger configuration) to trigger the 1401 sweep. Consult your MagPro documentation for more information.

# Notes on MagPro use

A straightforward arrangement to use a MagPro with Signal would be as follows:

1. Connect the serial port of your computer to the MagPro serial line input using a null serial line cable. Connect pin 1 of the DSUB 9 pin female connector to the 1401 digital output port 0 BNC socket found on the front of all modern types of 1401.

2. In the outputs page of the sampling configuration, make sure that digital output bit zero is enabled for use. Using the pulses configuration dialog set the initial level of digital output bit zero to 1 and place a pulse (which

will be low-going) in the outputs at the time when you want the MagPro to fire. This output pulse should be at least 50 microseconds long – 1 millisecond works well.

3. The sweep mode in the General section of the sampling configuration should be set to Fixed interval mode so that Signal controls the interval between sweeps and that this interval can be set long enough that the stimulator has enough time to recharge. The pulses configuration dialog is also used to set the sweep interval and optional random variation in the interval - the relevant values are found as part of the control track information.

4. The MagPro support uses Signal multiple states sampling, which should be set up in dynamic outputs mode. You can use any number of extra states, each extra state providing separate MagPro settings. Each set of pulse outputs should include digital output pulses to trigger the MagPro along with any other outputs required. The MagPro support will work correctly with manual control of the states or with any style of automatic states sequencing including protocols.

When Signal begins sampling with the MagPro support enabled, it checks for a correctly functioning MagPro device as part of the process of initialising for sampling. If a MagPro is found and enabled then Signal will carry out the initial configuration of the MagPro. While sampling is in progress, Signal will set up the MagPro using the current state data before each sweep, it will then delay the start of each sweep until the MagPro reports that it is armed and ready. At the end of each sweep the MagPro health is checked to make sure it is OK. Note that the checks on MagPro readiness can impose a significant extra inter-sweep delay. In particular, switching to or from from Power mode can take up to ten seconds.

If the MagPro coil temperature rises too high, the unit will disable itself and Signal will stop sampling. Once the coil temperature has dropped sufficiently you can press 'More' on the sampling control panel to resume sampling again.

If the MagPro is not triggered for more than a certain amount of time (five minutes by default) then the MagPro will disable itself and sampling will stop.

If manual control is selected, the MagPro will fire on the trigger with whatever settings it has been given using the front panel or with whatever settings it had prior manual mode being used.

# PowerMAG device

The PowerMAG transcranial magnetic stimulator produced by MAG & More are widely used to provide non-invasive neuronal and muscular stimulation. Signal provides support for the PowerMAG ppTMS device as an auxiliary states device using a USB connection acting as a serial line for control of the stimulator.

# PowerMAG safety notice

Transcranial magnetic stimulators are dangerous devices capable of causing serious harm and should only be used by qualified medical practitioners. Before using a PowerMAG device you should read the user's manual produced by MAG & More paying particular attention to the warnings and precautions section. It is your responsibility to ensure that Signal's control of a PowerMAG is set up in an appropriate and safe fashion for the intended use and to verify that it is operating correctly.

Signal carries out checks to ensure that the PowerMAG hardware is operating correctly. Firstly, if communications between Signal and the PowerMAG breaks down then Signal will disarm the PowerMAG and terminate sampling as unless the communications are working correctly you cannot be sure that the PowerMAG settings will be those that you have selected. Secondly, after sending new settings to a PowerMAG Signal always checks for a correct response and terminates sampling if is not received.

We must emphasise that CED is not responsible in any way for problems caused by use of Signal with PowerMAG equipment.

# PowerMAG introduction

The Signal auxiliary states system operates with a  PowerMAG pp-TMS stimulator, but only in the pp-TMS mode as this allows suitable software control of the stimulator behaviour. The externally triggered (via the splitter box), single pulse, r-TMS mode of operation is not supported by the auxiliary states system but it is perfectly possible for you to make use of this mode with Signal by connecting 1401 DACs to the splitter box and setting the DAC outputs to suitable levels for each sampling state.

The PowerMAG pp-TMS stimulator is designed to be more independent of the controlling software in Signal than most TMS devices. Rather than being programmed with power levels and other stimulus parameters and then being triggered by a pulse generated by the 1401, it is programmed with power levels and settings that define a sequence of stimulations, and then automatically executes this sequence when commanded. Because of this style of usage, it is important to make sure that the PowerMAG stimulus sequence matches the states sequencing that is set in Signal. This documentation only describes the controls available from within Signal; for a complete understanding of the behaviour of the stimulator you should consult the PowerMAG user manuals.

In addition to using a USB serial-line to set the stimulation parameters, operation with Signal requires that the sampling sweeps are triggered by TTL pulses generated by the stimulator. This ensures that data acquisition is precisely timed relative to the stimulus generation.

The PowerMAG support monitors the stimulator health; both waiting until the PowerMAG is ready for use before allowing sampling to proceed and terminating sampling if a hardware problem develops. To aid data analysis, Signal saves the PowerMAG stimulation parameters used in the data section variables of the sampled file. The parameters saved are the the two power levels only, these are placed in extra frame variables in the sampled data file. The saved values can be retrieved using the script language and used as measurements in trend plot generation.

There is a Getting started with TMS experiments section in the Magstim support documentation; most of this is applicable to PowerMAG devices.

# PowerMAG configuration

To configure a PowerMAG device in Signal, open the sampling configuration dialog and make sure that Multiple states are enabled in the General section. Select the States section and make sure that you are in *Dynamic outputs* or *Static outputs* mode. If PowerMAG usage has been added to the sampling configuration, clicking on the Auxiliary >> button will show a PowerMAG item in the lower section of the menu. If this item is not present in the menu, use the Add device item at the top of the menu to add it. If the Auxiliary >> button is present, but disabled, you are in *External Digital* states mode, change the mode to enable the button. Click on any PowerMAG item in the lower section of the menu to open the PowerMAG configuration dialog for that device.



It is meaningful to use the PowerMAG support without using multiple states, but this is not directly supported because the states system is so closely linked to the PowerMAG support. If you want to make use of PowerMAG support without using multiple states (so that the pulse parameters are recorded and PowerMAG health checks are made, but without variations in the stimulus) you should turn on multiple states, set only one extra state, set state zero to the stimulus parameters that you want, and only make use of state zero while sampling.

### Label
At the top of the configuration dialog there is a field where you can enter a device label used to distinguish the device from others.

### Type of stimulator
Below the label there is a selector used to select the PowerMAG type. You can set this to PowerMAG - ppTMS to use the PowerMAG stimulator or to Do not use if you do not want to make use of the PowerMAG support in this sampling configuration. Note that there are a number of other types of PowerMAG stimulator available from MAG and More but it is only possible to control the ppTMS variant from Signal.

### Don't generate settings text for window title
This check box prevents the PowerMAG support from generating text showing the current stimulus that Signal can display in the title bar of a sampling document. It is often useful to see what the PowerMAG settings are, but some users find it distracting. A separate check box in the sampling preferences must also be set to allow this text to be shown.

### Com port for communications

This is a selector for the COM port used to control the stimulator, all of the PC COM ports that are detected as available are displayed. You should select the port corresponding to the PowerMAG USB connection, which will most often be the highest-numbered COM port.

### Powers as % of RMT

When this check box is set, the power values entered for the individual states are percentages of the subject resting motor threshold (RMT) rather than a percentage of the overall device power (when the check box is clear). The RMT value in use, which is always a percentage of the overall device power, is entered to the right of the check box.

Following these items are three sections holding the actual PowerMAG settings; first there is a display of settings for all states which is used to select a state for editing (by clicking on the information for that state), then the two power levels that are used in that state and finally a list showing the stimulus sequence for the selected state. The available settings are:

### Power S1 (%)

This item sets the first stimulus power output as a percentage of the maximum available, you can set any value from 0 to 100. If device power is being specified as a percentage of the RMT, you can set any value from 0 to 100/RMT.

### Power S2 (%)

This item sets the second stimulus power output as a percentage of the maximum available, you can set any value from 0 to 100. If device power is being specified as a percentage of the RMT, you can set any value from 0 to 100/RMT.

### Stimulus sequence

Each state has a stimulus sequence defined consisting of a number of stimulus trains. Each train has an inter-stimulus interval (ISI) that sets the interval between the first and second stimulus, an inter-train interval (ITI) that sets the interval between stimuli within the train, an amount of random variation in the ITI and a stimulus repeat count, which sets the number of paired stimulations that will occur. You can include a large number of trains in one stimulus sequence. The stimulus sequence is set up and initiated whenever the Signal sampling state is changed and then executes automatically.

Use the Add button to add another train to the end of a sequence, and the Insert button to add a new train into the sequence before the currently selected train. Use the Edit button, or double click on an item, to edit the currently selected train settings, and use Delete to delete the currently selected train.

### Copy to ...

The button labelled Copy to ... provides a simple dialog that can be used to set up many states quickly. If you press OK it copies the currently selected state to all the other states that are within the selected range. This button is available for all types of PowerMag.

### Test

The button labelled Test checks if communications with the PowerMAG can be established. It checks that you have selected the correct COM port and that the PowerMAG is communicating correctly with Signal.

# PowerMAG connections and cabling

The PowerMAG stimulator uses a standard USB connector for external control; this should be connected with an ordinary USB cable to a spare USB connector on the PC, where it will appear as a COM port.

Separately, you need to trigger Signal sampling sweeps at the time of the stimulation. This requires a standard BNC cable which runs from the trigger-out connector on the rear of the PowerMAG to the Trigger input

connector on the front of the 1401. Note that the TTL pulses generated by the PowerMAG are active-high, so you need to make sure the Rising edge trigger check box in the sampling configuration general section is set.

# Notes on PowerMAG use

### Relationship between Signal states sequencing and the PowerMAG configuration

When using a PowerMAG stimulator with Signal, the settings for each state define a sequence of stimulations with varying timing as defined by the user, this sequence is initiated when the Signal sampling state changes. All of the stimulations within a given state use the same power level, so you should only need a different Signal sampling state when you need to change the power levels. There is an extra timing delay introduced by reprogramming the PowerMAG when the sweep changes so in general you should try to avoid unnecessary state changes. However this may not be straightforward.

Each stimulation generates a TTL output pulse which is used to trigger the 1401 and cause a sampling sweep to be taken, so when the state changes a preset number of trigger pulses will be generated (as set by the PowerMAG configuration), after which no more stimuli or trigger pulses will occur until the PowerMAG is reprogrammed with a new sequence of stimuli. This means that your design of the stimuli generated by the PowerMAG <u>must</u> match the pattern of states sequencing set within Signal, or data acquisition will 'hang' waiting for sweep triggers that never occur.

Consider for example a sampling configuration that uses Numeric states sequencing, with two repeats of each state. This means that the PowerMAG stimulation sequence that is defined for each state <u>must contain at least two stimuli</u>. If, for example, the settings for state 4 only cause a single stimulus to occur then when sampling switches to state 4 it will set up the stimulator and then start to wait for sweeps of data. The first sweep will be triggered without problems, but the trigger for the second sweep will not occur and sampling will hang while waiting for the second sweep for state 4 to occur. So it is <u>very important</u> that you bear this restriction in mind while designing your experiments.

A straightforward arrangement to use a PowerMAG with Signal would be as follows:

1. Connect a spare USB port on your computer to the PowerMAG using a standard USB cable. Connect the trigger-out connector on the rear of the PowerMAG to the Trigger input connector on the front of the 1401.

2. The sweep mode in the General section of the sampling configuration should probably be set to Peri-triggered so that Signal collects some data before the first stimulus (though Basic mode can be used if you don't care about pre-trigger data) and the Rising edge trigger checkbox should be set.

3. The PowerMAG support uses Signal multiple states sampling, which should be set up in dynamic outputs mode. You can use any number of extra states, each extra state providing separate PowerMAG settings. The PowerMAG support will work correctly with manual control of the states or with any style of automatic states sequencing including protocols but, as described above, it is important to ensure that the number of times each state is used in succession matches the number of stimuli that are generated by the PowerMAG settings for that state. This may make it difficult to fully take advantage of the flexibility offered by Protocol sequencing.

When Signal begins sampling with the PowerMAG support enabled, it checks for a correctly functioning PowerMAG device as part of the process of initialising for sampling. If a PowerMAG is found and enabled then Signal will carry out the initial configuration of the PowerMAG. While sampling is in progress, Signal will set up the PowerMAG using the current state data for each state. At the end of each sweep the PowerMAG health is checked to make sure it is OK.

If the PowerMAG coil temperature rises too high, the unit will disable itself and Signal will stop sampling.

# DuoMAG device

The DuoMAG transcranial magnetic stimulators produced by Deymed are widely used to provide non-invasive neuronal and muscular stimulation. Signal provides support for DuoMAG devices as an auxiliary states device using a USB connection for control of the stimulator. Signal can control either a single DuoMag XT or MP or a pair of DuoMAG MP stimulators as a single auxiliary states device.

# DuoMAG safety notice

Transcranial magnetic stimulators are dangerous devices capable of causing serious harm and should only be used by qualified medical practitioners. Before using a DuoMAG device you should read the user's manual produced by Deymed paying particular attention to the warnings and precautions section. It is your responsibility to ensure that Signal's control of a DuoMAG is set up in an appropriate and safe fashion for the intended use and to verify that it is operating correctly.

Signal carries out checks to ensure that the DuoMAG hardware is operating correctly. Firstly, if communications between Signal and the DuoMAG breaks down then Signal will disarm the DuoMAG and terminate sampling as unless the communications are working correctly you cannot be sure that the DuoMAG settings will be those that you have selected. Secondly, after sending new settings to a DuoMAG Signal always reads back the settings and disarms the DuoMAG and terminates sampling if they do not match the settings that were sent.

We must emphasise that CED is not responsible in any way for problems caused by use of Signal with DuoMAG equipment.

# DuoMAG introduction

For DuoMAG devices, the auxiliary states system sets the mode, power output and (where appropriate) inter-pulse timing and other settings, remote control of the stimulation power can be disabled for specific states to allow the user to control the power manually. This documentation only describes the controls available from within Signal; for a complete understanding of the effect these have upon the behaviour of the stimulator you should consult the DuoMAG user manuals.

In addition to using a USB connection to set the stimulation parameters, standard operation with Signal requires that the stimulation be triggered using one or more TTL pulses generated by the 1401 digital outputs. This allows the pulse timing to be precisely controlled relative to the Signal sampling. Manual triggering of the stimulator is also possible, as is triggering from other external hardware, in which case you would normally use the DuoMAG trigger to trigger the Signal sampling sweep.

The DuoMAG support monitors the stimulator health; both waiting until the DuoMAG is ready for use before allowing a sampling sweep to proceed and terminating sampling if a hardware problem develops. To aid data analysis, Signal saves the DuoMAG stimulation parameters used in the data section variables of the sampled file, note that these values are saved even if the DuoMAG is manually controlled. The parameters saved are the the power levels, the recharge delays and the TTL output delay. These values are placed in extra frame variables in the sampled data file. The saved values can be retrieved using the script language and used as measurements in trend plot generation.

There is a Getting started with TMS experiments section in the Magstim support documentation; nearly all of this is equally applicable to DuoMAG devices.

# DuoMAG configuration

To configure a DuoMAG stimulator in Signal, open the sampling configuration dialog and make sure that Multiple states are enabled in the General section. Select the States section and make sure that you are in *Dynamic outputs* or *Static outputs* mode. If DuoMAG usage has been added to the sampling configuration, clicking on the Auxiliary >> button will show a DuoMAG item in the lower section of the menu. If this item is not present in the menu, use the Add device item at the top of the menu to add it. If the Auxiliary >> button is present, but disabled, you are in *External Digital* states mode, change the mode to enable the button. Click on any DuoMAG item in the lower section of the menu to open the DuoMAG configuration dialog for that device.

It is meaningful to use the DuoMAG support without using multiple states, but this is not directly supported because the states system is so closely linked to the DuoMAG support. If you want to make use of DuoMAG support without using multiple states (so that the pulse parameters are recorded and DuoMAG health checks are made, but with complete manual control) you should turn on multiple states, set only one extra state, set state zero to use manual control, and only make use of state zero while sampling. This will allow you access to the DuoMAG configuration dialog to set the basic DuoMAG parameters.

### Label
At the top of the configuration dialog there is a field where you can enter a device label used to distinguish the device from others.

### Type of stimulator
Below the label there is a selector used to select the DuoMAG type. You should set this to the type of stimulator you will be using (the choices are DuoMAG XT / MP or Dual DuoMAG MP) or to Do not use if you do not want to make use of the DuoMAG support in this sampling configuration.

### Don't generate settings text for window title
This check box prevents the DuoMAG support from generating text showing the current stimulus that Signal can display in the title bar of a sampling document. It is often useful to see what the DuoMAG settings are, but some users find it distracting. A separate check box in the sampling preferences must also be set to allow this text to be shown.

### Stimulator-dependent controls
Below these basic stimulator configuration controls there may be more controls, these will vary according to the type of stimulator selected and will be described in the specific device configuration details (see below).

### DuoMAG settings
Following these are two more sections holding the actual DuoMAG settings; first there is a display of settings across all states which is used to select a state for editing (by clicking on the information for that state) and finally an area where the settings for the selected state may be edited. Again, the information shown here will vary according to the type of stimulator selected and is described in the separate specific device configuration details (see below).

**Copy to ...**

The button labelled Copy to ... provides a simple dialog that can be used to set up many states quickly. If you press OK it copies the currently selected state to all the other states that are within the selected range. This button is available for all types of DuoMAG stimulators.
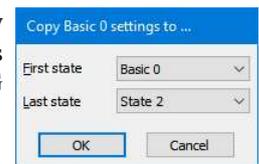
**Test**

The button labelled Test checks if communications with the DuoMAG can be established and displays information about what it has found, it also detects if you are using a DuoMAG XP and have selected dual MP devices. The Test button is available for all types of DuoMAG stimulators.

# XT / MP device configuration

The DuoMAG XT and MP devices are configured in the same manner:

**Stimulator selector**

When more than one DuoMag device is being used it is necessary to specify which device is controlled by which settings. This selector displays the serial numbers of all detected DuoMAG devices. Normally there will be just one and Signal will detect this and automatically set the selector. If you are using multiple devices you will have to use the selectors to set the device assignment as you wish.

**Powers as % of RMT**

When this check box is set, the power values entered for the individual states are percentages of the subject resting motor threshold (RMT) rather than a percentage of the overall device power (when the check box is clear). The RMT value in use, which is always a percentage of the overall device power, is entered to the right of the check box.

The settings available for the individual states are:

**Manual power**

Set this check box for the power level for this state to be controlled by the control wheel on the coil and not by Signal, clear it to give Signal control.

**No triggers**

Set this check box to indicate to Signal that the stimulator will not be triggered for this state. This will prevent Signal complaining about no trigger pulse being generated.

**Power (%)**

This sets the stimulator power output as a percentage of the maximum available, you can set any value from 0 to 100. This field is disabled for manually controlled states. If device power is being specified as a percentage of the RMT, you can set any value from 0 to 100/RMT.

**Recharge delay (ms)**

This sets the delay after the stimulator fires before it begins to recharge, you can use this to delay recharging (which can generate some noise) until after any evoked responses that you may be interested-in have occurred. You can enter any delay value between 0 and 600 milliseconds. Note that this field is not disabled for states with manually controlled power; the delay is always controlled by Signal.
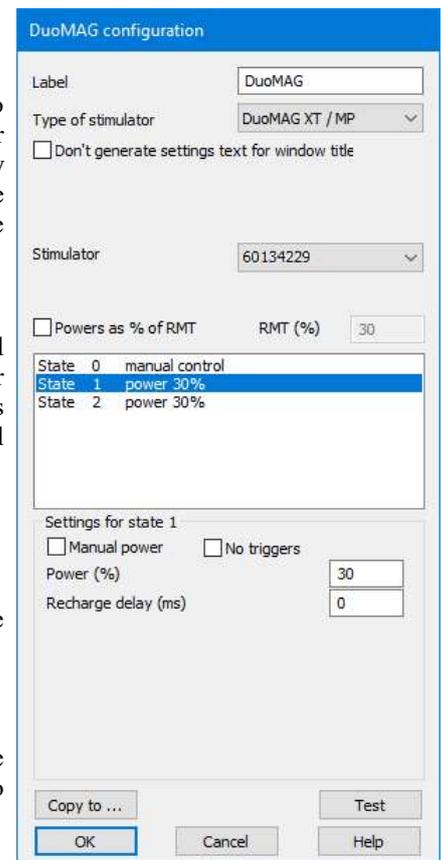
# Dual MP device configuration

Two DuoMAG MP devices can be controlled by Signal to allow generation of paired stimulation pulses with a set interval between the stimulations. When using this device configuration two MP devices must be connected to the PC, stimulus triggering can either be done using a delayed trigger generated by the primary MP device or the two devices can be independently triggered using two digital outputs on the 1401. When this device type is selected extra device configuration controls are displayed:

### Independent triggers

With dual MP devices there are two ways that the stimulation triggering can be carried out. With internal triggering the primary MP device is triggered by a digital output from the 1401 and it then triggers the secondary device via a TTL output pulse occurring after a set delay. When Independent triggers are used the two devices are separately triggered using two 1401 digital outputs. Independent triggers mode is recommended as this ensures that the stimulation timing is wholly accurate. Note that when independent triggers mode is used you must make sure that the TTL output of the primary device is **NOT** connected to the secondary device otherwise the stimulation timing will be incorrect and significant damage could be caused to the equipment. When independent triggers mode is not used the TTL output of the primary device **MUST** be connected to the secondary device otherwise the secondary stimulation will not be generated.

### Primary and secondary device selector

When using two MP devices it is necessary to specify which MP device is the primary device (the one which fires first) and which is the secondary so that Signal knows which settings to use for which device. The two selectors display the serial numbers of all detected DuoMAG devices. Normally, the secondary device will have an 's' character at the end of its serial number and Signal will detect this and automatically set up the correct primary and secondary devices. In some situations it may be necessary to use the selectors to adjust the device assignment manually.

### Powers as % of RMT

When this check box is set, the power values entered for the individual states are percentages of the subject resting motor threshold (RMT) rather than a percentage of the overall device power (when the check box is clear). The RMT value in use, which is always a percentage of the overall device power, is entered to the right of the check box.

The settings available for the individual states are:

### Manual power

Set this check box for the power levels for this state to be controlled by the manual stimulator controls and not by Signal, clear it to give Signal control.

### No triggers

Set this check box to indicate to Signal that the stimulator will not be triggered for this state. This will prevent Signal complaining about no trigger pulse being generated.

### Power (%)

This sets the primary stimulator power output as a percentage of the maximum available, you can set any value from 0 to 100. If device power is being specified as a percentage of the RMT, you can set any value from 0 to 100/RMT. This field is disabled for manually controlled states.

### Recharge delay (ms)

This sets the delay after the stimulator fires before it begins to recharge, you can use this to delay recharging (which can generate some noise) until after any evoked responses that you may be interested-in have occurred. You

can enter any delay value between 0 and 600 milliseconds. Note that this field is not disabled for states with manually controlled power; the delay is always controlled by Signal.

### Manual trigger

When using Independent triggers mode the normal arrangement is to trigger each stimulator using a separate TTL output pulse generated by the 1401. When this check box is set, a different arrangement is used where the primary stimulator is triggered manually, the Signal sweep is triggered by the TTL output of the primary stimulator (automatically programmed to occur with zero delay), and a TTL pulse to trigger the secondary stimulator is generated by the 1401 at a set time within the sampling sweep. This field is hidden if Independent triggers mode is not in use.

### TTL out delay (ms)

When not using Independent triggers mode, this sets the delay after the primary stimulator fires before it generates the TTL output pulse to trigger the secondary stimulator from 1 to 600 milliseconds (with a time resolution of 0.1 milliseconds). This field is hidden if Independent triggers mode is in use.

### Digital output used to trigger

When using Independent triggers (not using the manual trigger) mode, this sets the 1401 digital output used to trigger the primary device, this information is needed so that the interval between the trigger pulses can be checked to make sure it is long enough to avoid damage to the coil. This field is hidden if Independent triggers mode is not in use.

### Power 2 (%)

This sets the secondary device power output as a percentage of the maximum available, you can set any value from 0 to 100. If device power is being specified as a percentage of the RMT, you can set any value from 0 to 100/RMT. This field is disabled for manually controlled states.

### Recharge delay 2 (ms)

This sets the delay after the stimulator fires before it begins to recharge, you can use this to delay recharging (which can generate some noise) until after any evoked responses that you may be interested-in have occurred. You can enter any delay value between 0 and 600 milliseconds. Note that this field is not disabled for states with manually controlled power; the delay is always controlled by Signal.

### Digital output used to trigger

When using Independent triggers (not using the manual trigger) mode, this sets the 1401 digital output used to trigger the primary device, this information is needed so that the interval between the trigger pulses can be checked to make sure it is long enough to avoid damage to the coil. This field is hidden if Independent triggers mode is not in use.

# DuoMAG connections and cabling

Each DuoMAG stimulator is controlled using a standard USB connection. It is necessary to install the necessary drivers for the USB interface chips used, these will normally be installed as part of the standard Deymed software installation process. If necessary, the correct drivers may be downloaded from the FTDI website here: http://www.ftdichip.com/Drivers/D2XX.htm.

### Connections for triggering

Each DuoMAG stimulator has separate trigger input and trigger output connections, the trigger input to the DuoMAG is a BNC socket labelled TTL IN, the output is a BNC socket labelled TTL OUT. Both of these connections use standard TTL signals suitable for direct connection to the 1401 digital outputs and inputs.

#### *Single stimulator*

With a single stimulator in use the most common arrangement would be to have the stimulator triggered by the 1401, with a single BNC cable connecting a 1401 digital output to the TTL IN socket on the stimulator. A high-going TTL pulse (10 milliseconds would be a suitable length) should be generated on the appropriate digital output at the time that you wish the stimulator to fire, this pulse should be present in the outputs for all states.

A possible alternative arrangement would be to trigger the stimulator manually and use this to trigger the sampling sweep, using a single BNC cable connecting the stimulator TTL OUT socket to the trigger input on the 1401. As the stimulator generates a high-going TTL pulse when it fires, the Rising edge trigger check box in the general page of the sampling configuration should be set. Peri-triggered sampling mode allows the capture of pre-trigger data and is thus most suitable for this style of use.

### Dual MP stimulators, Independent triggers

This arrangement is very similar to a single stimulator triggered by the 1401, but you now have two BNC cables connecting two 1401 digital outputs to the TTL IN socket on both stimulators. A high-going TTL pulse (10 milliseconds would be a suitable length) should be generated on each digital output at the time that you wish the stimulators to fire, these pulses would normally be present in the outputs for all states but with varying timing to give you varying inter-stimulation intervals. There is no reason why a trigger pulse cannot be omitted on some states to give you a single stimulation.

A possible alternative arrangement here is to set the Manual trigger check box and trigger the primary stimulator manually and use this to trigger the sampling sweep, using a single BNC cable connecting the stimulator TTL OUT socket to the trigger input on the 1401. A second BNC cable connects a 1401 digital output to the TTL IN socket on the secondary stimulator, and you must generate a TTL pulse on the 1401 digital output at the time within the sweep where you want the secondary stimulus to occur (the primary stimulus will of course occur at time zero). As the stimulator generates a high-going TTL pulse when it fires, the Rising edge trigger check box in the general page of the sampling configuration should be set. Peri-triggered sampling mode allows the capture of pre-trigger data and is thus most suitable for this style of use.

### Dual MP stimulators, Internal triggers

With this arrangement the primary stimulator is triggered by the 1401, but the secondary stimulator is triggered by the primary stimulator. There are two BNC cables, the first connects a 1401 digital output to the TTL IN socket on the primary stimulator, the second connects the primary TTL OUT socket to the secondary stimulator TTL IN socket. A high-going TTL pulse should be generated on the appropriate digital output at the time that you wish the primary stimulator to fire, the primary stimulator then triggers the secondary after the preset delay.

## Notes on DuoMAG use

A straightforward arrangement to use a DuoMAG with Signal would be as follows:

1.  Connect the DuoMAG to your computer using a standard USB cable. Using a standard BNC cable, connect the TTL IN socket on the stimulator to the 1401 digital output port 0 BNC socket found on the front of all modern types of 1401.

2.  In the outputs page of the sampling configuration, make sure that digital output bit zero is enabled for use. Using the pulses configuration dialog set the initial level of digital output bit zero to 1 and place a pulse (which will be low-going) in the outputs at the time when you want the DuoMAG to fire. This output pulse should be at least 50 microseconds long – 1 millisecond works well.

3.  The sweep mode in the General section of the sampling configuration should be set to Fixed interval mode so that Signal controls the interval between sweeps and that this interval can be set long enough that the stimulator has enough time to recharge. The pulses configuration dialog is also used to set the sweep interval and optional random variation in the interval - the relevant values are found as part of the control track information.

4.  The DuoMAG support uses Signal multiple states sampling, which should be set up in dynamic outputs mode. You can use any number of extra states, each extra state providing separate DuoMAG settings. Each set of pulse outputs should include digital output pulses to trigger the DuoMAG along with any other outputs required. The DuoMAG support will work correctly with manual control of the states or with any style of automatic states sequencing including protocols.

When Signal begins sampling with the DuoMAG support enabled, it checks for a correctly functioning DuoMAG device as part of the process of initialising for sampling. If a DuoMAG is found and enabled then Signal will carry out the initial configuration of the DuoMAG. While sampling is in progress, Signal will set up the DuoMAG using the current state data before each sweep, it will then delay the start of each sweep until the DuoMAG reports that it is armed and ready. At the end of each sweep the DuoMAG health is checked to make sure it is OK. Note that the checks on DuoMAG readiness can impose a significant extra inter-sweep delay.

If the DuoMAG coil temperature rises too high, the unit will disable itself and Signal will stop sampling. Once the coil temperature has dropped sufficiently you can press 'More' on the sampling control panel to resume sampling again.

If manual control is selected, the DuoMAG will fire on the trigger with whatever power settings it has been given using the coil control wheel or with whatever settings it had prior to manual mode being used.

# Neurosoft device

The Neuro-MS transcranial magnetic stimulators produced by Neurosoft Ltd. are widely used to provide non-invasive neuronal and muscular stimulation. Signal provides support for Neuro-MS devices as an auxiliary states device using a USB connection for control of the stimulator. Signal can control a single Neuro-MS in Monophasic mode, two paired Neuro-MS stimulators as a single auxiliary states device in Paired Monophasic mode, or multiple Neuro-MS devices may be controlled independently as separate devices.

# Neurosoft safety notice

- Transcranial magnetic stimulators are dangerous devices capable of causing serious harm and should only be used by qualified medical practitioners. Before using a Neuro-MS device you should read the user's manual produced by Neurosoft Ltd. paying particular attention to the warnings and precautions section. It is your responsibility to ensure that Signal's control of a Neuro-MS device is set up in an appropriate and safe fashion for the intended use and to verify that it is operating correctly.

Signal carries out checks to ensure that the Neurosoft hardware is operating correctly. Firstly, if communications between Signal and the Neuro-MS breaks down then Signal will disarm the Neuro-MS and terminate sampling as unless the communications are working correctly you cannot be sure that the Neuro-MS settings will be those that you have selected. Secondly, after sending new settings to a Neuro-MS Signal always reads back the settings and disarms the Neuro-MS and terminates sampling if they do not match the settings that were sent.

We must emphasise that CED is not responsible in any way for problems caused by use of Signal with Neurosoft equipment.

# Neurosoft introduction

For Neuro-MS devices, the auxiliary states system sets the power output and (where appropriate) inter-pulse timing and secondary power levels, remote control of the stimulation power can be disabled for specific states to allow the user to control the power manually. This documentation only describes the controls available from within Signal; for a complete understanding of the effect these have upon the behaviour of the stimulator you should consult the Neuro-MS user manuals.

In addition to using a USB connection to set the stimulation parameters, standard operation with Signal requires that the stimulation be triggered using TTL pulses generated by the 1401 digital outputs. This allows the pulse timing to be precisely controlled relative to the Signal sampling. Manual triggering of the stimulator is also possible, as is triggering from other external hardware, in which case you would normally use the Neuro-MS trigger output to trigger the Signal sampling sweep.

The Neuro-MS support monitors the stimulator health; both waiting until the Neuro-MS is ready for use before allowing a sampling sweep to proceed and terminating sampling if a hardware problem develops. To aid data analysis, Signal saves the Neuro-MS stimulation parameters used in the data section variables of the sampled file. The parameters saved are the the power levels and the inter-stimulus interval. These values are placed in extra frame variables in the sampled data file. The saved values can be retrieved using the script language and used as measurements in trend plot generation.

There is a Getting started with TMS experiments section in the Magstim support documentation; nearly all of this is equally applicable to Neuro-MS devices.

# Neurosoft configuration

To configure a Neuro-MS stimulator in Signal, open the sampling configuration dialog and make sure that Multiple states are enabled in the General section. Select the States section and make sure that you are in *Dynamic outputs* or *Static outputs* mode. If Neurosoft device usage has already been added to the sampling configuration, clicking on the Auxiliary >> button will show a Neurosoft item in the lower section of the menu. If this item is not present in the menu, use the Add device item at the top of the menu to add it. If the Auxiliary >> button is present, but disabled, you are in *External Digital* states mode, change the mode to enable the button. Click on any Neurosoft item in the lower section of the menu to open the configuration dialog for that device.

It is meaningful to use the Neurosoft support without using multiple states, but this is not directly supported because the states system is so closely linked to auxiliary device support. If you want to make use of Neurosoft support without using multiple states (so that the pulse parameters are recorded and Neuro-MS health checks are made, but with complete manual control) you should turn on multiple states, set only one extra state, set state zero to use manual control, and only make use of state zero while sampling. This will allow you access to the Neurosoft configuration dialog to set the basic Neuro-MS parameters.

### Label
At the top of the configuration dialog there is a field where you can enter a device label used to distinguish the device from others.

### Type of stimulator
Below the label there is a selector used to select the Neuro-MS hardware configuration. You should set this to the type of stimulator you will be using (the choices are Neuro-MS Monophasic or Neuro-MS Paired Monophasic) or to Do not use if you do not want to make use of the Neurosoft support in this sampling configuration.

### Don't generate settings text for window title
This check box prevents the Neurosoft support from generating text showing the current stimulus that Signal can display in the title bar of a sampling document. It is often useful to see what the Neurosoft settings are, but some users find it distracting. A separate check box in the sampling preferences must also be set to allow this text to be shown.

### Neuro-MS settings
Following these are three more sections holding the actual Neuro-MS settings; first some controlstoselect the stimulator and select how power is handled, then a display of the settings across all states which is used to select a state for editing (by clicking on the information for that state) and finally an area where the settings for the selected state may be edited. Again, the information shown here will vary according to the type of stimulator selected and is described in the separate specific device configuration details (see below).

### Copy to ...
The button labelled Copy to ... provides a simple dialog that can be used to set up many states quickly. If you press OK it copies the currently selected state to all the other states that are within the selected range. This button is available for all types of Neuro-MS stimulators.

### Test
The button labelled Test checks if communications with the Neuro-MS can be established and that the serial number is correct and displays information about what it has found. The Test button is available for all types of Neuro-MS stimulators.

# Neurosoft Monophasic device configuration

The Neurosoft Neuro-MS device is configured in a simple manner:

### Stimulator selector

When more than one Neuro-MS device is being used it is necessary to specify which device is controlled by which settings. This selector displays the serial numbers of all detected Neuro-MS devices. Normally there will be just one and Signal will detect this and automatically set the selector. If you are using multiple devices you will have to use the selectors to set the device assignment as you wish.

### Powers as % of RMT

When this check box is set, the power values entered for the individual states are percentages of the subject resting motor threshold (RMT) rather than a percentage of the overall device power (when the check box is clear). The RMT value in use, which is always a percentage of the overall device power, is entered to the right of the check box.

The settings available for the individual states are:

### Manual power

Set this check box for the power level for this state to be controlled by the control wheel on the coil and not by Signal, clear it to give Signal control.

### Power (%)

This sets the stimulator power output as a percentage of the maximum available, you can set any value from 0 to 100. This field is disabled for manually controlled states. If device power is being specified as a percentage of the RMT, you can set any value from 0 to 100/RMT.

# Neurosoft Paired Monophasic device configuration

Paired Monophasic Neuro-MS devices can be controlled by Signal to allow generation of paired stimulation pulses with a set interval between the stimulations. When using this device configuration two Neuro-MS devices must be connected to the PC, stimulus triggering can either be done using a delayed trigger generated by the primary device or the two devices can be independently triggered using two digital outputs on the 1401. When this device type is selected extra device configuration controls are displayed:

### Stimulator selector

When more than one Neuro-MS device is being used it is necessary to specify which device is controlled by which settings. This selector displays the serial numbers of all detected Neuro-MS devices. Normally there will be just one and Signal will detect this and automatically set the selector. If you are using multiple devices you will have to use the selectors to set the device assignment as you wish.

### Powers as % of RMT

When this check box is set, the power values entered for the individual states are percentages of the subject resting motor threshold (RMT) rather than a percentage of the overall device power (when the check box is clear). The RMT value in use, which is always a percentage of the overall device power, is entered to the right of the check box.

The settings available for the individual states are:

### Manual power

Set this check box for the power levels for this state to be controlled by the manual stimulator controls and not by Signal, clear it to give Signal control.

### Power (%)

This sets the primary stimulator power output as a percentage of the maximum available, you can set any value from 0 to 100. If device power is being specified as a percentage of the RMT, you can set any value from 0 to 100/RMT. This field is disabled for manually controlled states.

### Power 2 (%)

This sets the secondary device power output as a percentage of the maximum available, you can set any value from 0 to 100. If device power is being specified as a percentage of the RMT, you can set any value from 0 to 100/RMT. This field is disabled for manually controlled states.

### Interval (ms)

This sets the delay after the primary stimulator fires before it generates the trigger pulse to trigger the secondary stimulator from 0 to 1000 milliseconds (with a time resolution of resolution of 0.1 ms up to 100 ms then 1 ms up to 1000 ms).

# Neurosoft connections and cabling

Each Neuro-MS stimulator is controlled using a standard USB connection. It is necessary to install the necessary drivers for the Neuro-MS system, these will be installed as part of the standard Neurosoft software installation process, which should be carried out (and correct operation verified, if possible) before connecting Signal to a Neuro-MS device or devices.

### Connections for triggering

Each Neuro-MS stimulator has a multi-pin Trig in/out connector, which provide separate trigger input and trigger output connections. Both of these connections use non-standard signals which are normally low, however they are TTL compatible and suitable for direct connection to the 1401 digital outputs and inputs.

*Single Monophasic stimulator*

With a single stimulator in use the most common arrangement would be to have the stimulator triggered by the 1401, with a single BNC cable connecting a 1401 digital output to the TRIG-IN pin on the stimulator. A high-going TTL pulse (1 to 10 milliseconds would be a suitable length) should be generated on the appropriate digital output at the time that you wish the stimulator to fire, this pulse should be present in the outputs for all states.

A possible alternative arrangement would be to trigger the stimulator manually and use this to trigger the sampling sweep, using a single BNC cable connecting the stimulator TRIG-OUT pin to the trigger input on the 1401. As the stimulator generates a high-going TTL-compatible pulse when it fires, the Rising edge trigger check box in the general page of the sampling configuration should be set. Peri-triggered sampling mode allows the capture of pre-trigger data and is thus most suitable for this style of use.

*Paired Monophasic stimulators*

With this arrangement the primary stimulator is triggered by the 1401 but the secondary stimulator is triggered by the primary stimulator after a preset delay. A single cable connects a 1401 digital output to the TRIG-IN socket on the primary stimulator. A high-going TTL pulse should be generated on the appropriate digital output at the time that you wish the primary stimulator to fire, the primary stimulator then triggers the secondary after the preset delay.

*Dual independent stimulators*

This arrangement is very similar to a single stimulator triggered by the 1401, but you now have two Neuro-MS devices independently connected to the PC by two USB cables and controlled as separate auxiliary states devices. Two cables are used to connect two 1401 digital outputs to the TRIG-IN pin on each stimulator. A high-going TTL pulse (10 milliseconds would be a suitable length) should be generated on each digital output at the time that you wish the stimulators to fire, these pulses would normally be present in the outputs for all states but with varying timing to give you varying inter-stimulation intervals. There is no reason why a trigger pulse cannot be omitted on some states to give you a single stimulation.

# Notes on Neurosoft use

A straightforward arrangement to use a Neuro-MS stimulator with Signal would be as follows:

1. Connect the Neuro-MS to your computer using a standard USB cable and install the Neurosoft software and device drivers. Using a suitable cable, connect the TRIG-IN pin on the stimulator to the 1401 digital output port 0 BNC socket found on the front of all modern types of 1401.

2. In the outputs page of the sampling configuration, make sure that digital output bit zero is enabled for use. Using the pulses configuration dialog set the initial level of digital output bit zero to 1 and place a pulse (which will be low-going) in the outputs at the time when you want the Neuro-MS to fire. This output pulse should be at least 50 microseconds long – 1 millisecond works well.

3. The sweep mode in the General section of the sampling configuration should be set to Fixed interval mode so that Signal controls the interval between sweeps and that this interval can be set long enough that the stimulator has enough time to recharge. The pulses configuration dialog is also used to set the sweep interval and optional random variation in the interval - the relevant values are found as part of the control track information.

4. The Neurosoft support uses Signal multiple states sampling, which should be set up in dynamic outputs mode. You can use any number of extra states, each extra state providing separate Neuro-MS settings. Each set of pulse outputs should include digital output pulses to trigger the Neuro-MS stimulators along with any other outputs required. The Neurosoft support will work correctly with manual control of the states or with any style of automatic states sequencing including protocols.

When Signal begins sampling with the Neurosoft support enabled, it checks for a correctly functioning Neuro-MS device as part of the process of initialising for sampling. If a Neuro-MS is found and enabled then Signal will carry out the initial configuration of the Neuro-MS. While sampling is in progress, Signal will set up the Neuro-MS using the current state data before each sweep, it will then delay the start of each sweep until the Neuro-MS reports that it is armed and ready. At the end of each sweep the Neuro-MS health is checked to make sure it is OK. Note that the checks on Neuro-MS readiness can impose a significant extra inter-sweep delay.

If the Neuro-MS coil temperature rises too high, the unit will disable itself and Signal will stop sampling. Once the coil temperature has dropped sufficiently you can press 'More' on the sampling control panel to resume sampling again.

If manual control is selected, the Neuro-MS will fire on the trigger with whatever power settings it has been given using the coil control wheel or with whatever settings it had prior to manual mode being used.

# DS8R current stimulator

The DS8R biphasic current stimulator from Digitimer can be used to generate Signal-controlled monophasic or biphasic pulses of 50 to 2000 microseconds duration with an output range of 2mA to 1000mA in 0.1 mA steps, however the actual current achieved may be restricted by a pulse energy limit of 300mJ per pulse and the skin/electrode resistance. The DS8R is fully controlled by Signal, allowing you to set a different mode and level of current for each state. Signal controls the DS8R stimulator using a standard USB connection to interact with the hardware.

# DS8R safety notice

The DS8R current stimulator can generate outputs of up to 400 volts, however the actual current achieved will be restricted by a pulse energy limit of 300mJ per pulse which avoids otherwise dangerous stimulations. The DS8R is suitable for human research applications, but because the maximum pulse energy of 300mJ exceeds the 50mJ limit imposed by the international standard for diagnostic stimulation devices, it does not have medical device certification and should not be used in a clinical situation. Before using the device you must should read the *DS8R Operators manual* paying particular attention to the *Precautions and Warnings* section. It is your responsibility to ensure that Signal's control of a DS8R is set up in an appropriate and safe fashion for the intended use.

# DS8R introduction

This documentation only describes the DS8R controls available from within Signal; for a complete understanding of the effect these have upon the behaviour of the stimulator you should consult the *DS8R Operators manual*.

The timing of the DS8R current pulses is controlled by a TTL signal connected to the front panel Trigger input. Signal can control the timing of the current pulses with one of the 1401 digital outputs in *Dynamic outputs* mode, or external equipment can control the pulsing in *Static outputs* mode. For safety reasons, the DS8R will limit the energy delivered to the subject to 300 mJ per pulse.

Signal monitors the stimulator health; waiting until it is ready for use before allowing a sampling sweep to proceed and terminating sampling if a hardware problem develops. To aid data analysis, Signal saves the stimulation current, duration, dwell time, recovery percentage, mono-phasic flag and stimulation polarity in the data section variables of the sampled file, note that these values are saved even if the DS8R is manually controlled. These values are placed in extra frame variables in the sampled data file and can be retrieved using the script language or used as measurements in trend plot generation.

# DS8R configuration

To configure the DS8R in Signal, open the sampling configuration dialog and make sure that Multiple states are enabled in the General section. Open the States section and make sure that you are in *Dynamic outputs* or *Static outputs* mode. If DS8R usage has been added to the sampling configuration, clicking on the Auxiliary >> button will show a DS8R item in the lower section of the menu. If this item is not present in the menu, use the Add device item at the top of the menu to add it. If the Auxiliary >> button is present, but disabled, you are in *External Digital* states mode, change the mode to enable the button. Click on any DS8R item in the lower section of the menu to open the DS8R configuration dialog for that device.

From the configuration dialog you can enable use of the DS8R, set overall device parameters, define the stimulator settings for each state in use and test for successful communications with the DS8R.

At the top of the dialog are settings that apply to all states. In the centre is a list of the currents set for each state. Below this is a field to edit the stimulation settings for the selected state. At the bottom are buttons to copy settings, get help, test the DS8R connection and accept the dialog settings.

### Label
At the top of the configuration dialog there is a field where you can enter a device label used to distinguish the device from others.

### Hardware selection
Set this to Use DS8R to use the stimulator or to Do not use if you do not want to make use of the DS8R support software in this sampling configuration.

### Don't generate settings text for window title
This check box prevents the DS8R system from generating text showing the current level that Signal can display in the title bar of a sampling document. It is often useful to see what the DS8R settings are, but some users find it distracting. A separate check box in the sampling preferences must also be set to allow this text to be shown.

### Settings for state...
The DS8R settings available for the individual states are:

#### Disable
Set this checkbox to disable the stimulator in this state

#### Manual
Set this checkbox to control the stimulator using the manual front-panel controls in this state

#### External command source

Set this checkbox if you want the stimulator current to be controlled by the voltage on the Analogue Voltage Amplitude Control Input on the rear of the DS8R box.

#### Stimulus Output (mA)

This control sets the stimulation current to be used, in milliamps. You can enter any value between 2 and 1000 mA, in 0.1 mA steps.

#### Stimulus Mode

This sets the stimulator mode, you can select Mono-phasic or Bi-phasic mode.

#### Polarity Mode

This sets the stimulation polarity, you can select Positive, Negative or Alternating polarity.

#### Pulse Duration (us)

This control sets the duration of the stimulation pulse, in microseconds. You can enter any value between 50 and 2000 microseconds, in 10 microsecond steps.

#### Recovery Phase Ratio (%)

This control sets the amplitude (stimulation current) of the recovery phase (for Bi-phasic mode only) as a percentage of the stimulus phase current. At 100%, the recovery phase will have the same amplitude and duration as the stimulation phase, with values less than 100% the recovery current is reduced and the duration of the recovery phase increased in proportion to preserve charge balancing.

#### Interphase Interval (us)

This control sets the interval between the stimulation phase and the recovery phase (for Bi-phasic mode only) in microseconds. You can enter any value between 1 and 990 microseconds, in 10 microsecond steps.

#### Beep if out of compliance

This control enables an audible warning from the stimulator if the skin+electrode resistance is too high for the stimulator to deliver the requested current.

### Copy to ...

The button labelled Copy to ... provides a simple dialog that can be used to set up many states quickly. If you press OK it copies the currently selected state to all the other states that are within the selected range.
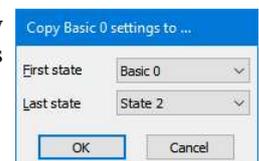
### Test

This button tests if communications with the DS8R can be established. It checks that you have the correct COM port, that the serial line is connected and that the DS8R is operating correctly.

The Help, Cancel and OK buttons have their usual meanings.

### Simple use

If you want to use the DS8R from Signal very simply with a single stimulation setting, you must still enable multiple states. Set one extra state (so states are enabled), and set the desired stimulation for state zero, and then sample using state zero only. This gives you access to the configuration dialog to set the basic DS8R parameters and enables checking of DS8R behaviour during sampling and the recording of the stimulation settings.

# DS8R connections and cabling

The DS8R uses a standard USB A-B cable for control, this cable connects the DS8R to the computer that is running the Signal software. See the *DS8R Operators manual* for more information about the control connection.

The BNC Trigger input on the rear of the DS8R should be connected to the 1401 digital outputs using a standard BNC cable.

# Notes on DS8R use

A straightforward arrangement to use a DS8R with Signal, assuming we use digital output bit 0 to control the current pulse timing, would be as follows:

1. Connect your computer to the DS8R using a USB connection. Using a standard BNC cable, connect the DS8R rear Trigger BNC socket the 1401 digital output 0 BNC socket found on the front of all modern types of 1401.

2. In the Outputs page of the sampling configuration, make sure that digital output bit zero is enabled for use. Using the pulses configuration dialog, set the initial level of digital output bit zero to 0 and place a pulse (which will be high-going) in the outputs at the time when you want the DS8R to fire. This output pulse should be at least 1 millisecond long.

3. The DS8R support is designed to work with Signal multiple states sampling, which should be set up in *Dynamic outputs* mode. Set one extra state for each stimulation setting you require and define the stimulations that you require. Each state should generate a digital output pulse to trigger the DS8R plus any other outputs required. The DS8R support will work with manual control of the states or with any style of automatic states sequencing including protocols.

When Signal begins sampling with the DS8R support enabled it checks for a correctly functioning device. If a DS8R is found, Signal initialize the device and enables the stimulator as necessary. While sampling is in progress, Signal sets the output current and other parameters using the predefined state data before each sweep. At the start and end of each sweep the DS8R health is checked to make sure it is OK. When sampling finishes, the DS8R is disabled.

Because of the extra overhead in setting up current levels for each sweep and checking that the device is operating correctly, you may find that the maximum sweep rate with the DS8R is slightly reduced.

# CED 3304 current stimulator

The CED 3304 current stimulator can be used to generate controlled constant-current stimulations of up to 10 milliamps and is fully controlled by Signal, allowing you to set a different level of current for each state. Signal controls the CED 3304 programmable constant current stimulator using a serial line or USB connection to interact with the hardware.

# CED 3304 safety notice

The CED 3304 current stimulator can generate outputs of up to 90 volts, which lies within the potentially lethal range. Before using the device you must read the *CED 3304 Owners Handbook* paying particular attention to the warnings and precautions section. In particular, you are reminded that the CED 3304 is not qualified for human use. It is your responsibility to ensure that Signal's control of a CED 3304 is set up in an appropriate and safe fashion for the intended use.

# CED 3304 introduction

This documentation only describes the CED 3304 controls available from within Signal; for a complete understanding of the effect these have upon the behaviour of the stimulator you should consult the *CED 3304 Owners Handbook*.

The CED 3304 produces current pulses in one of four current ranges: 10 and 100 microamps and 1 and 10 milliamps. The current range is selected by a manual switch on the CED 3304. You specify a current range when you define the stimulus settings in Signal; Signal will not let you sample if the wrong range has been set on the CED 3304.

The timing of the 3304 current pulses is controlled by a TTL signal connected to the front panel Trigger input. Signal can control the timing of the current pulses with one of the 1401 digital outputs in *Dynamic outputs* mode, or external equipment can control the pulsing in *Static outputs* mode. For safety reasons, the CED 3304 will switch off a pulse after a preset limit is reached – see the 3304 documentation for details of how to control this limit.

Signal monitors the stimulator health; waiting until it is ready for use before allowing a sampling sweep to proceed and terminating sampling if a hardware problem develops. To aid data analysis, Signal saves the stimulation

current in the data section variables of the sampled file. This value, in uA, is placed in an extra frame variable in the sampled data file. The saved values can be retrieved using the script language and used as measurements in trend plot generation.

# CED 3304 configuration

To configure the CED 3304 in Signal, open the sampling configuration dialog and make sure that Multiple states are enabled in the General section. Open the States section and make sure that you are in *Dynamic outputs* or *Static outputs* mode. If CED 3304 usage has been added to the sampling configuration, clicking on the Auxiliary >> button will show a CED 3304 item in the lower section of the menu. If this item is not present in the menu, use the Add device item at the top of the menu to add it. If the Auxiliary >> button is present, but disabled, you are in *External Digital* states mode, change the mode to enable the button. Click on any CED 3304 item in the lower section of the menu to open the CED 3304 configuration dialog for that device.

From the configuration dialog you can enable use of the CED 3304, set the serial line port used to communicate with the hardware and other overall parameters, define the current settings for each state in use, set the active level for the Trigger input and test for successful communications with the CED 3304.

At the top of the dialog are settings that apply to all states. In the centre is a list of the currents set for each state. Below this is a field to edit the current for the selected state. At the bottom are buttons to copy settings, get help, test the CED 3304 and accept the dialog settings.

### Label
At the top of the configuration dialog there is a field where you can enter a device label used to distinguish the device from others.

### Hardware selection
Set this to Use CED 3304 to use the stimulator or to Do not use if you do not want to make use of the 3304 support software in this sampling configuration.

### Com port for communications
Signal connects to the CED 3304 either via a serial communication port, or through a USB port, which is setup as a virtual COM port. This field sets the COM port that the CED 3304 is connected to, you can select any of the PC COM ports that are detected as available. If you use the USB connection, follow the instructions at the start of the Operation chapter of the *CED 3304 Owners Handbook* to obtain a suitable device driver for your operating system.

### Current range selection
This field sets the maximum current that you can enter for each state. You can select from 10 and 100 microamps and 1 and 10 milliamps. This control matches the current range rotary switch on the front of the CED 3304 but does not override its setting – the actual range switch setting must match the setting in this dialog or an error will be generated when sampling is begun.

### Active high trigger
Check the box for an active high trigger (current is generated while the CED 3304 Trigger input is at a high TTL level). Clear the box for active low operation (current generated when the Trigger input is at a low TTL level). When you are driving the trigger input from the 1401 digital outputs, it is usual to set an active high trigger.

### Don't generate settings text for window title

This check box prevents the 3304 system from generating text showing the current level that Signal can display in the title bar of a sampling document. It is often useful to see what the 3304 settings are, but some users find it distracting. A separate check box in the sampling preferences must also be set to allow this text to be shown.

### Settings for state...

This field displays and lets you edit the desired current for the state selected in the list of states. You can enter any value (in microamps) from zero to the maximum available, for example in the 10 uA range you could enter: 0 or 8.234 or 10.

### Copy to ...

The button labelled Copy to ... provides a simple dialog that can be used to set up many states quickly. If you press OK it copies the currently selected state to all the other states that are within the selected range.
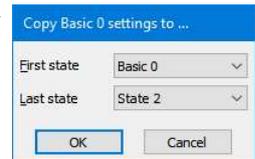
### Test

This button tests if communications with the CED 3304 can be established. It checks that you have the correct COM port, that the serial line is connected and that the CED 3304 is operating correctly.

The Help, Cancel and OK buttons have their usual meanings.

### Simple use

If you want to use the CED 3304 from Signal very simply with a single current setting, you must still enable multiple states. Set one extra state (so states are enabled), and set the desired current for state zero, and then sample using state zero only. This gives you access to the configuration dialog to set the basic CED 3304 parameters and enables checking of CED 3304 behaviour during sampling and the recording of the stimulation settings.

# CED 3304 connections and cabling

The CED 3304 uses a USB A-B cable or a 9-pin serial-line cable (both are supplied with the unit) for control. Use one cable or the other. If you plug in both, the USB connection will take precedence. See the Operation chapter of the *CED 3304 Owners Handbook* for more information about the control cables.

The front panel Trigger input should be connected to the 1401 digital outputs or other TTL pulse source using a standard BNC cable. The front panel Data and Clock inputs are not used with Signal and should be left unconnected.

### 3304 serial cable
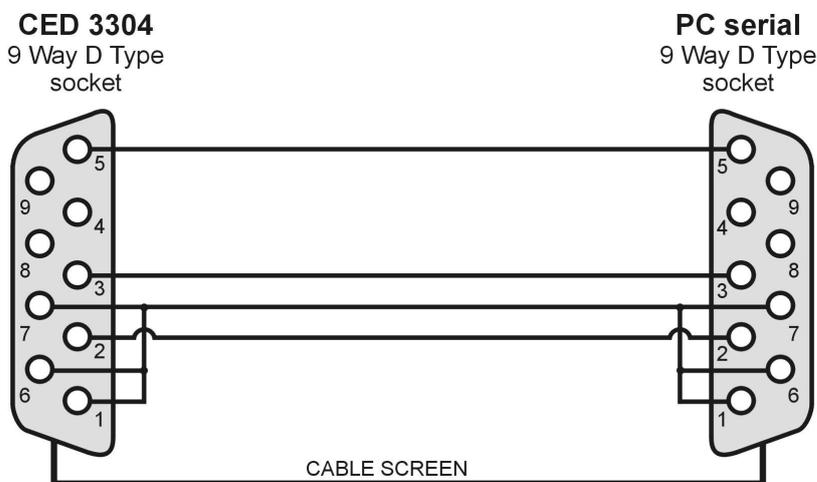
# Notes on 3304 use

A straightforward arrangement to use a CED 3304 with Signal, assuming we use digital output bit 0 to control the current pulse timing, would be as follows:

1. Connect your computer to the CED 3304 using the either a serial or use a USB connection. Using a standard BNC cable, connect the CED 3304 front panel Trigger BNC socket to the 1401 digital output 0 BNC socket found on the front of all modern types of 1401.

2. In the Outputs page of the sampling configuration, make sure that digital output bit zero is enabled for use. Using the pulses configuration dialog, set the initial level of digital output bit zero to 0 and place a pulse (which will be high-going) in the outputs at the time when you want the CED 3304 to fire. This output pulse should be as long as the required stimulation.

3. The CED 3304 support is designed to work with Signal multiple states sampling, which should be set up in *Dynamic outputs* mode. Set one extra state for each output current setting you require and each state should generate digital output pulses to trigger the CED 3304 plus any other outputs required. The CED 3304 support will work with manual control of the states or with any style of automatic states sequencing including protocols.

When Signal begins sampling with the CED 3304 support enabled it checks for a correctly functioning device. If a CED 3304 is found, Signal will check the range switch setting, initialize the device and enable the trigger. While sampling is in progress, Signal sets the output current using the state data before each sweep. At the start and end of each sweep the CED 3304 health is checked to make sure it is OK. When sampling finishes, the CED 3304 trigger is disabled.

Because of the extra overhead in setting up current levels for each sweep and checking that the device is operating correctly, you may find that the maximum sweep rate with the CED 3304 is reduced.

# Technical support

## Contacting CED

### Technical support

We have software and hardware technical support desks that are available during UK working hours (and somewhat beyond). You can call us by telephone but our preferred medium is email as you can send example files and scripts as enclosures (even from within Signal with the File menu Send Mail option).

USA and Canada: 1 800 345 7794
World-wide: +44 1223 420186
email: softhelp@ced.co.uk
email: hardhelp@ced.co.uk

You can also find up to date information, example scripts, and Signal down-loadable updates on our **Web site** at https://ced.co.uk

We also have an on-line forum where both CED staff and other CED users are able to share suggestions and ask questions.

If you send us an attachment with your email, please have mercy … don't send us gigabytes. We have a medium speed link and decent email server so a few megabytes is fine, but for more than that we have a mechanism available on our web site that is a much easier way to send us large data files - this can cope with files that are many gigabytes in size, if necessary! Most problems that need a file can be illustrated with a file section (use the File menu Export option to chop out a small piece) - this is still preferable as transferring gigabytes of data over the Internet does take rather a long time.

Before you contact us please make sure you have checked that the information you need is not in the manuals or the on-line help. The on-line help is more up-to-date than the manuals and the Index is always worth a try; remember to try a few synonyms when searching for information.

Please tell us the serial number of your copy of Signal (this can be found in About Signal in the Help menu), the Signal version number, the operating system version, and any hardware-related information that might be relevant to the problem. If you have a crashing problem, please read the information about Dr Watson (below).

### Using Dr Watson to report crash information

If Signal crashed out with a system error (General Protection fault, or the like), please send us all the information that the system gives you as often we can pinpoint the offending line in the code given the register dump. The best way to get the information is by using the `DrWtsn32` program. If this program is enabled, it writes a log file called `DrWtsn32.log` that describes what happened during a crash. This file is written to the folder:

`%systemdrive%\Documents and Settings\All Users\Application Data\Microsoft\Dr Watson`

unless you have explicitly changed this by running the `DrWtsn32.exe` program and modified the save path. `%systemdrive%` is the drive where your operating system is installed, usually `C:`. If you are not getting any log files written, open a command prompt and use the command:

`DrWtsn32 -i`

to enable it. You can also run the program by typing `DrWtsn32` at the command prompt to configure other options. The log file is a text file that you can open and read. Normally, each crash is appended to the end of the file, so if your log file is huge, you may wish to only send us the most recent information from the end. Sending us this log file plus a description of what you were doing often makes finding and fixing problems much quicker. If the problem is easy to reproduce, please send us all the steps required to reproduce it. If it only happens very occasionally with no obvious cause, the Dr Watson log file is essential. We give a very high priority to finding and fixing crashing problems; we will generate a fix for the next release and may even be able to send you a fix or a work-around.

### Sales related enquiries

Write, telephone or email us at:

Cambridge Electronic Design Ltd
Technical Centre,
139 Cambridge Road,

Milton,
Cambridge   CB24 6AZ
UK

Telephone: +44 1223 420186
email: sales@ced.co.uk

If you want price list or product information, you could try our World Wide Web page:
https://ced.co.uk

# Revision history

The following changes, additions and fixes have been made to Signal within version 7.00. You can also get update information, downloads and script examples from our web site. The revision history lists items under the following headings:

**New**        Extra features in the program or improvements in operation.

**Fixes**      Repairs of broken features (bugs) and other corrections.

**Changes**    An alteration to Signal (which may also be listed in New or Fixes) that may require you to change your use of Signal, existing scripts or program operations. We try to minimise these.

The **Known issues** section, if present, lists known problems for which there is no fix currently available.

# Revision 7.07

### April 2022

Version 7.07 of Signal is a maintenance update providing a number of fixes to the software. The main Signal development work has now moved on to version 8, future releases of version 7 are expected to only contain bug fixes.

## Features

1. There is now a check box in the DuoMag settings to disable checks on triggering.

2. You can now specify the output pulses for dual DuoMag MP devices  in either order.

## Fixes

1. The level dialog for virtual channel expressions now accepts negative values.

2. Standard display now turns on channel numbers.

3. You can now undo showing channel numbers.

4. Modifying a virtual channel expression from within the virtual channel dialog now causes the channel to be redrawn in other views.

5. Modifying the virtual channel expression from a script will now cause dependent virtual channels to update.

6. Previously, it was possible to interactively carry out a process such as waveform averaging or measurement generation during sampling when this processing was already executing as part of the sampling. This is now prevented.

7. The PulseAdd() script function now works for digital channels.

8. Marker times are kept quantised, and duplicate markers at the same (quantised) time are now prevented.

9. The ViewList() script function now lists sequencers views correctly.

10. The copy pulses to states operation in the Pulses dialog is now detected as a change and so enables the OK button.

11. Previously, importing ADI data files with high section counts could cause the importer to fail.

12. The YRange() script function now works in grid views as documented.

13. Signal no longer fails if the sampling configuration dialog is closed while browsing for a folder in the Automation tab.

14. Previously Signal could lose keyboard markers if the PC had been left on for many days, this is now fixed.

15. Previously when sampling without multiple states, the auxiliary states control bar might be visible and the auxiliary states items in the sample menu could be available.

16. Previously, changing a TMS stimulator RMT value using the auxiliary states control bar would not affect the actual device settings. This has now been fixed.

17. Previously the information on available 1401 firmware versions was incorrect so that Signal might suggest that an update was available when it was not.

# Revision 7.06

### April 2021

Version 7.06 of Signal is a maintenance update with few new features and a large number of fixes to the software. The main Signal development work has now moved on to version 8, future releases of version 7 are expected to only contain bug fixes.

## New

1. The `FilePath$()` script function allows a parameter of -6 for the path to application data.

2. The Copy button in the About dialog now generates text showing path information.

3. The `BinZero()` script function can now be used to set the start time for a frame in a file view.

4. The `ToolbarEnable()` script function can count enabled buttons, report if Idle set.

5. The DS8R auxiliary states device now stores stimulus parameter information in data section variables when being manually controlled.

6. A new `SampleClampHP()` script function has been added to allow control of the holding potential during sampling.

7. The `ChanKey()` script function has been extended to allow a key to be moved outside the visible area of the view.

## Fixes

1. Line drawing mode now draws correctly with a square root axis.

2. Signal attempts to create any missing key data paths on startup, a message will be generated in the log window if this does not succeed.

3. Previously it was possible to inadvertently disable writing to disk by using `SamplePause()` when sampling was already paused.

4. The error message for the next step in a states protocol being wrong now quotes the correct range.

5. Sampling very large numbers of 1401 ports could cause errors due to an excessively long string being generated. This is now fixed for 0 to n port lists.

6. Large port counts in a sampling configuration could cause setting-up for sampling to take an excessively long time.

7. Previously, `FrameTag(0)` could crash Signal.

8. The debug output generated for command 8 of the MagPro stimulator has been corrected.

9. Importing some data files with channel comments could cause a complete crash.

10. Previously, the script compiler could get very confused by illegal characters.

11. When trying to read clipboard text, Signal will retry several times after any failure in case the clipboard is unavailable.

12. The ViewColourSet\Get script functions now work on an info view.

13. Sliders in user dialogs that are set to use integer values behave better, particularly when using the arrow keys to move the slider.

14. Slider tooltip displays while dragging avoid duplicate tooltips.

15. Signal is now protected against a possible crash when opening a corrupted resource file.

16. Saving a grid view clears the modified (*) marker in the view title.

17. Digital outputs generated using the pulses system are correctly positioned before marker generation so that they are synchronous.

18. Info and grid windows came up as "Unknown" in global debug window.

19. Grid documents were previously always saved, even when the user said not to save them.

20. The timing calculations for output pulses have been improved to avoid possible erroneous rounding effects.

21. The Cursor() and CursorLabelPos() script functions now record correctly.

22. Previously, creating user dialogs or using the Message() script function with the main window hidden could jam-up Signal.

23. The PowerMAG support now ensures that powers are multiples of 0.5% MSO.

24. Signal conditioner controls are once again available online if sampling was started directly from the sampling configuration dialog.

25. If, while sampling, you deleted virtual channels and then duplicated the view, previously the deleted virtual channels would re-appear.

26. Data view Y axes now behave sensibly if big ticks are turned off using YAxisMode(), previously this caused channel titles to be drawn horizontally.

27. Sampling with the PowerMAG stimulator in ppTMS mode could miss the first trigger after switching to a new sampling state.

28. Previously when sampling using the PowerMAG or Neurosoft stimulators the power levels stored in frame variables in the new data file was incorrect.

29. Previously the pulses dialog would show a spurious timing fault at the end of an arbitrary waveform output using more than one DAC.

30. The TRIG output sequencer instruction has been adjusted to prevent possible problems with external trigger pulses.

31. The font dialogs now initialise the Style field correctly.

32. Previously, the POINTS and XOR sequencer instructions for the micro1401-4 did not work correctly.

33. The behaviour of the File information dialog while sampling has been improved, you can now change the column widths without them being immediately reset.

## Changes

1. Waveforms are no longer rectified when drawn in dots mode with a square root axis.

2. The TDT importer will not fail outright if some .sev files are missing, but will import any file that are present and record the missing ones.

3. Data view X axes now still show the time units if big ticks have been turned off using XAxisMode().

4. The units for TMS stimulator power levels saved in frame variables are now either "% MSO" or "% RMT" according to the device configuration.

# Revision 7.05

## December 2019

Version 7.05 of Signal extends the Auxiliary states device support by adding support for NeuroSoft Neuro-MS TMS stimulators, and provides improved Y axis optimisation by allowing it to be carried out automatically after

each sweep is sampled along with generally improved axis optimisation for XY views. We have also made a number of minor improvements and fixes.

## New

1.  Support for the Neuro-MS transcranial magnetic stimulators from NeuroSoft has been added.

2.  Automatic Y axis optimisation of data channels at the end of a sampling sweep is now supported via a check box in the sampling configuration.

3.  A new SampleAutoOpt() script function has been added to give scripted access to this feature.

4.  The XY view axis optimisation mechanisms has been much extended with a new Autoscale Axes system.

5.  The new Micro1401-4 hardware from CED is now fully supported, this type of 1401 supports dynamic clamping unlike previous types of Micro1401.

6.  When sampling with a Power3A or Micro4 system, the 200 MHz system clock available in these systems will be used to obtain higher timing precision, for example of ADC sampling rates.

7.  A new HCursorVisible() script function has been added to control horizontal cursor visibility.

8.  The System() function has been extended to return GUI thread timing info.

9.  The System$() function has been extended to provide Signal command line access.

10. A new FileTimeDateSet() script function has been added, this can be used to set the sampling time start for a data file.

11. A new ChanKey() script function has been added, this can be used to control the position and appearance of keys in XY views.

12. A new XYSetData() script function has been added, this can be used to edit the data in an XY view.

13. Real marker channels can now use secondary marker values to decorate the displayed data with error or confidence range bars.

14. A new ChanDecorate() script function has been added to provide scripted access to this feature.

15. The dialogs used to edit script variables have all been enhanced.

## Fixes

1.  EditCopy() leaked a GDI handle when used to place a bitmap on clipboard.

2.  CursorVisible() with -1 cursor number to select all cursors now works correctly.

3.  Large channel background bitmaps are now handled much more efficiently.

4.  SampleSeqWave() did not work correctly with only one arg.

5.  Previously, MagStim auxiliary stimulator settings failed to load correctly from an old-style sgc file.

6.  The dynamic clamp setup dialog now correctly shows the 'zero DACs between frames' state during sampling.

7.  The Selection$() script function now works correctly on grid views.

8.  In the pulses dialog, the step value for a square pulse with varying length was being corrupted if the pulse times were not displayed in seconds.

9.  The data buffer size for imported files is now limited to a safe value, this fixes some cases where huge buffers were being used, causing memory problems.

10. The MagStim auxiliary device support was failing to correctly convert power values specified as a percentage of RMT to the actual power levels wanted.

## Changes

1.  Signal has been converted to use new-style XML file access mechanisms by using a newer much improved XML support library.

2.  The XML information for HH(A/B) dynamic clamping models was illegal according to the full XML standard and this data was refused by our new, faster, better library. Old XML cannot be read without editing the sampling configuration file manually, a version of Signal version 6 which automatically fixes this XML will be released soon.

3.  The MagStim support software now automatically attempts to restore remote control if it is lost, a warning message will be generated in the log window if this occurs.

4.  The mechanisms used to ensure that the colours drawn are clearly visible have been tidied up so that they are a bit less fussy.

5.  The MagPro TMS device support now works with some older versions of MagPro firmware though we would recommend anyone using a MagVenture system to update their firmware to the latest version

# Revision 7.04

## September 2019

Version 7.04 of Signal was released in order to get rid quickly of the installer for the faulty version 7.03, which was released in error. It contains no new features.

## Fixes

1.  Sampling configurations are now read back correctly from disk, version 7.03 failed to read the information relating to windows and views, online processing and virtual channels.

2.  The installer now installs the correct importer for Spike2 data files.

# Revision 7.03

## August 2019

Version 7.03 was released in error at around this time. This software was never intended for release and contains known faults. We apologise to our users for any inconvenience this has caused.

# Revision 7.02

## March 2019

Version 7.02 of Signal extends the functionality of the auxiliary states support by allowing the device settings dialogs to be used online, allowing TMS auxiliary device power to be specified as a percentage of the resting motor threshold (RMT) value and adding an auxiliary states control bar for quick access to the devices. We have also made a number of minor improvements and fixes.

## New

1.  Measurement generation using Expression mode will accept Y value expressions such as "H1" (and measurement expressions such as `Mean(t1, t2{, c})` if the optional channel number is provided), previously only expressions relating to an x axis position could be used.

2.  The auxiliary states device settings dialogs can now all be used during sampling, in general only the per-state settings may be changed while online.

3.  A new dialog has been added to simplify copying auxiliary state device settings, this this is accessed using a Copy to ... button which replaces the Copy all and Copy above buttons previously present in all auxiliary states settings dialogs.

4.  Auxiliary states device support for TMS devices now allows the power levels to be optionally specified as a percentage of resting motor threshold (RMT), the `SampleAuxStateParX()` script function has been extended to match.

5.  A new auxiliary states control bar has been added to simplify use of these devices during sampling, this provides a single quick access variable (resting motor threshold (RMT) for TMS devices) along with a button to open the main settings dialog.

6. The title and units of virtual or memory channels can now be changed during sampling.

7. A new `Chan()` script function has been added, this turns strings into channel numbers.

8. You can now type time interval expressions such as *1/200* as a sample interval for virtual channels.

9. You can now allow external text files to be opened as Unicode using the `FileOpen()` script language function by adding 32 to the `mode%` parameter.

10. The support for the MagStim Rapid TMS stimulator as an auxiliary states device now includes a recharge delay parameter to reduce possible noise.

11. A new colour scale dialog is available from the Display preferences, this can be used to define colour scales to be used for colour mapping.

12. The `EditPaste()` script command has been extended to support moving a bitmap into an integer array.

13. The `EditCopy()` script command has been extended to copy real and integer arrays to the clipboard as bitmaps.

14. A new `EditImageSave()` script command has been added, it saves bitmap images held on the clipboard to a file.

15. The `Spline2D()` script command has been extended to use the colour scales to define mapping colours and to generate a rectangular grid of result values in addition to the current bitmap-format output.

## Fixes

1. The Measurements to trend plot settings dialog now enables and disables the New button correctly, especially when using an Expression measurement.

2. If the DS8R auxiliary states device support was used with no DS8R device connected (and with use without hardware enabled) previously this would crash Signal.

3. The day of week conversion carried out by the `TimeDate()` script function used to fail for Saturday and Sunday, it now works correctly.

4. The `MeasureToChan()` script function now accepts points mode for cursor 0 iteration.

5. The `MeasureY()` script function is now recorded with the correct measurement type code.

6. The Channel to match option now records correctly for virtual channels.

7. "File information..." no longer appears twice in the right-click popup menu for memory views.

8. Changing the state label in the Pulses dialog is now correctly detected as a change to the pulses data.

9. Signal no longer crashes if you attempt to apply a FIR filter to a frame which does not contain any data.

10. Spinners now work correctly for the pulse delay item in the Pulses dialog when not using absolute times.

11. A single DAC and 8 digital outputs will now render correctly in the Pulses dialog.

12. Script functions with more than 20 arguments are now handled correctly.

13. A leak of Windows resources when setting up for sampling has been corrected.

## Changes

1. The option to allow sampling with a missing auxiliary states device is now in the sampling preferences instead of the settings dialog for the individual auxiliary states devices.

2. The number of auxiliary states devices is now limited to 10, previously the relevant settings dialog was not accessible for any devices beyond this limit.

3. When exporting an XY view to a MATLAB data file, a MATLAB variable is now generated for any XY channels with no visible data.

# Revision 7.01

## August 2018

Version 7.01 of Signal extends the functionality of the auxiliary states support by allowing multiple auxiliary states devices to be independently controlled. We have also added a new control to those possible with script-generated dialogs, added a few missing script functions that control the sampling configuration, and extended various file-related script language functions.

## New

1. Signal can now make use of multiple auxiliary states devices, any mix of different (or the same) types of device can be used. A label can be assigned to the individual devices to allow them to be distinguished.

2. New `SampleAuxStateNew()` and `SampleAuxStateRemove()` script functions have been added to allow manipulation of the list of auxiliary states devices in use.

3. New `SampleAuxStateParX()`, `SampleAuxStatePar$()` and `SampleAuxStateValX()` script functions have been added to give access to the settings of multiple auxiliary states devices.

4. The error messages generated by the auxiliary states system are much more informative, particularly as regards failure to set up the device at the start of sampling.

5. A new progress bar control has been added to the controls available in script-generated dialogs.

6. The `FileTimeDate()`, `FileTime$()` and `FileDate$()` script functions now all work on any view with an associated file and can return a number of different times and dates.

7. The `FileName$()` script function has been extended to allow it to return useful sections of the complete path and file name such as the entire path and the whole file name.

8. Accelerator keys (single keystrokes) for the sample pause at sweep end and sample trigger options have been added.

9. The X axis and Y axis dialogs adjust and disable controls as appropriate when the axis is set to be a scale bar.

10. An extra bit has been added to the mask parameter in the `ViewList()` script function so that you can get the view handle of the currently running script and include files included by the running script.

11. New `SampleAbsTime()` and `SampleMaxWaveRates()` script functions have been added.

12. New `SampleXConv()`, `SampleXCUnits$()` and `SampleXCPointsPerUnit()` script functions have been added to give access to the controls for sampling driven by an external ADC convert pulse.

13. Active cursor searches will now operate on data from the frame buffer when the buffer is displayed.

14. When a sampling configuration is added to the sample bar, the initial state of the Write to disk option is set by reading this option from the configuration file.

## Fixes

1. Grid controls in the customise display dialog are disabled when the corresponding axis is drawn as a scale bar.

2. The current sampling state as displayed by the states control bar is restored when the membrane analysis window closes.

3. The Load configuration right-click option in the sample bar now works again.

4. The script menu Turn Recording Off command was not bringing the new script view to the front, this is now fixed.

5. The `MeasureToXY()` and `MeasureToChan()` script functions now allow virtual channels to be used for Cursor 0 iteration

6. The PowerMAG auxiliary states support could hang if a state cycle started with the same state as the previous cycle ended with.

7. Use of the DS8R stimulator support could crash Signal if no DS8R hardware was present.

8. Scripted control of the  secondary power levels and recharge delays in the DuoMAG stimulator now works correctly.

9. The Compile Script menu item is now correctly greyed out for non-script views.

10. Measurements to XY view analysis that used expression mode for the cursor 0 iteration would give up after processing a single frame.

## Changes

1. As part of the changes needed to support multiple auxiliary states devices, the auxiliary device selector has been removed from the Sampling page of the Preferences dialog, instead the devices to be used are defined in the States section of the Sampling configuration dialog.

# Revision 7.00

## March 2018

As this is a new version, we start with a clean sheet, so there are no fixes. This version is contemporaneous with version 6.05a of Signal and inherits all bug fixes up to that point. We have tried very hard to make version 7 backwards compatible with version 6, CFS data files, sampling configurations and resource files are all 100% compatible and version 6 scripts should run without any problems.

## New

1. Support for the PowerMAG transcranial magnetic stimulators from MAG & More has been added.

2. Support for the DuoMAG transcranial magnetic stimulator from Deymed has been added

3. Support for the DS8R biphasic constant current stimulator from Digitimer has been added

4. An information window has been added to provide displays that can be used make timings and other information visible from across the lab.

5. Clamping membrane analysis is now carried out on all states if the state to be analysed is set to -1.

6. Screen prints now draw fonts at a size proportional to the available space, the view titles now draw in the same font as on screen and at a similar size to the screen.

7. Screen prints now draw Grid views much more satisfactorily, the font is scaled by the physical width of the output rectangle to get the most satisfactory column count.

8. The virtual channel, script evaluation, script debug edit variable, change colours, open/close analysis shortcuts, dynamic clamping models, customise display and windows dialogs are now all resizable.

9. Auto-averaging by states now allows you to not generate an average for state 0.

10. The communications port selector used in various places now shows better names for the ports.

11. The channel information dialog now shows the expression for virtual channels.

12. The file information dialog now shows the channels used in clamping sets.

13. New `ArrCWT()`, `ArrRange()` and `ArrMapImage()` script functions have been added.

14. Error information is now logged if a signal conditioner, auxiliary state or auxiliary telegraph support file fails to load.

15. The MagPro transcranial stimulator support now includes a recharge delay option which can be used to reduce noise during evoked responses.

16. A new `SampleMembAna()` script function has been added which allows a script to display or remove the clamping membrane analysis dialog.

17. A new `SampleConfig$()` script function has been added to retrieve the file name from which the current sampling configuration was loaded.

18. Finishing-off of arbitrary waveform output is now done in such a manner that the post-waveform level is always correct.

19. The script language has new numeric operators for bitwise not (~) and for shifting integers (<<, >> and >>>) and allows the use of `&` for `band`, `&&` for `and`, `|` for `bor` and `||` for `or`, `%` for `mod` and `!` for `not`.

20. The script language allows you to use * to multiply a string, for example `"+-"*3` is `"+-+-+-"`.

## Changes

1. Integer variables in the script language are now 64 bits in size, which allows a vastly greater range of integer values.

2. The 1401*plus* data acquisition system is no longer supported - if you wish to make use of a 1401*plus* you will have to use Signal version 6.

3. Using the `DlgEnable()` script function on an image item now hides or shows the image.

4. Marker data times are now all quantised to 1/100 of the smallest waveform data interval (or 10 microseconds if there are no waveform channels) to match what happens when marker data is saved in a CFS file. Because of the way marker data is stored, this change has reduced the maximum number of points in a frame to 40 million per channel.

5. The times of digital markers that are logged while sampling are now quantised to 1/100 of the sequencer clock interval. Because of the way that marker data is stored this has the effect of forcing the sequencer clock period to be no less than the sampled frame length divided by 40 million.

6. File views now offer the original file name in the File Save As dialog rather than the view title.

7. The `MatLabOpen()` script function now returns an error code on failure rather than halting the script.

8. Overdrawn channels only fade 75% of the way to the background colour so that they always stay visible.

9. The `Len()` script function now operates on multi-dimensional arrays.

10. The general data file handling has been adjusted to handle reading very large amounts of data much better.

# Common questions

# I've set a sampling configuration. How do I sample data with Signal?

Use the File menu New command and select a data document or click the Run Now button in the sampling configuration dialog. This opens a new, empty data file. You can start sampling either by using the Sample menu Start sampling command, or clicking the Start button in the Sampling control panel.

# Signal beeps at me when I try to open a new data file for sampling or exit from the program.

If you have a new data file, either sampling or about to sample, you cannot open another or exit from Signal. If you try to open a new file or exit from the Signal menu, the program beeps to remind you that the data file is open. A script will give an error if you try to open a new file.

This problem usually happens when you have been working with a script that samples data. Such a script may open a data file "hidden", so it is not obvious that there is a data file. Use the Window menu Show command to find any hidden windows.

# What is the Log window for?

The Log window is a special text window which is always present, though it may be hidden. It is a convenient place for storing text and is mainly used by scripts. In certain circumstances, such as difficulties when connecting to MATLAB or problem in using an auxiliary device, Signal will put extra debug information into the log window.

# When I sample data, where is it saved during sampling?

Signal saves new data in a file called DATA plus a sequence number in the path (folder) set by the Sampling page in the Preferences dialog. Alternatively, if file auto-naming is in use, the file name is set by the file automatic name generation, the search for files with the same name (for generating the unique number) is carried out in the directory set by the file auto-naming path, or the current directory if that is not set. If no path is set in the sampling preferences, the data goes to the current directory. We recommend that you set a path in the sampling preferences so that you can control where the temporary data is saved. The data is stored as a normal Signal CFS data file, but without the .CFS file extension. When you close the file, you are prompted to supply a file name and the original file is renamed if the new name is on the same drive (volume), or copied if the new name is on a different drive.

# Do I have to use scripts to use Signal?

No, you do not. Many users of Signal use the standard data capture and built-in analysis routines to capture data, and then export pictures and values using the clipboard or data files and make measurements using the cursors.

However, Signal can do a great deal more than is exposed by the menu-driven interface and often a few lines of script can automate a tedious manual procedure. If you suspect that a script could help your work, but you do not have the time or inclination to learn how to write one, you might consider using our script writing service. Contact CED for more information.

# How do I get started writing scripts?

Start by reading the Script manual as far as the alphabetical list of commands section. This should give you an overview of the script language. Use the script recorder to generate short scripts corresponding to actions such as opening a file, re-arranging windows or analysing data to see what script functions are used for these actions, and read the documentation on these script functions to learn what they do.

Next, look at the scripts supplied as part of Signal. These will give you a flavour of how the language is used in larger scripts. You can look up any key word in a script by placing the text cursor on the keyword and then pressing the F1 key.

We supply a copy of the Signal Training Course manual with each copy of Signal. This contains many script examples and explanations. It also has tutorials covering a wide range of common scripting tasks.

There are also scripts available from our web site (Click here, then click on Signal scripts in the left-hand panel). These range from tutorial examples to full applications.

We also run training coursed in the UK and around the world. Contact CED for more information.

# Which version of Windows is best for Signal?

Signal runs under Windows XP (SP2 required), Vista and Windows 7 and 8, plus all 64-bit versions of Windows. Signal is a 32- or 64-bit program and it functions well with any of these systems. Currently CED would recommend Windows XP or Windows 7 for use with Signal.

# Why can't I make contact with my CyberAmp from Signal?

Assuming that the correct communications port is set in your CEDCOND.INI file, and that you installed Signal and selected CyberAmp support, the most common reason for lack of communication is that the rotary switch on the rear of the CyberAmp is not set to 0.

We have also seen problems on computers where the COM port has been set up in the computer BIOS as an infrared communication device.

# Why does my CyberAmp with input probes 401, 402, 405 and 414 give no signal?

These probes are described as differential in their specification, so you have selected the differential input setting. However these probes convert the signal to single ended. If you select the differential input, the same signal is applied to both CyberAmp differential inputs, and the result is to test the common mode rejection of the system. Select single ended or inverted inputs with these probes.

# If I zoom in to show a very small (and we mean very small) x axis range, then print the data, the printed output sometimes shows more data than there was on screen. Is this a bug?

No. When you look at a file or memory view that is zoomed in the x (time) direction, you are looking through a rectangular window into a longer view, through which you can scroll. This underlying view can be around 2,000,000,000 pixels wide. If you keep zooming in on the x (time) axis, the limit to how far you can go is the size of the underlying view. The more pixels per inch your display has, the narrower the underlying display when you are at maximum zoom.

Although this is a limitation, to put it in perspective, for most common display, the limiting width of the underlying view when you are at maximum zoom is around 400 miles (around 600 km). Imagine a strip of chart paper that long!

When you print a picture, the same limit applies, but a printer has a higher resolution than the typical screen (typically 4 to 10 times as many pixels per inch). Thus the maximum zoom on a printer is typically 4 to 10 times less than for the screen. When you try to print in this case, Signal adjusts the times to show more data. This effect is rarely of any consequence.

# If I sample data fairly fast (total rates of around 100 kHz or more) the mouse response gets very jerky or sluggish. Why and what can I do about it?

If you have the 1401 ISA interface card (the standard interface card supplied with a 1401), the maximum data transfer rate from the 1401 is around 250,000 bytes per second. These transfers normally happen using DMA (Direct Memory Access) which does not burden the processor too heavily.

DMA cannot be directly used with memory above 16MB on the ISA bus, so if your computer has more than 16 MB of memory, and the data for the transfer happens to lie above 16 MB, the data must be transferred by the processor. This can occupy a significant amount of processor time and make the system response very slow or jumpy.

First of all, ensure that you have got the latest Windows 1401 device drivers, you can contact CED or visit the CED web site for more information. With the latest 1401 device drivers, the 16 MB limit is evaded by using a separate DMA buffer for transfers above 16 MB. This is almost as fast as direct DMA and will solve most of the problems.

The best solution is to use a faster interface. There are three other types of interface for the 1401 that have been introduced recently: the Quad rate ISA card which is capable of moving memory much more quickly (up to 4x faster, but still has the 16 MB limit), the PCI interface card, which can transfer data much more quickly and can do DMA to any portion of memory and the USB interface. The Quad rate card operates under any PC operating system and is 100% compatible with the standard ISA card. The PCI interface card is also much easier to install - it has no configuration jumpers, and is likely to be the only interface card usable with even a relatively new PC. The USB interface does not require any interface card at all and is available with all modern types of 1401 - it is generally the preferred interface.

Contact CED if you need more information about these interface options.

# Signal works fine looking at old files, but when I try to sample data it fails or gives an error message. How do I sort out 1401 problems?

Along with Signal, the installer provides a program, TRY1432, that tests the use of the 1401. Run Try1432 and click on the self-test check box so that all the tests are selected. Click on the Run Once button to run the tests; any

error messages will be displayed in the lower part of the Try1432 display. Contact CED with the Try1432 test results and details of any Signal error messages to resolve the problem.

# When I try to read a file created by Signal with SIGAVG, modify it with MCF or convert it with C2S, these programs cannot access the file data. How do I ensure compatible files?

Signal creates and uses CFS files, which can be read and understood by many programs. However, many programs have limitations on the types of CFS data they can use. Signal can read and write CFS data files using 2-byte integer or 8-byte real data, but all of the DOS programs mentioned can only handle integer data.. Signal data files created by sampling will always use integer data, but Signal has a choice of what data type to use for other files, in particular when saving memory view data to disk. There is a field in the Data page in the Preferences dialog to control what data type is used in these circumstances, set it to Integer to always create data files compatible with older DOS programs.

# I get error -544 when I try to sample in Windows NT, NT 2000 or Vista. Why?

This error code usually means that the system has been unable to lock down memory needed for data acquisition. Signal tries to avoid this problem by requesting an increase in the *Minimum Working Set* of the program, but this request can fail if:

- The size requested is too large for the system to operate safely.
- The user does not have the right to change the allocation (see below).

The *About Signal...* command in the Help menu shows the current Minimum and Maximum Working Set sizes set. Signal attempts to set the Minimum Working Set to 800kB and the Maximum Working Set to 4000kB, but you can override these values in the registry. As this is an advanced option with system-wide implications, we do not provide an easy way to do this from inside Signal.

The most usual cause is that you are logged on as a User with insufficient privilege to increase your Working Set.

### What is the Working Set Size

You can think of the Working Set Size as the amount of physical memory allocated to a process (in this case the Signal program). A program may use many megabytes of memory space (logical memory), but only the memory needed for immediate use has to be present in physical memory, the rest can be saved on disk and loaded when needed. In general, the more physical memory a program can use, the faster it runs as it does not need to wait for information to be loaded from disk.

However, Windows is a co-operative environment, and if any process (program) grabs a lot of physical memory, there is less for others to use, and in extreme cases, the system may grind to a halt. As far as the operating system is concerned, it likes all processes to use the minimum possible physical memory. Each process (like the Signal program), is allocated a minimum working set size and a maximum working set size which the operating system uses as a guide to how much physical memory a process uses.

### Minimum Working Set Size

This is the amount of physical memory that the operating system will always try to give a process, even when memory is running very low or your process is not running. If there are many processes running, the sum of the minimum working set sizes must always be less than the amount of physical memory in the system. The value given a process by the system by default depends on the amount of physical memory in the system and can be as low as 120kB, and lower in some circumstances. Signal needs to lock down around 68kB of memory when it samples data, so this leaves very little to run the program. If the minimum working set size is too small, the operating system will refuse to lock the memory for sampling, resulting in error -544.

### Maximum Working Set Size

The maximum working set size is the maximum amount of physical memory that the operating system normally lets your program have if memory is in short supply. If there is no other demand for memory, your process can have more memory than this. The default values for this can be quite small, values of around 1 MB are common.

## What Signal tries to do

When Signal starts up, it tries to set the minimum working set size to 800 kB (enough to sample data and run), and the maximum size to 4 MB. You can override these sizes in the system registry. You must create and set the registry keys as `DWORD` values:

```
HKEY_CURRENT_USER\Software\CED\Signal\Win32\Minimum working set
HKEY_CURRENT_USER\Software\CED\Signal\Win32\Maximum working set
```

to the required sizes in kB that you need. If you do not supply a key, the default size will be used. If the size you request is too large for the system to grant, the request will be ignored for both parameters. Signal will never set the sizes less than the sizes the system gives you by default. If you need to find the default system sizes, set the sizes in the registry to 0, run Signal and open the About Signal… window.

You can set these keys with `RegEdit` which is part of the Windows operating system. Signal does not create these keys, but will use their values if they exist.

If you run large scripts that use a lot of memory, you can improve performance by increasing the maximum size, but if you overdo it, you will reduce system performance by starving other processes (including system processes) of memory.

## What to do if you cannot set the size

If the *Help* menu *About Signal* box does not report 800, 4000 kB (or the sizes you have requested in the registry), then either you are not running under Windows XP, NT or NT 2000, or you do not have sufficient "rights" to change the allocation. The following describes how we altered the rights on our system. Your system administrator will know how to do this, but they might find the following will save them some time checking the system manuals.

We describe how to do this for the local machine: it may be that your machine is administered remotely in which case the general description is correct, but the details may be different.

The new right you need is *Increase scheduling priority* (in low-level system documentation this is called the *SeIncreaseBasePriorityPrivilege* privilege and in programmer information it is also known as *SE_INC_BASE_PRIORITY_NAME).*

Rights are assigned to user Groups, so you must either add this right to the list of rights enjoyed by the Group you are a member of, or you must create a new Group with this right and become a member of it. In the descriptions below, steps to create a new group are labelled X. To extend the rights of an existing group, skip the steps labelled X and substitute the name of the existing Group for all mentions of *1401 Users*. The details (we include information about older versions of Windows for reference) are:

### Windows NT 4

1.  When logged on as Administrator, run the *User manager*, which is in *Programs:Administrative tools (common).*

X1  Use the *User:New Local Group* menu command to create a local group called *1401 Users*.

X2  Add those users who need to use Signal to the list of members of this group.

2.  Use the *Policies:User rights* menu command to display the *User rights* dialog box.

3.  Check the *Advanced user rights* check box at the bottom of the dialog.

4.  Select *Increase scheduling priority* in the *Rights* list box.

5.  Press the *Add* button and add *1401 Users* to the list of groups which possess this right.

6.  Exit from *User manager*.

### Windows 2000, XP, Vista and Windows 7/8

1.  When logged on as Administrator, open the *Control panel*, switch to *Classic View* if using *Vista* and then open *Administrative tools*.

X1. Run *Computer management*, select the entry *System tools:Local users and groups*.

X2. Select the *Groups* folder, use the *Action* menu to add a new local group called *1401 Users*.

X3. Use *Add…* to make the appropriate users into members of the new group.

X4. Exit from *Computer management*

2.  Run *Local Security Policy*, select the entry *Local policies*, subentry *User rights assignment*.

3.  Select *Increase scheduling priority*, use *Action : Security...*in NT 2000, use *Action : Properties* in XP or Vista.

4.  Click on *Object types...*, check the *Groups* box then click *OK*.

5.   Add *1401 Users* to the list of groups with that user right.

6.   Exit from *Local Security Policy.*

Next time you log on as a member of *1401 Users*, you should find that *Help:About Signal* shows the expected memory available and that the error -544 no longer occurs.

**What to do if it still doesn't work**

Don't panic! It works for everyone else, so there is a logical reason for failure. Please check that you have the latest service pack for your OS installed; a customer had error -544 problems with a Dell system running NT 2000 service pack 1 that refused to work until service pack 2 was installed.

# Why aren't the external digital states working?

The most common cause of this is that an input has been enabled which is not connected.

# Why are the file paths in my script not behaving correctly?

A common scripting problem is strings holding a file path that does not work. This is often because the \ character (which is needed within file paths) is treated specially when string constants are generated or a format string is interpreted in Print() or Print$() - it indicates that the next character is to be interpreted to produce a special character. For example "\r" is converted to a single carriage return character, "\t" is converted to a tab character.

To get a single \ character you need to put two of them into your string constant thus "\\", and if you want to end up with a single \ in a file path generated via a Print$ you need to have four of them:

```
path$ := Print$("C:\\\\Signal\\\\%s", filename$);
```

this is because the string is interpreted twice; once to produce the string constant that is fed to the  Print$ function, and once by Print$, and each time pairs of \\ characters are converted intro single ones.

# License information

CED software is protected by both United Kingdom Copyright Law and International Treaty provisions. Unless you have purchased additional licenses as described below, you are licensed to run one copy of the software. Each copy of the software is identified by a serial number which is displayed by the Help menu About Signal… command. You may make archival copies of the software for the sole purpose of back up in case of damage to the original. You may install the software on more than one computer as long as there is **No Possibility** of it being used at one location while it is being used at another. If multiple simultaneous use is possible, you must purchase additional software licenses.

**Additional software licenses**

The original licensee of a CED software product can purchase additional licenses to run multiple copies of the same software. CED supplies an additional manual set with each license. CED does not supply additional software media. As these additional licenses are at a substantially reduced price, there are limitations on their use:

1.   The additional licenses cannot be separated from the original software and are recorded at CED in the name of the original licensee.

2.   All support for the software is expected to be through one nominated person, usually the original licensee.

3.   The additional licensed copies are expected to be used on the same site and in the same building/laboratory and by people working within the same group.

4.   When upgrades to the software become available that require payment, both the original license and the additional licenses must be upgraded together. If the upgrade price is date dependent, the date used is the date of purchase of the original license. If some or all of the additional licenses are no longer required, you can cancel the unwanted additional licenses before the upgrade.

5.   If you are the user of an additional license and circumstances change such that you no longer meet the conditions for use of an additional license, you may no longer use the software. In this case, with the

agreement of the original licensee, it may be possible for you to purchase a full license at a price that takes into account any monies paid for the additional license. Contact CED to discuss your circumstances.

6. If you hold the original license and you move, all licenses are presumed to move with you unless you notify us that the software should be registered in the name of someone else.

# Index

## – E –