

Signal for Windows

Training Course manual

May 2011

Copyright © Cambridge Electronic Design Limited 1998 - 2011

The information in this book was prepared and collated by the staff of Cambridge Electronic Design Limited as teaching material for the Signal Training days held in Cambridge and elsewhere. We hope that you find the materials useful and instructive, if you have any comments on this book, please pass them on to Tim Bergel (tim@ced.co.uk) at CED.

Version 1 September 1997
Version 2 September 1998
.
.
Version 14 October 2007
Version 15 March 2008
Version 16 October 2008
Version 17 May 2011

Published by:

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

Telephone: Cambridge (01223) 420186
Fax: Cambridge (01223) 420488
Email: info@ced.co.uk
Web site: www.ced.co.uk

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

Preface

This book is a compendium of material presented at the Signal training days held in Cambridge, England and elsewhere. This material, particularly the scripting sessions, expands on the information in the standard manuals. We hope you find this book useful and instructive.

If you are attending a training day course, the script examples are available on disk and are described here so that you can review them in your own time. The book should help to remind you of the topics covered. For those not attending the training day, the script examples can be downloaded from the CED web site and the book treated as an extra introduction to Signal.

Introduction to Signal.....	5
Sampling data with Signal.....	12
Advanced sampling	18
Analysis menu	22
Advanced analysis.....	27
Patch and Voltage Clamp.....	34
Script Introduction.....	45
Script toolbox	56
Scripts and Signal data	65
Signal scripts online	72
Appendix - DOEMG - a worked example	78

Introduction to Signal

This session introduces the Signal program, demonstrates the use of menu items and explains the ideas behind Signal.

Sampling data with Signal

This session explains how to configure Signal to sample data. It covers the various ways that Signal can sample data and generation of output pulses and waveforms during sampling. It also shows how to set up Signal to analyse data on-line.

Advanced sampling

This session looks at some of the more complex mechanisms available within Signal sampling, in particular the use of multiple states.

Analysis menu

This session provides an in-depth look at the features available from the analysis menu in Signal, including waveform averaging, generation of power spectra and direct data modification.

Advanced analysis

This session looks at active cursors, trend plots, curve fitting and digital filtering.

Patch and Voltage Clamp

This session looks at leak subtraction as well as formation and analysis of idealised traces including subsequent histogram formation.

Script introduction

This session introduces you to the mechanics of using the script language, introduces various concepts central to the script language and describes the script language keywords.

Script toolbox

This session builds on the script introduction to show how to use the library of built-in functions.

Scripts and Signal data

This session introduces covers access to data in Signal files and analysis mechanisms.

Signal scripts online

This session covers the techniques required to write scripts that control or interact with the data acquisition process including online analysis and control of pulse outputs.

Appendix

A complete script that would be a useful starting point for developing your own script.

Introduction to Signal

Introduction to Signal

This talk introduces a new user to Signal. Signal has many features; too many to go into in any detail in the time available. The purpose of this session is to give you an outline of what Signal is, what it can do, how to get started with it and where to find items in the menus.

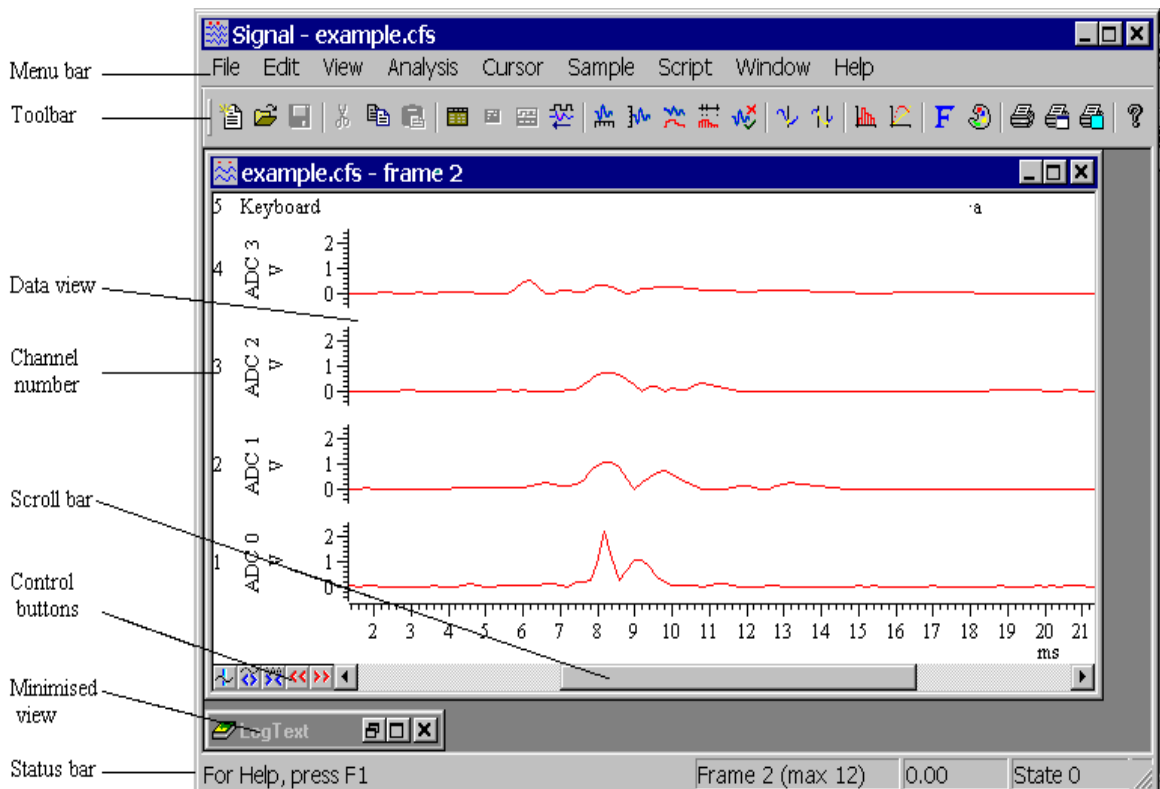
What is Signal

Signal is many things to many people. To some it is a simple data capture and review program, acting like a computerised oscilloscope. To others it is a user-customisable data analysis engine, capable of working rapidly through many megabytes of data to reduce it into a manageable form.

The basic concept behind Signal is that it handles frame-based data; data that consists of multiple sections or frames, where each frame is (usually) very similar to all the others. For example, a Signal data file might hold 100 frames, each frame holding two waveform channels of data sampled for one-tenth of a second. Times for data are frame-relative, with each frame covering the same time-range; from 0 to 0.1 seconds.

Basic screen layout

When you launch the Signal application you will see a blank screen except for the menu, toolbar and status bar. The picture below shows some of the features you will see when you work on data files:



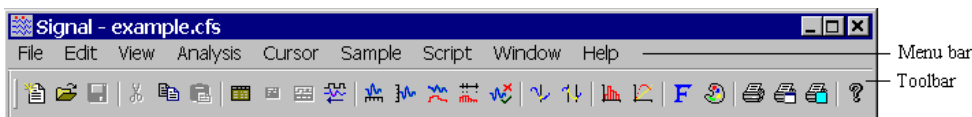
You use the menu bar to command the program as you would any other Windows application. The toolbar immediately below provides short cuts to menu bar commands. The status bar at the bottom provides feedback (for example it shows the function of a toolbar button under the mouse pointer) and information about the data displayed in the current data view.

The data view occupying the central area shows previously sampled data read in from a file. We refer to such a window in Signal as a **File view** because the window shows data from a file, data held wholly in memory is shown in a **Memory view**. You can have multiple channels of data in a data view, each channel has an associated channel number that uniquely identifies the channel.

At the bottom of each data view is a scroll bar and the control buttons. You use these to choose the section of data that is displayed in the window, to move through the frames and to create a new cursor. You can hide the x-axis and the controls from the View menu Customise display command.

The menu bar

The menu bar in Signal holds the lists of commands available from the application. The menu and the associated toolbar are the simplest means of accessing the functionality available from Signal.




The illustration above shows the menu bar and its associated toolbar (the line of buttons). The toolbar provides short cuts to the more common functions available from the pull down menus. It can be hidden to give more room for the data views if necessary. To find out what a toolbar button does, leave the mouse pointer on top of the button for a few seconds. Dimmed icons in the toolbar correspond to functions which are not available. The available functions vary with the active window. The menus are:

- | | |
|-----------------|--|
| File | From here you can load, save and print documents and create new documents and save and load sampling configurations. There is also a recently used file list and if you are attached to a Mail server you can send the current document by email. |
| Edit | The cut, copy and paste commands are found under this menu plus commands for searching and replacing text. An important item in this menu is the preferences which are used to configure system options such as where new data files are stored until you save them permanently. |
| View | View manipulation commands which control the look of the data windows and what data is displayed; many of these commands are duplicated by keystrokes or by clicking with the mouse. |
| Analysis | The main built-in analysis functions live here. These range from averaging and power spectrum generation to creation of trend plots, channel data manipulation, frame buffer operations and marking data frames for analysis. |
| Cursor | Cursor values and dual-cursor measurements such as areas, means and slopes can be accessed here, in addition to commands manipulating the cursors themselves. |
| Sample | Setting up of the sampling configuration and any signal conditioners is done here. This menu also duplicates the functions of the sampling control panel. |
| Script | Use this to evaluate (execute) one line of script, record your actions or to run a script. You are allowed to load as many scripts as you can fit in memory using the File menu. From here you can select one to run, or load and run a script directly. |

-
- | | |
|--------|---|
| Window | This contains commands for laying out windows, duplicating views and hiding or showing views. |
| Help | This gives access to the help contents, index and a keyword search mechanisms. This also includes information about the version of Signal that you are using. |

Using Help

Signal has a comprehensive help system installed along with the main software. You can use the help menu or the toolbar button  to access it directly and search for the information you need or use the context sensitive help. Context sensitive help is available for any dialog by pressing F1 while using the dialog and also from the script and evaluate windows. To use script context-sensitive help, place the cursor somewhere in the script text and press F1. If Signal recognises the text surrounding the cursor as a script keyword it will jump straight to the related help section. Keywords include all of the built-in functions.

The on-line help contains almost all of the user manuals that are supplied with Signal and will generally be more up-to-date than the printed documentation.

About channels

A Signal data file holds a number of channels, each of which holds a particular type of data. These are the different channel types that Signal can sample and derive from other channels.

- | | |
|-----------|--|
| Waveform | A waveform is stored as a list of data points, normally 12 or 16-bit ADC data. The points are equally spaced in time, normally all waveform channels in a file have the same spacing. |
| Marker | A Keyboard marker is a time stamp with four associated codes in the range 0-255. The first code is used to hold the ASCII code for a key. Each time a key is pressed during sampling when the sampling data window is active, the key value is saved in the file at the key-press time. A Digital marker is much the same but the values are sampled by the 1401 when a suitable trigger is given. |
| Idealised | Not actually part of the data file itself, an idealised trace is generated for the purposes of patch clamp (single channel) research by detecting level transitions in a data file channel. |

A typical Signal data file might contain a number of channels of waveform data (voltage information) and a keyboard marker channel which contains key presses indicating various stages in the experiment.

About frames

A Signal data file contains a number of frames of data, each of which holds the same channels. Only one frame of data is displayed at a time in a data view, this is the current frame for the view. If you wish to display two different frames from the same file, use **Window Duplicate** to generate a second view and then set each view to a separate frame. In addition to the channel data there are a number of other data values attached to the frame; for example a single-line comment, a state code and a tag flag.

Frames are central to Signal; all of the built in analysis mechanisms are designed to analyse data from a set of frames (the same section of data from each frame being used) and Signal data sampling is wholly frame-based.



Control of data views

It is possible to manipulate the data views using the mouse to move through the data rapidly to interesting areas using the scroll bar and thumb. There are also View menu functions to optimise specific channels or all channels along with duplicate views to enhance the screen display. Many controls have short-cut key equivalents.

Keyboard display control shortcuts

Scroll data down / up	Cursor down / cursor up
Decrease Y range / increase range	Ctrl cursor down / Ctrl cursor 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 cursor left / Ctrl cursor 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
Hide selected channels	Del
Customise display	Ctrl Del

Mouse control

Many screen display functions can be accessed by using the mouse. You can double-click on the X or Y axis to access the axis control dialogs, or right-click in the data area to get a pop-up menu. Drag functions allow you to zoom the channel by clicking and holding down the mouse button on the data areas of the screen. Dragging the mouse pointer  in any direction will zoom into the data area. Zooming out requires the Ctrl key to be pressed while dragging and the pointer will appear with a negative symbol . These drag operations can be undone by using the Edit menu Undo command. Double clicking with the mouse on a channel data area zooms or un-zooms the channel.

The x-axis can be doubled in length, or halved in length using the zoom-in and zoom-out buttons in the bottom left corner of a data view. The adjacent previous and next frame buttons move through the frames in a file. The scrollbar at the bottom of a data view indicates the current display position with respect to all of the data in the frame and can be used to move through the frame.

Copy and paste


You can use the Windows clipboard to move data between frames or files. The copy operation saves all of the waveform data visible on the screen to the clipboard, the paste operation will overwrite visible channels with clipboard data. Aspects of the channel data such as channel titles or units are not copied. Data on the clipboard can also be pasted into the waveform outputs control panel.

XY views

In addition to the File and Memory data views, Signal can also generate XY views. An XY view shows sets of (x, y) values as points, for example as a trend plot or scatter-graph. XY views can be created using the analysis menu and are also a valuable tool for use within Signal scripts.

Measurement tools

The simplest way to take measurements from Signal data is directly; using the mouse pointer. If you move the mouse crosshair cursor about within a data or XY view, you will see changing text in the status bar looking somewhat like 1,12.842,1.057. The three numbers are the channel, X position and Y position of the mouse pointer. Difference measurements can also be taken by holding down the Alt key and 'dragging' a rectangle across the data view; the size of the rectangle in X and Y axis units will be displayed adjacent to the mouse pointer.

Measurements can be taken from data views by using cursors. Up to 10 cursors can be created in a window using the **Cursor...New cursor** menu option, or by clicking the cursor button  in the bottom left corner of the data view.

Cursor measurements can be taken at the position of each cursor, using the **Cursor...display Y values** menu option which displays the measurements in a new window. The cursor times and y-values can be set relative to any particular cursor by checking the **Time zero** and **Y zero** boxes respectively and selecting the reference cursor to use with a radio button.

Cursor measurements can also be made between pairs of cursors using the **Cursor...Cursor regions** menu option. A new cursor measurements window is created in which one of various types of measurement can be made between adjacent cursors. For waveform channels, the measurements are:

Curve area The area over the baseline made by joining the points where the cursors cross the data trace, with sections below the baseline subtracted. Each waveform point makes a contribution to the area of its amplitude above the baseline multiplied by the time between samples on the channel. Previously this was defined as **Area**.

Mean The mean value of all the waveform points in the region.

Slope The slope of the least squares best fit line to waveform points in the region.

Area The area over the zero baseline, with sections below zero subtracted. Each waveform point makes a contribution to the area of its amplitude multiplied by the time between samples on the channel. If a zero region is specified, the amount subtracted from the other regions is adjusted for the relative widths of the two regions. Previously this was defined as **Area/0**.

Sum	The sum of all the waveform points in the region.
Modulus	The area over the zero baseline, but with all values absolute. Each waveform point makes a contribution to the modulus of its absolute amplitude value multiplied by the time between samples on the channel. If a zero region is specified, the amount subtracted from the other regions is adjusted for the relative widths of the two 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.
Amplitude	The value shown is the difference between maximum and minimum values found between the cursors.
SD	The value shown is the standard deviation from the mean of the values found between the cursors. If there are no values between the cursors the field is blank.
RMS	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.
Extreme	The value shown is the maximum absolute value found between the cursors. Thus if the maximum value was +1, and the minimum value was -1.5, then this mode would display 1.5.
Peak	The value shown is the maximum value 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.

These measurements may be made relative to a zero region defined by checking the **Zero region** box and selecting a cursor region with radio button.

Data from the cursor measurements or regions window can be copied to the clipboard by selecting sections of the window and using Edit:Copy. You can select sections by clicking on them, rows and columns by clicking on a row or column heading and all of the window by clicking in the top left rectangle.

Active Cursors

You can set up the Signal cursors to be active, so that they automatically move to a new position in each data frame, for example to the position of the maximum in the data. This facility can be very useful in taking measurements from the data or in generating trend plots. In addition to having cursors move automatically when moving from one frame to the next it is also possible to have a cursor which iterates through features within each frame whilst measurements are taken for an XY plot. More details of this are given in the Advanced Analysis chapter.

Exporting graphics and text to other programs

You can export data from Signal as graphics or text suitable for import into common spreadsheet or word processing packages such as Microsoft Excel or Word. Data can be exported using the clipboard (using **Edit:Copy**), or by means of files (using **File:Export As**). The different formats in which data can be exported are:

Bitmap images

Bitmaps are OK, but they suffer if you stretch them and if lines are not horizontal or vertical, they tend to look rather jagged when printed. This is because the bitmap is limited to the resolution of the screen. The device independent bitmap option that appears when pasting uses the same resolution but contains extra information to allow colours to be adjusted for the destination.

Vector images

These are referred to as Picture when pasting. They are made up out of lines which can be scaled to suit the document without losing resolution. When you print a vector image, you get the full resolution of the printer, not the original resolution of the screen and usually a much better result. You can also import a vector image into a drawing program and then edit the fonts and line thickness.

Text output.

Text is generated in rows and columns in a format suitable for Excel.

Sampling data with Signal

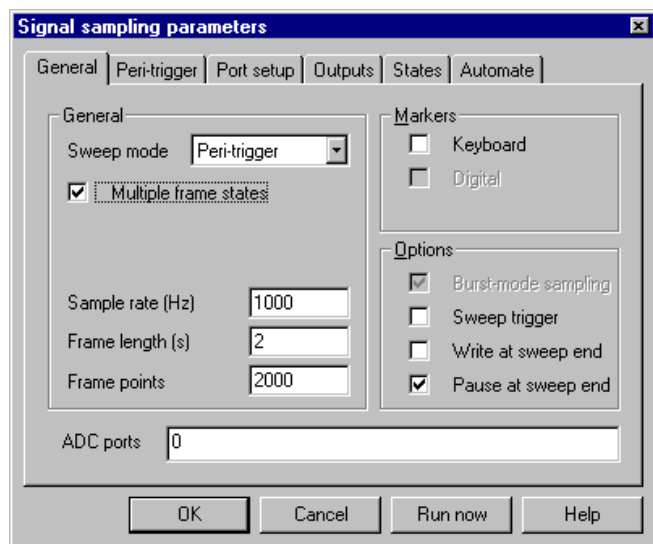
Sampling data with Signal

This session will investigate how data can be sampled with Signal. Data sampling is the process by which Signal captures external signals using any of the CED 1401 data acquisition interfaces, displays and analyses these signals in real time, and records the signals on the hard disk of the host computer. Signal can also generate output pulses synchronous with the sampling and can have multiple sweep states to distinguish between different experimental conditions. Data is sampled by creating a new file which causes the system to prepare to sample data in the manner set up in the Sampling configuration dialog.

Configuring Signal sampling

Using the **Sample** menu you can access the sampling configuration dialog, from which you can set up sampling parameters such as the number of channels to sample, the points per channel and the sampling rate. The sampling configuration dialog is rather complex, so I shall not be able to describe it all, see the main Signal documentation or the help file for complete coverage. The dialog is divided into sections which are selected by clicking on the tabs at the top. The sections are:

General This holds a selector for the sweep mode, the multiple states enable checkbox, plus items to set the sweep length, sampling rate and channels and selectors for various other options. The same sampling rate is used for all waveform channels. With some sweep modes it is possible to define different sweep lengths for different states.



Peri-trigger If the sweep mode is Peri-triggered, this section allows you to set the pre-trigger points and the trigger type and parameters.

In addition to sampling points before the trigger, peri-triggered mode allows triggering by a threshold crossing on a sampled channel, which can be very useful.

Port setup This section allows you to calibrate and name the data sampled from each ADC port, and provides access to the signal conditioner, if available and telegraph output controls.

Outputs This section is used to define voltages and pulses output during sampling. There are two mechanisms available within Signal for generating outputs; you can use the built-in pulse editor to generate sets of pulses by manipulating a graphical representation of the outputs that will be generated or, for very complex behaviour, you can generate the outputs directly using a sequence: a set of low-level instructions that will be executed by the 1401 while sampling is in progress. We will look at the use of the graphical pulse editor in more detail below.

- States** If multiple states are enabled, this section allows you to set the number of states and how the states are sequenced. See the Advanced sampling chapter for more about multiple states.
- Automate** This section allows you to set limits which, when reached, will cause sampling to stop automatically and to set a template used to generate names for new files. It also provides controls for automatic artefact rejection.

The sweep mode

The selector for Sweep mode in the General page defines the mode or style of sampling used. Currently there are seven modes available; Basic, Peri-trigger, Outputs frame, Fixed interval, Fast triggers, Fast fixed interval and Gap-free:

Basic mode

In Basic mode the trigger, if any, for a sweep of data capture is an external TTL pulse on Event zero at the start of the sweep. Any pulse outputs start and finish at the same time as the sweep. As the name implies, this is the simplest mode of operations.

Peri-trigger mode

In Peri-trigger mode the trigger point for a sweep can be located at any point within the sweep, allowing pre-trigger data to be collected. A variety of forms of trigger can be used, ranging from a TTL pulse to at threshold crossing on a sampled waveform channel. The pulse outputs start at the time of the sweep trigger, not at the start of the sweep, and run to the end of the sweep.

Outputs frame mode

In Outputs frame mode it is the pulse outputs which are triggered, the trigger being a TTL pulse on Event zero. The sampling sweep is started at a defined point within the pulse outputs. This allows output pulses to be generated both before and after the sampling sweep. This mode allows variable sweep lengths as an option.

Fixed interval mode

Fixed interval mode is just like Outputs frame mode, but the triggers are generated internally using a timer in the 1401. The interval between triggers can be constant or a random variation within specified limits can be provided. This mode also allows variable sweep lengths as an option.

Fast triggers mode

Fast triggers mode is like Basic mode except multiple frame states and incremental pulsing are not available. This keeps the inter-sweep interval to a minimum.

Fast fixed interval mode

Fast fixed interval mode is like Fast triggers mode but uses a fixed interval between sweeps rather than requiring an external trigger.

Gap-free mode

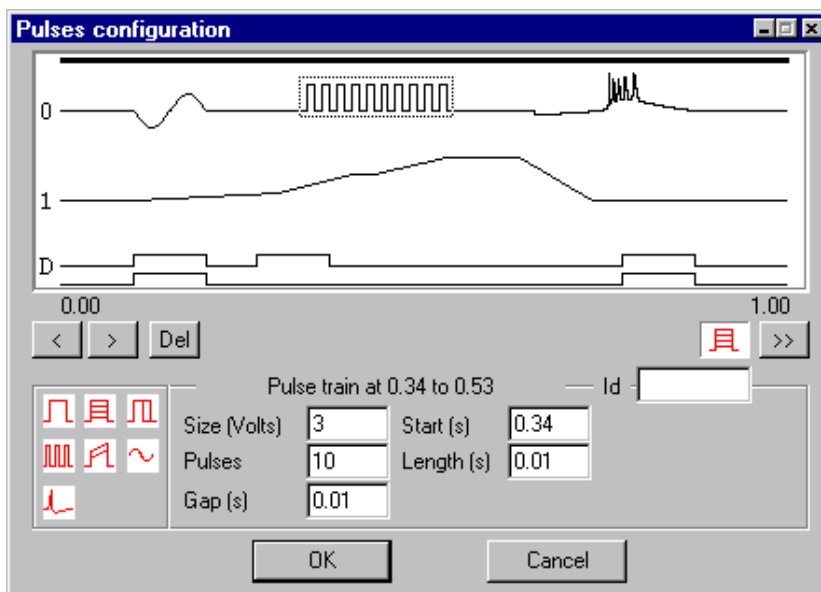
This is like Fast fixed interval mode except that each sweep starts as soon as the previous one has finished. This happens in such a way that the interval between the last point of one sweep and the first point of the next is the same as the interval between sample points within a sweep.

Generating outputs while sampling

Users often need the 1401 to generate outputs while Signal is sampling a sweep of data. Signal can output static levels, square or ramp pulses, sine waves, pulse trains and arbitrary waveforms using the 1401 DACs and digital outputs.

The outputs are configured using the Outputs page in the sampling configuration, which holds calibration information for the DACs and enable buttons for the various outputs. The Outputs page also provides the pulse configuration dialog. This displays the outputs graphically, provides a 'toolbox' of pulses that can be inserted and also a parameters area for adjusting the pulse data directly.

The dialog allows pulses to be added, selected, moved or removed by clicking or dragging using the mouse. The parameters for the selected pulse can be edited directly.



The types of pulse available are:

Initial level

This specifies the state that the outputs are set to before the sampling sweep starts, this item is always present and cannot be inserted, deleted or moved. The initial level has a **Level** parameter 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, as are all DAC values. It also has **Step change**, **Repeats** and **Steps** parameters. These are described below user Pulse variations. If variable sweep lengths have been set then this is also the place to set the number of **ADC points**. For digital outputs this item is **Initial bits**.

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. **Start**, **Size** and **Length** parameters define the pulse.

Square pulse with varying amplitude

This specifies a square pulse whose amplitude varies automatically. The pulse definition parameters are the same as for the simple pulse, in addition there are parameters controlling the built-in variation. This type of pulse is not available for digital outputs. The behaviour of the variation is described under *Pulse variations*, below.

Square pulse with varying duration

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.

Square pulse train

This specifies a series of non-varying square pulses, with parameters to set the number of pulses and the pulse spacing. This type of pulse is available for DACs and for the digital outputs.

Ramp pulse 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 can be applied to either the pulse start or end, or both. This type of pulse is only available for the DACs.

Sine wave segment

This specifies a sine wave output of fixed duration, amplitude and frequency, it is only available for the DACs.

Arbitrary waveform

This 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 synchronously. The data arrays can be set up in two ways; by using the Signal script language functions or by copying and pasting Signal data with the clipboard. Only one arbitrary waveform item can be inserted into a set of outputs.

Pulse variations

Some pulse types can be set up so that they vary automatically, providing an easy way to generate a range of pulses. The pulse types that can vary are initial level, square pulse with varying amplitude, square pulse with varying duration and the ramp pulse. All of these use the same parameters to control the variation, only the varied aspect depends upon the pulse type. The pattern of variation is: the pulse is generated n times using the pulse values set, then n times with one 'step change' added, then n times with two steps and so on. This repeats until the maximum number of changes has been reached, at which point the cycle restarts with the basic pulse values. The number of changes can be set to zero to give no variations.

Sampling with multiple states

Normally each sweep of sampling uses the same set of pulses, but when using multiple states sampling you can define a number of different sets of output pulses and switch between them within one file. This powerful facility is covered in detail in the Advanced sampling chapter.

Sampling data

The File menu **New** command is used to create a new document, if you select a **Data** document then a new data view will be created in which data will be sampled. Alternatively, you can press the **Run now** button in the sampling configuration dialog. Control over the sampling process is provided by means of a floating control panel or from the **Sample** menu. In addition to the sampling control panel, the pulses configuration dialog can be used to adjust the output pulses during sampling.

Sampling control panel

The sampling control panel is used to start, control and stop data capture. Sampling is started by clicking the **Start** button. If the **Event 1 start** box is checked, after the Start button is pressed the sampling is actually started by supplying a trigger pulse to the Event 1 input of the 1401.

The **Abort** button can be used to discard the new file and kill off sampling immediately. The sweep trigger item enables and disables sweep triggers.

Once data capture has begun, the options change to include **Stop** data capture (end the experiment) or **Restart** (discard all data accumulated so far and prepare for a new recording).

If sampling is set to write sweeps to disk automatically at the sweep end, the **Reject** button can be used to override this and discard sweeps. If sweeps are not written automatically, this button changes to **Accept** and can be used to select sweeps.

If sampling is set to pause at the sweep end, when each sweep ends the **Continue** button is enabled, press this to allow the next sweep to begin. While sampling is paused the **Accept/Reject** button can be used to add or remove the sampled sweep from the file.

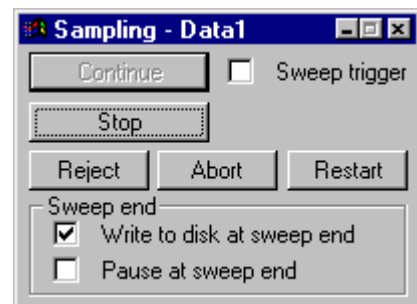
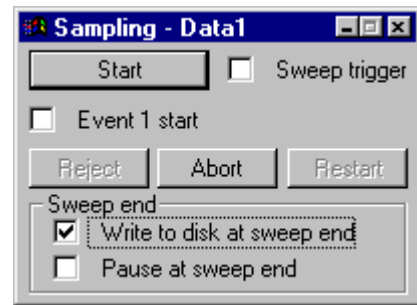
Frame zero

A sampling document is different from other data documents because it starts at frame number zero while all others start at frame 1. Frame zero is a special frame that holds the transitory data for the sweep currently being sampled whereas frames 1 onwards hold data that has been written to the new file.

Finishing off sampling

Sampling can be stopped by pressing **Stop** or by any of the sampling limits being reached. When sampling stops, the sampling control panel changes to provide **More** and **Finish** buttons. Pressing **More** resumes data acquisition while pressing **Finish** finally finishes off the sampling. When sampling finishes, frame zero of the data document disappears. If there are no saved frames the data document and view are destroyed, giving the same effect as pressing **Abort**. If there are saved frames and the appropriate checkbox is set in the **Preferences** dialog, a dialog will be provided for entry of the frame comment.

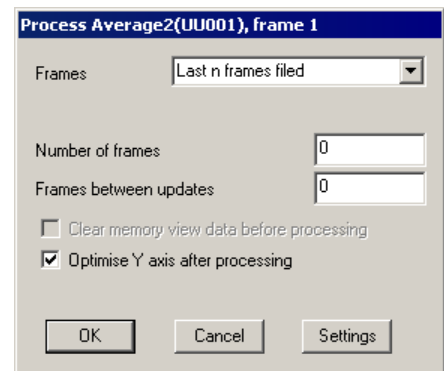
A sampling document which has finished sampling is essentially the same as a document loaded from disk. However (unless the **Automate Save file to disk** option is enabled) the data has not yet been saved in a permanent disk file, though it is stored on disk. To keep the data, you must save the new data using the **File** menu **Save** or **SaveAs** commands. If you try to close the document without saving the data Signal will check that this is really what you want to do.



Analysing sampled data on-line

You can set up any on-line analyses required and customise the display before starting the sampling by pressing the **Start** button. To set up a waveform average you select that analysis exactly as described in the Signal introduction. However when you click on **New** the on-line processing dialog appears:

This dialog is very similar to the off-line processing dialog, it controls which frames to include in the processing and how to update the memory view holding the analysis result. The most common mode is **All filed frames**, with an update every 0 frames (every frame). This dialog disappears once you select either **OK** or **Cancel**, you can recall it with the **Process** command in the **Analysis** menu.



Saving sampling configurations

Once you have generated a sampling configuration and have proved that the sampling works as you want it to, it is usually worth saving the configuration for later use. This can be done by using the **File** menu **Save Configuration As** option and entering a suitable name. You can reload the configuration at any point by using the **File** menu **Load Configuration** option.

If you want to save a sampling configuration that includes the position of data views, memory views and on-line analysis parameters then this is best done while you are sampling or just before pressing **Start** - so that you can see what you are saving. See the main Signal documentation for more on saving configurations. You can also save the configuration after you have finished sampling.

If your sampling configuration contains unwanted duplicate views or memory views, these can be removed by starting sampling, so that you can see the views, deleting unwanted views and rearranging the rest and then saving the new configuration.

Whenever Signal starts, it tries to load an initial configuration from the file `last.sgc`. This file is created automatically by Signal whenever sampling finishes without an error and without being aborted by the user. Thus Signal will normally be set up to sample in the same way as it last did so successfully. This behaviour can be overridden by creating a default configuration file which will be loaded in preference to `last.sgc`. You do this by using the **File** menu **Save default configuration** option, which saves the configuration in the file `default.sgc` (normally in `C:\Signal`).

Advanced sampling

Advanced sampling

This session covers the more complex sampling features available in the Signal software, in particular the use of multiple states to generate different outputs in different sampling sweeps.

Multiple states sampling

When used without multiple states, Signal generates the same output pulses for each sampling sweep. A cyclical variation in pulses can be achieved by using pulse variations but the range of possibilities is strictly limited. Using multiple states sampling you can define different sets of outputs and switch between them during data acquisition. This switching can be done manually, or various forms of automated states sequencing can be used, all sampled frames are tagged with the state number. Signal can also sample a different length of sweep with each state if required.

State zero and idling

When multiple states are not used, all sampling uses the only state available, 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 perhaps a sub-threshold stimulus that allows the health of the preparation to be checked but does not generate any useful data.

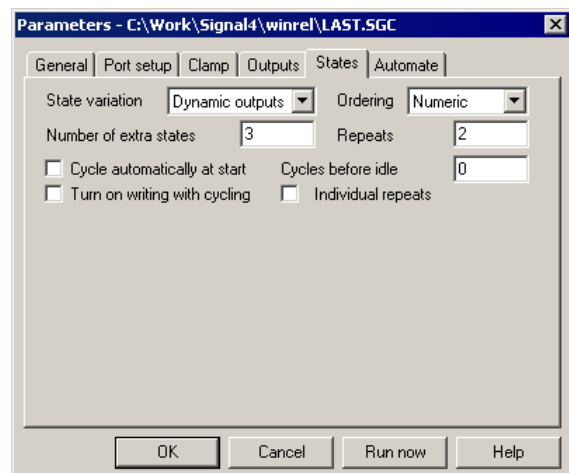
This design choice is most visible in the ‘idling’ behavior of Signal states sequencing. Idling means that Signal will stop sequencing states, switch to state zero and turn off writing to disk. Controls in the states configuration allow you to specify when states sequencing will automatically idle; there 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 and Signal is idling.

Defining multiple states

Multiple states sampling is enabled using a checkbox in the General tab of the sampling configuration. When this checkbox is set an extra tab appears in the sampling configuration for configuring multiple states.

The **State variation** selector at the top left of the page selects the type of multiple states to use. We are only going to look at **Dynamic outputs** in any detail as this is by far the most useful and versatile form of multiple states, the other forms will be briefly touched-on at the end of this chapter.

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 a label for the state) are set up using the **Pulses** configuration dialog – which gains an extra control to select the state with which you are working. The states page defines how many states there are and how they are sequenced – how and when Signal will switch from state to state during sampling.



You can have from 1 to 256 extra states in addition to state zero. As mentioned above, Signal will often only make use of the extra states and reserve state zero as a background or idle state.

States sequencing

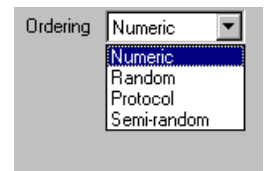
When sampling with multiple states, the states control bar (below) is provided. This can be used to manually select the state in use at any time during sampling or it can start off automatic Signal states sequencing. Manual control of the states is straightforward; click on the button or use the selector to select the next state. Note that the buttons and selector use the label you set for a state if one is available. A button to switch to idling is also available.



Automatic states sequencing

There are three numeric modes provide simple numeric or randomised states sequencing while protocol mode allows more complex sequences of states.

In the three 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 and how and whether the ordering is to be randomised. The **Repeats** item in the dialog sets how many of each state are to be used to make up one cycle of sequencing while **Cycles before idle** sets how many cycles (a cycle being extra states*repeats) of sequencing are wanted. The three types of numeric sequencing are:



Numeric The states are used in numerical order with each state being used the specified number of times. Thus for 3 states and 4 repeats, **Numeric** sequencing gives:

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 uses each state the specified number of times but the order of the states within a cycle is randomised. So we might get:

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 another cycle of sequencing starts the order is re-randomised.

Semi-random This is a slight modification of Random mode where the states are not randomised across an entire cycle but instead randomised within one set of states. Thus 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. As you may have realised, this mode achieves exactly the same as setting the number of repeats to 1 with **Random** mode.

Protocol sequencing is the most complex and flexible form available from Signal, 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.

Protocols are defined by using the protocols dialog which is obtained by pressing 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.

The checkboxes at the top of the protocol details set general options and define what happens at the start of protocol execution. The **Create toolbar button for protocol item** provides a separate button for this protocol in the states control bar, **Run protocol automatically at start** does what it says; sets this protocol to be run when sampling starts.

The table below these checkboxes defines the protocol steps. There are ten steps in a protocol, each one with a **State**, a **Repeats** count and a **Next** step, specifying the state to be used, the number of times to repeat it and the step to go to when this step is done respectively. If you set the next step to zero then this step is the end of the protocol.

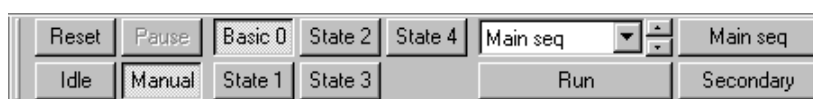
When protocol execution begins it starts with step 1. The state set by the **State** field in step 1 is set and is used the number of times set in the **Repeats** field. Following this the protocol switches to the step that is set by the **Next** field. This process continues until it is ended by encountering a **Next** item of zero. In the example shown above, 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 run forever or it could be set to 3 for a more complex sequence.

The **Repeat count for entire protocol** item below the step table controls the number of times that the protocol repeats before it ends. The items below the repeat count control what happens when protocol execution ends. The **At end** selector selects either **Finish** or a protocol that will be 'chained-to', chaining to a protocol allows more complex sequences than is possible with just ten steps.

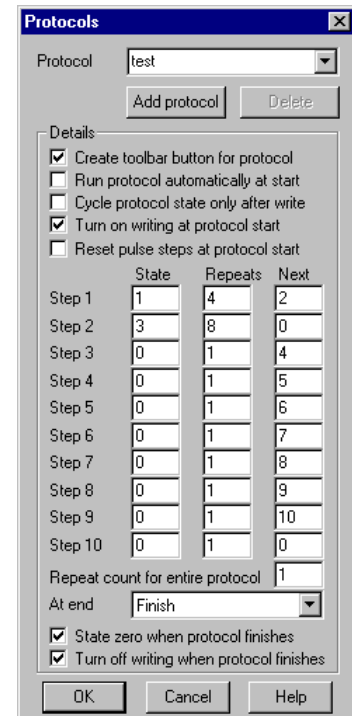
The checkboxes at the bottom of the dialog control what happens when a protocol actually finishes; they are disabled if the protocol chains to another protocol. When the test protocol shown above is executed the following sequence of states would be generated:

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. You get a different states control bar when protocol sequencing is in use:



it provides a protocol sector and Run button to start off protocols plus individual protocol buttons as required.



Other forms of multiple states

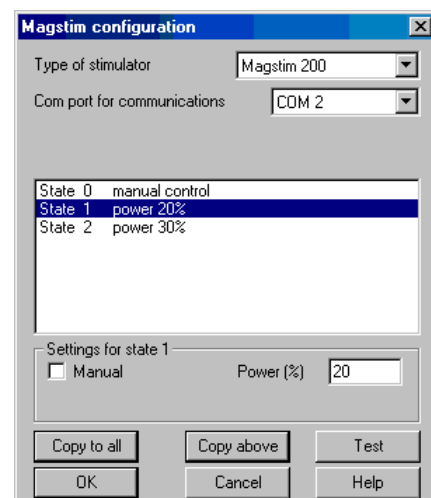
The type of multiple states sampling described above, where a different set of pulse outputs is defined for each state, is called *Dynamic outputs* mode and is by far the most powerful and useful form of multiple states. Two other forms of multiple states sampling are available: *External digital*, where external equipment generates digital outputs (normally corresponding to the stimulus used) that are sampled and used to tag the sampled data with the correct state value, and *Static outputs*, where the state in use sets unchanging DAC and digital outputs – this mode sets up the outputs considerably earlier than can be achieved by using *Dynamic outputs* and is suitable for controlling external equipment where the stimulus to be used is ‘signalled’ in advance of the sweep trigger using the 1401 outputs.

Auxiliary states hardware

The use of auxiliary states hardware allows the multiple states system to achieve still more by automatically configuring external hardware differently for each state as well as adjusting the behaviour of the 1401. This external hardware is controlled by specially written software, which allows more sophisticated and flexible control than that allowed by, for example, *Static outputs* multiple states. The most interesting auxiliary states hardware option currently supported by Signal is the Magstim range of transcranial magnetic stimulators.

The configuration dialog for a Magstim allows us to define a serial line to use for device control and provides a Test button to check serial line communications. The output power level for each state can be defined as well as setting some states to revert to manual control of the output power.

While sampling 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 or high coil temperature error develops. To aid data analysis, Signal saves the Magstim stimulation parameters used in the data section variables of the sampled file.



Text sequencer control of outputs

For most purposes the pulses configuration dialog provides a quick and easy way of defining output pulses that is sufficiently flexible for your needs. If necessary, it is possible to obtain still more complex behaviour by using the text sequence editor instead. This allows you to write a sort of program using the sequencer instruction set that will be executed during the sampling process, allowing a vast range of possibilities. For example, a sequence could wait until 0.1 seconds into the sweep, then monitor an input channel until the voltage goes above a specified threshold (said threshold being updated for each sweep by a script running on the PC) before generating an output pulse. A complete description of the text sequencer is beyond the scope of this manual, but you should be aware that it is available if needed.

Analysis menu

Analysis menu

Introduction

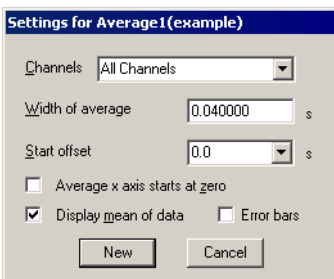
This session covers the built-in analysis functions within the Signal software. Many of these features will be of use generally, however some are specifically aimed towards patch and voltage clampers, in order to add the functionality previously only available with our old DOS software. The main Signal features appropriate to clamp experiments are trend plots, curve fitting and leak subtraction.

Overdrawing frames

More of a display method than an analysis technique, the overdrawing of frames is controlled through the View menu. The frames to be displayed and the way in which they are displayed first needs to be defined, this is done using the Overdraw settings... option in the View menu. This contains a standard set of frame set selectors plus options controlling the colour in which overdrawn frames are drawn, limiting the total number of overdrawn frames (particularly useful for avoiding online delays) and a set of options controlling 3-D overdrawing. 3-D overdrawing shows the overdrawn frames as a 3-D 'stack' with the current frame at the front and other, older, frames behind. Once frame overdrawing is defined it can be turned on and off using the Overdraw frames option.

New Memory View

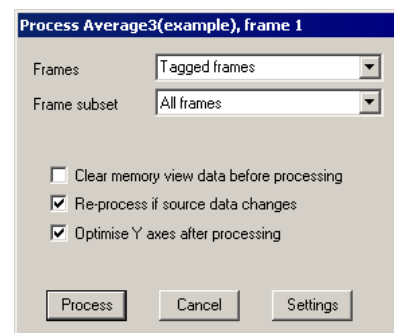
Signal contains a number of built-in analysis mechanisms which create new memory views holding the results of the analysis of data from a source file view.



Analyses are performed using the Analysis...New Memory View option from the Signal menu. Analyses are configured by selecting the appropriate analysis from the list of types available, and then completing an analysis parameters dialog specific to the chosen analysis.

This dialog shows a typical example of the first stage of the analysis setup both on and off-line. You can see that at the top of this dialog you can choose the channels from the source data view to analyse. This is followed by the width of the average to generate and below it is the start offset, within the frame, of the data to be averaged. Checkboxes are used to select for a forced X axis start at zero, display of data as a mean, and generation of error information. This dialog defines the memory view that will be created, once you press the **New** button the new memory view will be constructed.

Once the new memory view has been constructed, you are presented with this dialog, or a similar dialog for on-line processing, which is used to set the source frames to be analysed. You can specify the frames to be processed in a number of ways, such as all tagged frames, untagged frames, frames with a given state or you can enter frame numbers and ranges directly. The **Frame subset** allows further specifications of the exact frames you wish to process. For example: you can specify all tagged frames with state code 3. If the **Clear memory view before**



process checkbox is unchecked as shown then the new result will be added into the old. If the **Optimise Y axis after processing** checkbox is checked as shown then the memory view Y axes will be readjusted after processing to display all of the results.

There are five types of waveform analysis provided by Signal. Each analysis produces a new window containing the result of the analysis. A wide range of additional analyses are possible using a Signal script, but these possibilities will not be discussed here.

Waveform average

This is a very straightforward analysis, it averages waveform channels using the start of the frame as a reference time. You can apply an offset to this reference time, and set a width for the average less than the source frame width to select the data that is actually averaged. The results of the analysis can either be displayed as an average (mean) or the sum of all values averaged can be shown instead. You can also select whether to generate and display error data in the average or not.

Auto average

The Auto-Average analysis is used to produce multiple averages from a data file, each average being generated by a set number of source frames. Apart from the fact that the averaged data forms a multiple-frame memory view, auto-averaging is very similar indeed to the standard waveform averaging. Two parameters set the way that source frames are used to create averages; **Frames per average** sets the number of frames that are averaged into each resulting average, while **Frames between averages** sets where the next average starts. If both numbers are set to 4, for example, then each average takes data from four frames, and the second average starts four frames on from the first so the averages neither overlap nor have gaps between them. If you change **Frames between averages** to 6, then there will be a gap of two unused frames between the averages, if you set it to two then they will overlap by two frames.

Amplitude histogram

This analysis produces a histogram with amplitude on the x-axis and time on the y-axis; it shows the relative frequency of occurrence of different waveform (voltage) levels. Each bin of the histogram will show the amount of time a data trace is within a given amplitude range. Unlike most of the analyses built into Signal, amplitude analysis can only be carried out on a single channel (because the X axis range used depends upon the channel data scaling).

Power spectrum

A power spectrum displays the frequency components of a waveform signal measured in units of root-mean-square power. The analysis is performed using a mathematical operation called a Fast Fourier Transform (FFT) which resolves a block of waveform data points into power components for frequencies running from 0 Hz to half the sampling rate for the waveform data.

The FFT transforms raw data blocks which are 2^n points in size, where n can be set by the user in the range 4 to 12 (16 to 4096 points). The data that is processed by the analysis must contain at least as many data points as the FFT block size, only a single block of data from each frame will be processed. A better frequency resolution will be achieved by supplying a bigger block size, but at the cost of precision in locating the frequency components in the original waveform (because larger block sizes require larger numbers of data points in the raw waveform).

A discontinuity occurs at the block boundary and this can introduce unwanted frequencies into the result. To overcome this a windowing function can be applied to attenuate the raw data towards the block boundaries.

Leak Subtraction

Leak subtraction is a specialised analysis used by Voltage and Patch clamp researchers. The details will be covered in the Patch and Voltage Clamp chapter.

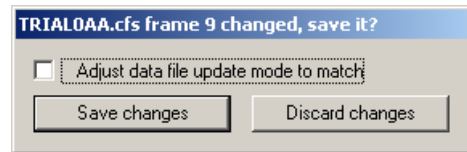
Direct data modification

Using the Analysis menu **Modify Channels** function, or the equivalent keyboard shortcuts, the data in the current frame can be zeroed, rectified, scaled, smoothed, integrated and differentiated, amongst other possible manipulations. In all cases, the modification operates upon all selected channels, or all visible channels if none are selected.

Channel arithmetic is also available at the bottom of the menu, a single channel can be specified and its data added to or subtracted from (or multiply and divide) all selected or visible channels.

Changed data update

Signal holds a copy of the data for the current frame of a data file in memory for quick access and will allow you to change this copy of the data freely, either manually in various ways or by means of a script. When Signal is about to clear out this local copy of the data, for example when switching to another frame, it has to decide what to do with any changed data. What happens is controlled by the File menu **Data update mode** item for this file – you can choose to discard all changes, write all changes back to disk, or to be prompted each time for a decision on what to do.



The preferences dialog contains a selector for the default data update mode, which will be used with each file initially. Though it may be changed on a file-by-file basis with the file menu this setting is not stored with the file and will be reset to the value in the preferences when the file is re-opened.

Tag/untag frame

Each frame in a Signal data view can be either tagged or untagged, the intention behind tagging was to provide a quick way of flagging a group of frames for any required purpose, for example to indicate frames whose data contains stimulus artefacts. Whenever you have to select a set of frames in Signal, two of the possible choices will be all tagged frames and all untagged frames. You can toggle the tagging state of a frame using this menu item or **Ctrl+T**.

Append/delete frame

It is possible to append frames to an existing data view, these can either be a blank frame with zeroed data, or a frame containing a copy of the data in the current frame. You can also append a frame to most types of memory view created by processing, for example power spectrum analysis, and then process a different selection of source frames into it. Appended frames will be added to the data file and made permanent when the view is closed, before that point they can be deleted using the **Delete frame** menu option. It is not possible to delete frames that are in the CFS data file on disk.

Frame buffer manipulation

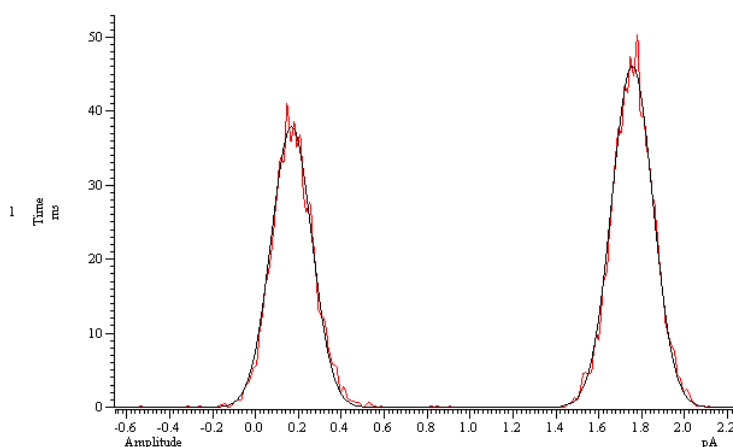
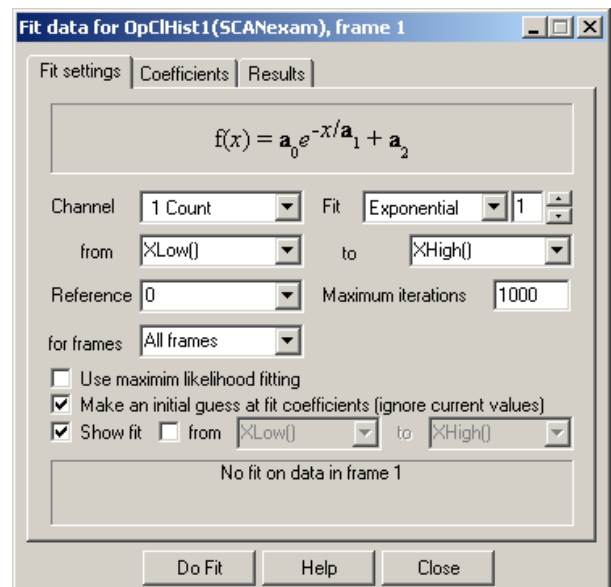
The frame buffer is a separate frame of data that is held 'behind' the current frame in a data view. The display can be toggled between the current view and the buffer by using the View menu or Ctrl-B shortcut. If the buffer is being displayed then it and not the view data will be affected by the Modify Channels functions.

Using the analysis menu, the frame buffer can be used in a number of ways, for example to copy reference data to the buffer and then subtract it from a number of other frames or to generate an interactive average. These operations can be carried out interactively or the analysis menu Multiple frames dialog can be used to carry out buffer operations using a number of frames. The multiple frames dialog can also apply the modify channels functions.

You can move frames of data around by using the Windows clipboard. Edit:Copy will put data from the visible channels onto the clipboard, while Edit:Paste will copy data from the clipboard into the current view. This mechanism can be used to move data from one file to another.

Curve fitting

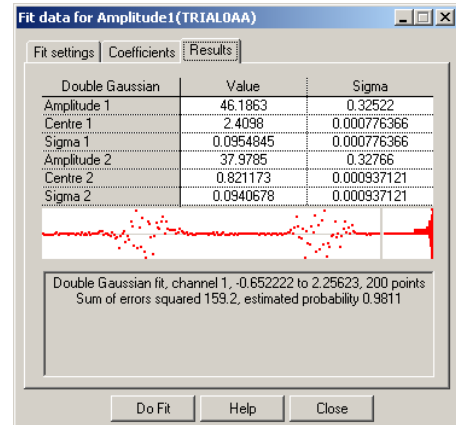
Curve fitting in Signal allows you to select an equation which is fitted to view data. To carry out the actual fitting process which, in essence, consists of calculating the values of coefficients in the equations to give the closest match to the data and optionally draw a curve on top of the raw data to show the equation curve. The equations that are available in Signal version 5 curve fitting are single- and double-exponentials, single- and double-gaussians, sines, sigmoids and polynomials up to order 5. Fitting is available in file, memory and XY views. To carry out curve fitting you use the Analysis:Fit data... menu option (also available via a mouse right-click), which provides the curve fitting dialog.



This dialog allows you to select the equation to be used and its order, and the channel and time range containing the data to be used. The control to the right of the equation selector sets the number of terms (order) of the selected equation. For example set this to 2 and the fit function to Exponential to fit a double exponential. The Reference selector allows you to set a reference (zero) time for fits (particularly useful for exponentials) as distinct from the time at which the fitting starts.

The other items control the frames to be fitted, the number of iterations to be executed before giving up, the display of the fitted curve and automatic estimation of the initial coefficient values.

The second tab in the dialog allows you to set the initial values for the fit and the limits to these values, to fix coefficient values so they cannot be altered by the fit and to initialise the coefficients using an estimate based on the current data. Once fitting has been carried out, this tab will show the actual fit coefficients calculated, which can be directly edited.



The third tab (see right) displays numerical information about the fit result. A graphical representation of the errors in the fit together with the distribution of the errors is also displayed. The numerical fit results, including an estimate of fit probability and the residual Chi-squared error, can be copied to the clipboard (use mouse right-click).

Advanced analysis

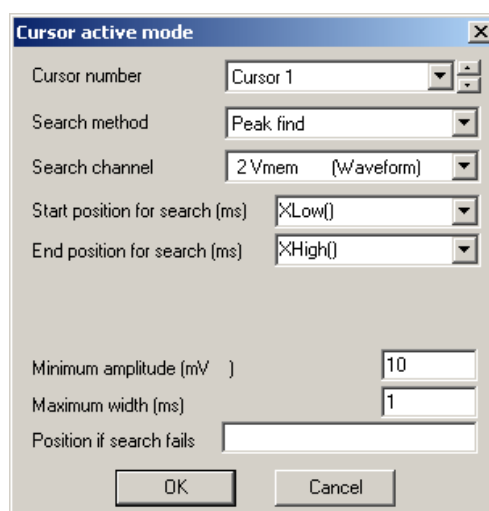
Advanced analysis

This chapter will cover the more advanced and specialised analysis tools available from Signal, including active cursors and trend plots, curve fitting and digital filtering.

Active Cursors

Normally, cursors in Signal are static and simply stay at the X axis position where they are put. Active cursors behave differently; moving automatically to the position of some feature in the data each time the current frame or the data in the frame, changes. There are a whole host of features that can be searched for by the cursor; searches for maximum and minimum values, for peaks and troughs and for threshold crossings, among many others. You should look at the main Signal documentation for details on the individual active cursor modes and how they work.

The cursor active mode is set using the Active Cursors dialog, which is available either via the Cursor menu or by right-clicking on a cursor. Using the dialog, you specify an active mode plus any parameters (for example a threshold level) needed, the channel to search for the feature and a time range for the search. A crucial aspect of setting up active cursors is that you can 'cascade' the searches by using the position of another, lower numbered, cursor to set the search range. For example, you could set cursor 1 to find the maximum value in the frame, then set cursor 2 to search for the first trough after cursor 1, then set cursor 3 to search between cursor 1 and 2 for the steepest falling slope.



Measurements and Trend Plots

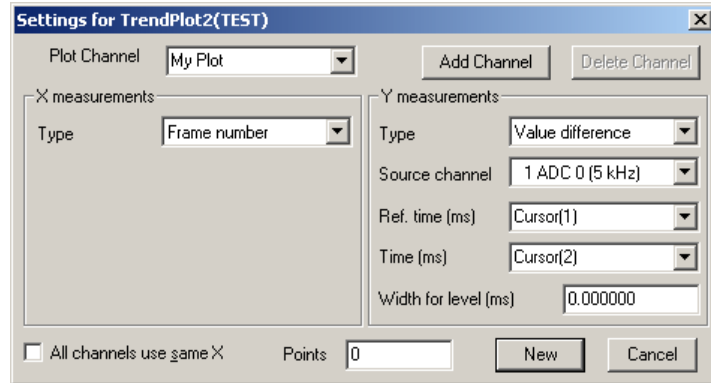
Signal can take measurements automatically from a set of data file frames and plot the result in an XY view to generate a trend plot. These measurements can be very straightforward (the mean value in a channel) or they can use active cursors and the range of cursor region measurements available to generate very complex analyses. Trend plots have been designed so that they are generated in much the same way that a memory view would be in that they have a settings dialog and a process dialog. Trend plot generation is of particular interest to researchers using whole-cell clamping but it is applicable to many other areas of work.

In the analysis menu go to **New XY view**, then **Trend plot...** This rather daunting dialog is actually not as bad as it looks as the X measurements and Y measurements sections are identical in every way to allow you to plot any measurement against any other.

Trend plots can contain several channels, each generated by pairs of measurements for X and Y. This means it is possible to plot more than just one set of points at a time into the same XY view; with a single pass through the data file it is possible to take several different readings and plot them together on the same graph. Each channel is plotted using a different symbol to make it easy to compare one set of readings against another.

Open the `test.cfs` file and make sure you have at least two cursors in the data view. After bringing up the trend plot dialog, change the name of, “Channel 1” to “My plot”. This you do simply by typing into the Plot Channel field of the dialog. Next you can change the X measurement type to “Frame number” and the Y measurement type to “Value difference”. As you can see there are fields in the dialog which are hidden whenever they are not needed.

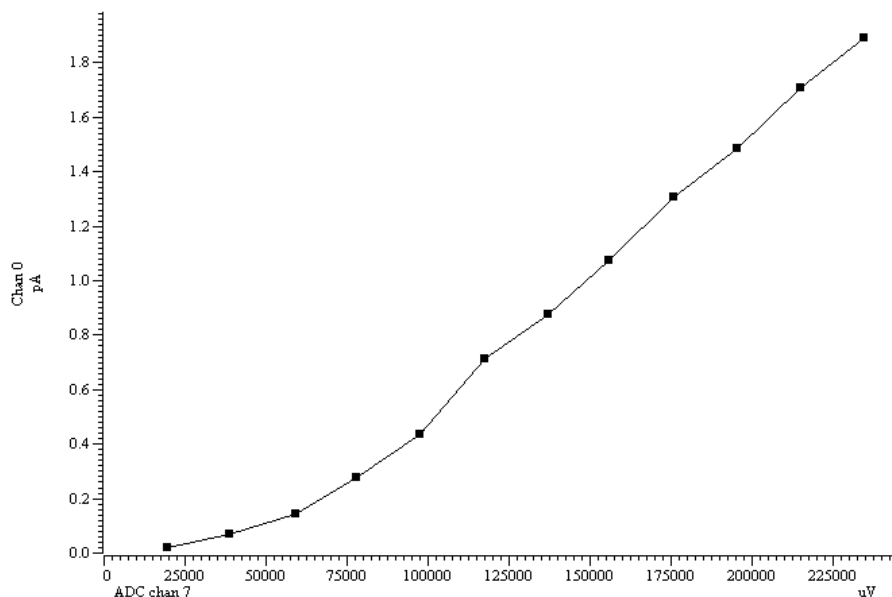
We want to take readings from channel 1 so select “1 Chan 0” as the source channel. For the reference time select “Cursor(1)”. This is the time at which a value will be read and subtracted from the value measured at the time given in the next field. Width is as for active cursors; it determines over what width a reading is averaged before being used.



Leave Points in plot set to 0. If it were set to something like 10, then the 11th point would displace the 1st. This can be useful on-line for keeping a running check on some aspect of the data.

At this point you should place Cursor 1 before the stimulus pulse and Cursor 2 close to the first main peak in the data. Make sure Cursor 1 has a static mode and Cursor 2 has a peak seek mode with a hysteresis of about 1. This is where the power of the active cursors comes into play. As Signal steps through each frame, Cursor 2 will jump to the peak in the data and the measurement will be taken from there.

Pressing **New** produces a process dialog almost identical to that you might expect for forming an average. As with other forms of processing this can be done on-line. Indeed it is possible to plot a graph from auto-averaged data, all of this analysis being done automatically, on-line!



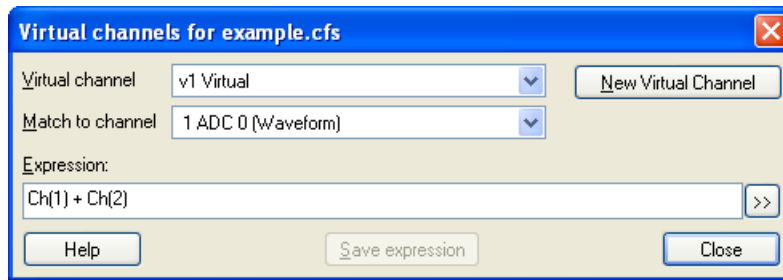
Now open the Actions.cfs file. This contains an incrementing stimulus in channel 1 and action potential data in channel 2. In the Analysis menu go to New XY view, then Measurements.... This method of generating an XY plot allows multiple measurements to be taken from each frame. The mechanism for doing this revolves around Cursor 0. The upper part of the Measurements dialog is used to set the active mode for this. You can now set Cursor 0 to search for peaks above 10 mV and up to 1 ms wide and more than 1 ms apart. You should also set the minimum step size to 1 ms. We will not bother with the Skip if field. This allows an expression to be entered which if false

will cause that iteration of Cursor 0 to be thrown away and not contribute to the XY plot. We will also not require the user to check each iteration to make sure it is valid. The remainder of the dialog has much the same fields as for trend plots described above. Set the X Measurements to Time at point with the time being Cursor(0). Set the Y Measurements to Value difference on Channel 1 between 0 and 700 ms, averaged over 100 ms. We want individual measurements within each frame and not the Average measurements within frame so leave that checkbox un-checked. Clicking New then Process produces a n XY plot of the stimulus heights against all the action potential times. The initial draw mode for the plot is unlikely to be what you want so you would then go on to change this. Alternatively you could at the stage copy the plot as text for use elsewhere.

Virtual channels

Virtual channels hold waveform data derived by a user-supplied expression from waveform and marker channels and built-in function generators. No data is stored; the channels are calculated each time you use them. 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 the source waveforms and by smoothing event rates. The script language equivalent of this command is `VirtualChan()`. If you use a virtual channel as a source channel for a new memory view (eg using Waveform Average) then the channel in the memory view will behave just as if it had been created from a real channel and will be written to disk if the view is saved.

There are menu commands to create a new virtual channel or edit the settings of existing virtual channels. Both commands open the Virtual channel dialog:



The Virtual channel field is used to select a channel when you have more than one virtual channel. Match to channel is used to select an existing waveform-based channel (but not a virtual channel) from which to copy the sample interval and data alignment. Alternatively, you can select Use manual settings and type in the interval and alignment yourself. The Expression field holds an expression that defines the virtual channel. Expressions are composed of scalars, vectors and operators. A scalar is a number, such as 4.6 or $\text{sqrt}(2)$. A vector is displayable channel data. You can use the four standard arithmetic operators plus (+), minus (-), multiply (*) and divide (/) together with numbers, parentheses and some mathematic and channel functions. The result of combining a vector and scalar with an operator is a vector, for example, the expression $\text{Ch}(1)+1$ is a vector, being the data points of channel 1 with 1.0 added to each of them.

Build expression

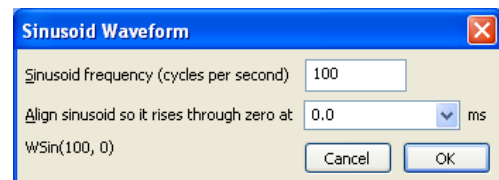
There is no need to remember all the expression functions; just click the >> button and choose from a list of possible items to add. You can choose from:

Waveform from channel

Select this 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 replaces the selection in the Expression field.

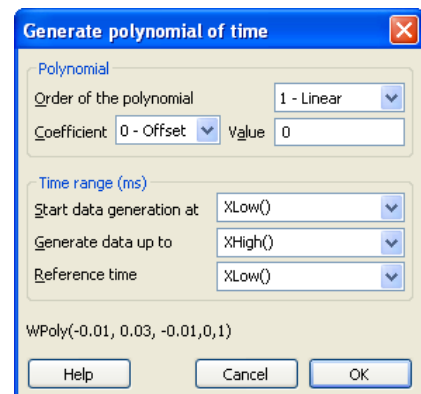
Generate waveform

Select this to generate the commands that create a channel based on a sine, square or triangle wave or on a waveform envelope or based on linear time or on a polynomial of time. You build the commands using a dialog. All dialogs display the equivalent command in their lower left corner. The result replaces the selection in the Expression field.



The $\text{WPoly}()$ command is more complex, and can be used to generate polynomials in time relative to a reference time. Such curves can be used to create complex envelopes, or to subtract out a curve generated by the interactive curve fitting routines.

The $\text{WT}()$ 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.



Channel process functions

The commands in this section process the selection in the **Expression** field in some way. For example, if the **Expression** field holds $\text{Ch}(1) + \text{Ch}(2)$ and you want to rectify the channel 1 data before adding it, select $\text{Ch}(1)$, click the **>>** button and select **Channel process functions**, then select **Rectify**.

Mathematical functions

The commands in this section apply a nominated mathematical function to the selection in the **Expression** field, which will usually be vector expressions. For example, if the **Expression** field holds $\text{Ch}(1)$ and you want to square the channel 1 data before using it, select $\text{Ch}(1)$, click the **>>** button, select **Mathematical functions**, then select **Square the selection**.

Mathematical functions: Poly()

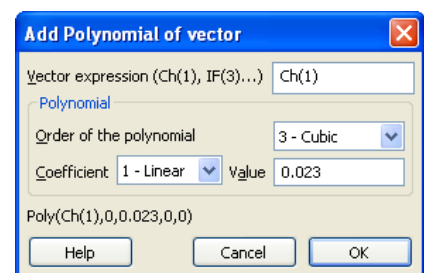
Select this to generate a polynomial. The *Vector expression* field is set to whatever was selected when this dialog was opened, or is set to $\text{Ch}(1)$ if nothing is selected. This field is not tested for validity. Each element x of the vector is replaced by a polynomial in x . You can set the order of the polynomial (order means the highest power of x used) in the range 1 to 5. The command is:

$\text{Poly}(x, a_0, a_1, a_2, a_3, a_4, a_5)$

where x is the vector expression (you can also use a scalar, but this is not very useful) and the a_0 to a_5 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 a_5 , for a cubic omit a_5 and a_4 . To set coefficient values in the dialog, use the *Coefficient* field to select the required coefficient and then set its value.



Frame, tag and state functions

These are simple functions giving access to the frame details of the data. You can for example, use the $\text{Tag}()$ function to multiply part of your expression to set that part to zero for un-tagged frames. Use $(1 - \text{Tag}())$ if you wanted to zero anything when tagged.

Mathematical operators

These commands replace the selection with $+$, $-$, $*$ and $/$ to remind you that you can use these operators to add, subtract, multiply and divide vectors and scalars.

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.

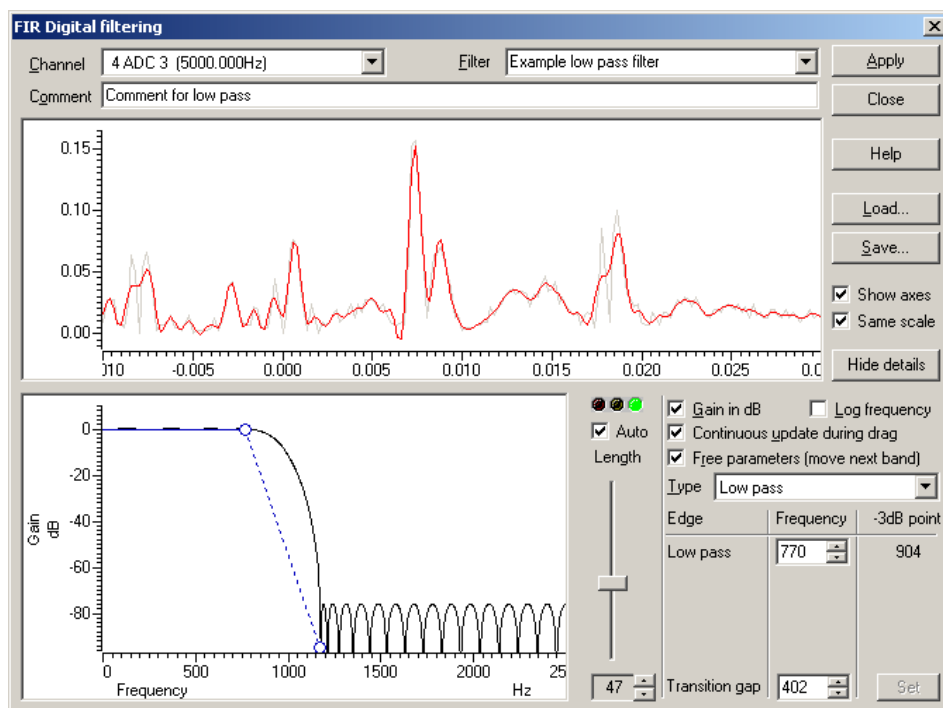
Digital filtering

Often data in a file will contain frequency components which are not wanted. For example, this may be high frequency noise, baseline drift or mains interference. In many cases such as these it may be possible to remove these components of the data using digital filtering.

FIR filtering

Load "example.cfs" from the examples disk then go to the Analysis menu and select "Digital filters" then "FIR filters...". The FIR digital filtering dialog appears with the initial filter set to "Example all stop filter". The lower part of the dialog shows two data traces. The grey trace shows the unfiltered data and the red shows the filtered. You can experiment with selecting different channels and different filters. By default the filtered data is drawn at the same scale as the unfiltered data. Un-check the "Same scale" box to see the filtered data optimised to the display range. Axes may be turned off to allow more room for the data display by un-checking "Show axes".

As it stands this is obviously a very limited facility. To make full use of the digital filtering you will need to click on "Show details". This reveals the rest of the dialog containing the filter characteristics. The drop down list that appears here contains the different filter types. The list at the top of the dialog being names for these filters and can be changed to anything you like.



The filter characteristics may be edited either graphically or by entering values. To change the filter graphically, simply click and drag on the part of the graph corresponding to the filter characteristic you wish to change. The traffic lights next to the graphical display shows the quality of the filter. A red light indicates that the filter characteristics have too much wobble to be useable. Amber means it is just useable and green shows a good filter. The check box below this allows Signal to automatically adjust the number of coefficients in the filter. More coefficients will allow the filter to be more accurately defined but will be slower as well as increased problems with edge effects at the start and end of each sweep. Adjustments to the filter are normally transmitted to the previewed data at the top of the dialog. If this causes Signal to become sluggish then you may uncheck the "Continuous update" box and the display will only update when the mouse is released.

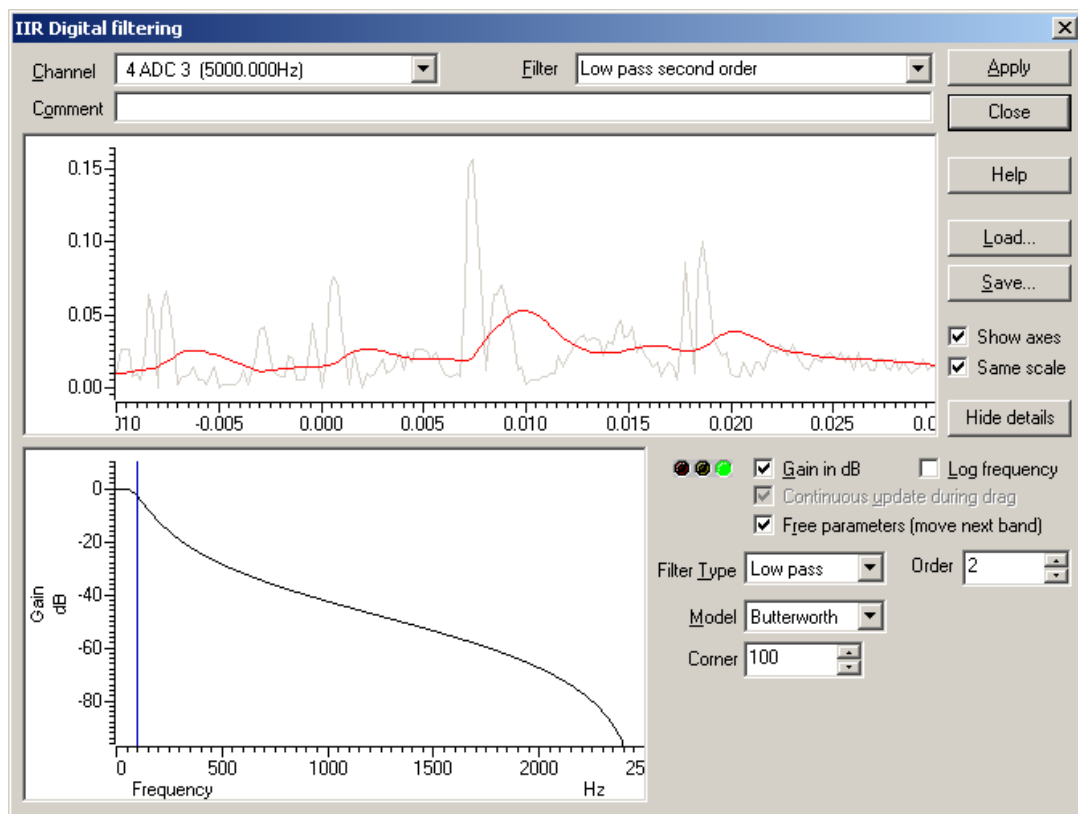
The filter characteristics are normally shown on a logarithmic scale. To see them with a linear scale uncheck the "Gain in dB" box. The x-axis can also be drawn as logarithmic by checking the "Log frequency" box. If a feature is dragged to a point where other parameters are a limitation then further dragging will be ignored unless the "Free parameters" box is checked in which case the other features of the filter will then move to accommodate the drag.

Pressing the "Apply" button causes Signal to prompt for a list of channels and frames to filter. As each frame has its data altered Signal will prompt to see if you wish to keep the changes unless you have set your preferences for it not to do this.

IIR filtering

IIR filters more closely mimic analogue filters that you may be used to. They have the advantage that they can have much sharper cut-offs than FIR filters and they are also causal. That is the resulting waveform is not influenced by data which has not yet happened. The main disadvantage is that there is a time delay in the output which is frequency dependant. This means that peaks in the data will be shifted to later times and the output may significantly change shape.

Go to the Analysis menu and select "Digital filters" then "IIR filters..." The FIR digital filtering dialog appears. This is very similar to the FIR filter dialog



The "Model" defines the type of analogue filter to emulate for which a corner frequency is simply the -3dB frequency. The "Resonator" will require a "Centre" frequency and a "Q" value which corresponds to the sharpness of the filter's cut-off.

Patch and Voltage Clamp

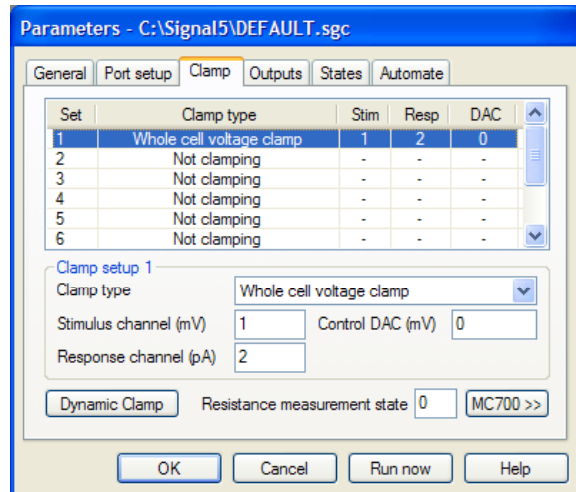
Patch and Voltage Clamp

This chapter is for anyone involved in whole cell or patch clamp data acquisition and analysis. It covers on-line clamping options as well as leak subtraction, idealised trace formation and open/closed time histograms.

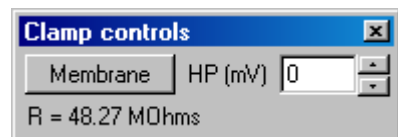
Online clamp support features

The online clamp support consists of a separate page within the sampling configuration dialog where the clamp setup can be configured, plus 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. Up to eight clamp setups can be defined along with a state (set of pulse outputs) to be used for membrane analysis.

Clamp setups define the data file channels that hold the current and voltage data (so that analysis can find the correct data) and checks of the units for these channels (so that current and voltage values can be scaled correctly to amps and volts). In addition the setup defines the DAC used to control the membrane current or voltage (so that the holding potential and pulses can be manipulated). The **Experiment type** item can be set to **Not clamping** to disable use of that clamp setup or to **Whole-cell** or **Single channel**, each either voltage clamp or current clamp. If all the setups are set to **Not clamping**, then all of the online clamping support is disabled so you can sample without any on-line clamping analysis. 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 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. If Signal detects that you are using illegal units then this will be shown in red indicating there is a problem.



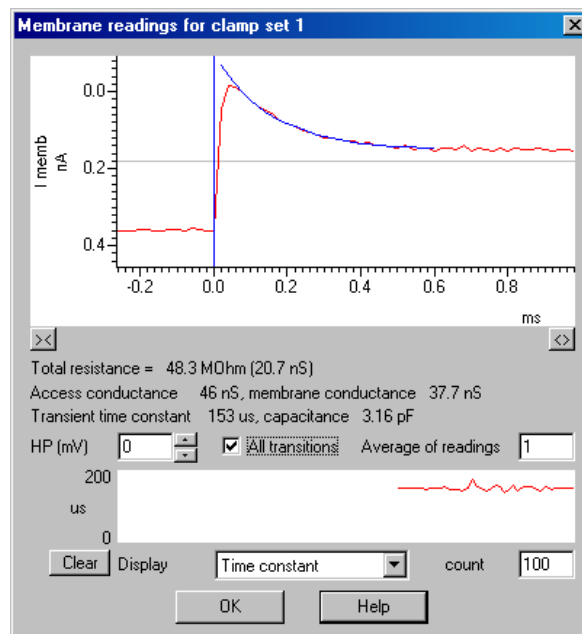
When a sampling configuration with at least one enabled clamping setup is used, the clamping toolbar is created in addition to the standard Signal sampling toolbars. The clamping toolbar contains separate controls and information for each clamping setup. It contains three items; controls for the holding potential, a text display area and a button marked **Membrane** used to provide the membrane analysis dialog. Whenever the specified state is sampled the resistance measurement will be updated.



The membrane analysis 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 measures the total resistance (and total conductance), access and membrane conductance, the time constant for the decay of the capacitance transient and the membrane capacitance.

The two buttons labelled \gg and \ll (below the waveform display area) can be used to increase and decrease the time range shown. The All transitions checkbox, 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. You can average the analysis results that are displayed over a number of sweeps by using the Average of readings control. The Display selector selects the measurement to be displayed as a graph, 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 checkbox operates).



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.

Dynamic clamp

Dynamic clamp is a tool which effectively allows conductance to be added or subtracted from a cell during a current clamp experiment. The conductance can either be a fixed waveform or calculated continuously as the solution to a differential equation. Each one of these waveforms or calculations is stored in a "model" which is then transferred to the 1401 for execution during sampling. As you might imagine, dynamic clamp is computationally intensive and for this reason a Power 1401 Mk II is needed to make it work. Access to the model setup is by the Dynamic Clamp button in the clamp setup page of the sampling configuration dialog. Clicking this button brings up the list of dynamic clamp models with buttons to add more models or edit or delete existing models. There is also a check box which turns the dynamic clamp off between sweeps when data is not being sampled. This effectively means that no conductance is added to or subtracted from the cell between sweeps.

The list of types of dynamic clamp model that can be added to fall into four main groups: Hodgkin-Huxley models; synaptic conductance; leak conductance and noise. Clicking on the Add \gg button drops down a list of available model types. Before any model can be used however, you will first of

all need to make sure that the membrane potential is being sampled (in units of mV) and that a control DAC has been set up to inject the current needed to simulate the conductance. The control DAC needs to be calibrated in either pA or nA in order for Signal to convert conductances calculated in nS into current to be injected. In addition to this you will probably want to sample a copy of the injected current in order to see what is happening. Some models require more than one membrane potential to be sampled (i.e. both sides of a synapse) and an addition control DAC as well.

If we click on the Add >> button and select the Hodgkin-Huxley (Alpha/Beta) option the model setup dialog will appear. The Model Name is just for you to put some sort of reminder as to what the model is for. The name will appear in the model list when this model is accepted. The Input Channel is the channel number (as shown in the file view) of the data channel on which the membrane potential is sampled. It is not the ADC port number. The Control DAC is the DAC which supplies the command voltage to the amplifier for converting to a current to inject. G_{max} is the maximum conductance that can be injected. If this value is negative then conductance will be subtracted instead of added. Introducing a pharmacological block of a particular channel type then reinstating the behaviour with dynamic clamp would need a positive G_{max} whereas removing a channel type without using a pharmacological block would need a negative G_{max} . E_r is the reversal potential. If you are sampling with multiple states, there will be a separate copy of the model for each state. The state being edited will be shown in the drop-down at the bottom of the dialog. You can copy the settings from one state to other states using the Copy To... button. The details of the Activation and Inactivation parameters can be obtained by pressing F1. This will also show you details of the equations used and the means by which Signal solves them.

Hodgkin-Huxley (Alpha - Beta)

Model Name: Na+

Input Channel (mV): 1 G_{max} : 50 nS
Control DAC (pA): 1 E_r : -10 mV

Activation **Inactivation**

Include Include

p : 1 q : 1

$F_\alpha(x)$: $x/(\exp(x)-1)$ $F_\alpha(x)$: $x/(\exp(x)-1)$

k_α : 0.32 ms^{-1} k_α : 0.128 ms^{-1}

V_α : -52 mV V_α : -48 mV

S_α : 4 mV S_α : 18 mV

$F_\beta(x)$: $x/(\exp(x)-1)$ $F_\beta(x)$: $x/(\exp(x)-1)$

k_β : -0.28 ms^{-1} k_β : 4 ms^{-1}

V_β : -25 mV V_β : -25 mV

S_β : -5 mV S_β : 5 mV

Disable model in selected state

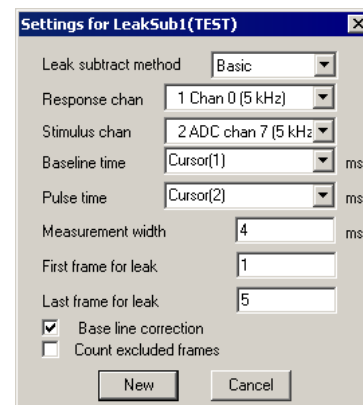
Basic 0 Copy To...

OK Apply Cancel Help

During sampling a Models button will appear on the Clamp bar. This will take you to the list of models. On-line it is not possible to add or delete models. You can however, disable or enable them and even change most of the parameters within each model. Parameters such as the Control DAC cannot be changed on-line.

Leak subtraction

Leak subtraction is a specialised analysis primarily used by voltage clamp researchers, the basic technique is to use a small stimulus, one that does not cause the cell membrane ion channels to turn on and measure the current flow through the membrane impedance (made up from resistive and capacitive components). This 'leak' measurement is then scaled to give the expected non-ionic conductance during a larger pulse and subtracted from the recorded traces to leave only the ion-channel effects. In order to leak subtract with Signal you have to record both the stimulus and responses of the cell. You also have to be able to assume that the leak currents are ohmic – that they scale linearly with respect to the stimulus magnitude.



The first item in the settings dialog is the subtraction method. All the methods use the same basic principal of forming an average leak from one set of frames and subtracting them from the other after scaling the leak according to the size of the stimulus. The **Basic** method involves simply defining a range of frames over which the leak will be calculated. The **P/N** method requires values for p and n, where n is the number of frames to use for forming the leak and p is the number of frames to subtract it from before the cycle repeats. The final method is **States**. Here a frame state code is specified and any frames having this state are used to form the leak which is then subtracted from all the others.

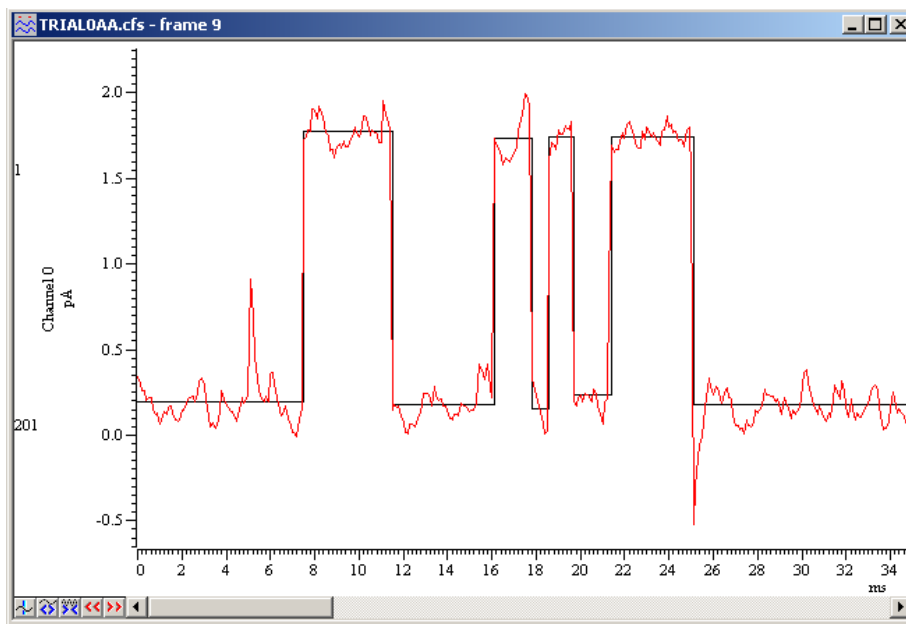
Response chan and Stimulus chan are self-explanatory, I hope. Note that the only channel that will be changed is the response channel – all other channels are copied to the processed view unchanged. **Baseline time** sets a time when the stimulus pulse is not happening; **Pulse time** is a time when it is, while the width of the level measurement made is set by **Measurement width**. If you check the **Base line correction** checkbox, Signal will subtract any DC offset from the resulting response trace so that its value at the baseline time is 0.

As an example, load test.cfs from the examples disk. Add two cursors and place cursor 1 at about 5 ms and cursor 2 at about 25 ms. Bring up the Leak Subtraction dialog. Select “Basic” as the method with “1 Chan 0” and “2 ADC chan 7” as the response and stimulus channels respectively. The baseline time is “Cursor(1)” and the pulse time is “Cursor(2)”. A measurement width of 4 ms is about right for this file and the leak can be formed from frames 1 to 5. After pressing “New” process all frames to produce a memory view with 7 new frames formed from the last 7 frames of test.cfs.

Leak subtraction can be done on-line, though when this is the case you may find it best to use the P/N method.

Idealised trace analysis

In its simplest form patch clamp data will consist of a data trace measuring current through a patch of cell membrane containing a single ion channel. In principle, an idealised trace is a representation of this data with each period in a particular state (either open or closed) shown as a horizontal line of fixed amplitude and the transitions between states shown as vertical lines representing an instantaneous transition between the two states. In Signal the idealised trace may be drawn over the original data trace for comparison, or they can be separated.

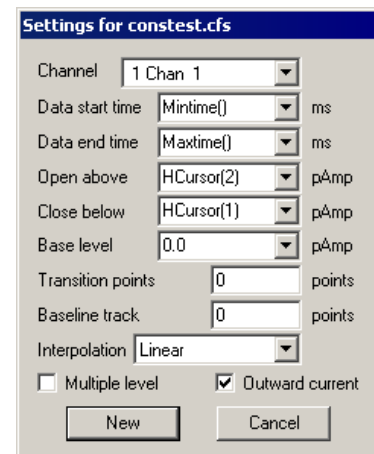


Idealised trace generation

In the Analysis menu select the Open/Closed times sub-menu and then select New idealised trace (Threshold)... This is a simpler method of generating an idealised trace than the SCAN method.

A standard processing settings dialog just like those used to generate memory views or trend plots. The difference here is that instead of generating a view, idealised trace generation produces a virtual channel. Virtual channels appear to the user just like any other channel in a cfs file, but they are not stored in the file itself, they are stored instead in the .sgr file associated with the cfs file, or even generated automatically 'on the fly'.

The Channel field of the dialog selects the data file channel containing the patch current data to be analysed to create the idealised trace.



Data start time and Data end time are the times at which you wish the idealised trace analysis to start and end. The first part (event) of the trace will be flagged as a first latency. The last event will be flagged as truncated.

With Outward current selected an opening in the channel is taken to be represented in an upward direction in the file view. Open above and Close below are thresholds which the data must cross to change from one state to another. Having two thresholds like this provides a way of preventing

noise from generating false triggers. Transition points can be set to make the threshold crossing detection jump on by a number of points to avoid multiple transitions being detected during the transition period.

With Interpolation set to **None** any threshold crossing is assumed to take place at the time of the data point found to be over the threshold. **Linear** interpolation calculates the time of the threshold crossing based on the assumption that the waveform signal changed linearly over time between sample points.

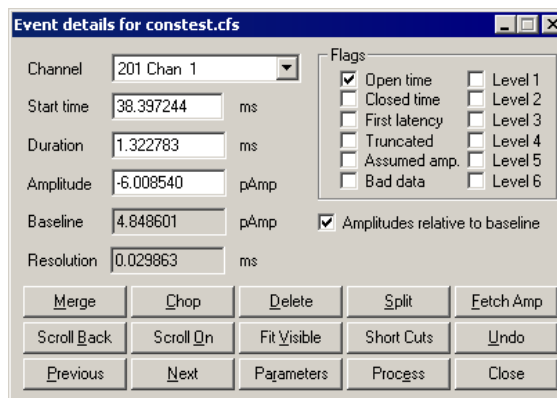
Base level becomes important when dealing with multiple level data. This is usually caused by the patch having multiple channels in it. It is assumed that when n channels open then the current will be n times further from the base level. The thresholds for transitions between levels are based on this assumption. You would normally set the base line to be at the level of current when the channel is closed. Since this level may drift during the experiment it is possible for Signal to record data values in the closed state and maintain the baseline as a running average of these values. The number of points to do this over is also set in the dialog.

On hitting **New** the idealised trace channel is created and the standard process dialog appears allowing you to choose which frames you wish to process. It is worth noting that this can also be done on-line as can the subsequent histogram formation.

Editing an idealised trace

You can edit the idealised trace interactively, dragging horizontal or vertical lines using the mouse cursor. You can also select individual events (horizontal sections) by clicking on them with the mouse.

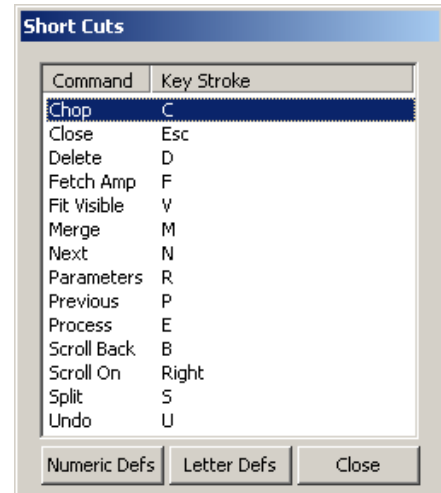
Using the **Analysis** menu select **Open/Closed times** then **View and modify event details** option (also available via a mouse right-click). This opens a dialog showing the event information and allowing you to change the event values directly. The dialog contents will change as you select different events. The dialog **Previous** and **Next** buttons allow you to step through the events. **Scroll On** and **Scroll Back** will do the same thing but will do so using a smoothly scrolling display. Repeatedly hitting the scroll buttons will double the scroll speed each time. You can stop a scroll by clicking on the data area or by clicking on another button in the dialog.



Merge will combine 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** 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. The **Split** button is used to split an event into three, to provide a new event in the centre of the old one. In an analogous manner, the **Delete** button deletes the current event; amalgamating the events on either side.

Fetch Amp will scan backwards through the trace to find an event of the same type and adopt its amplitude from there. **Fit Visible**, **Parameters** and **Process** are used as part of the SCAN method of idealised trace formation so are described later in the chapter as is the **Resolution** field. All trace editing can be undone by using the edit menu or by clicking on the **Undo** button.

Since editing an idealised trace can be very labour intensive, we have added the ability to customise keyboard shortcuts to make the process as ergonomic as possible. Click on **Short Cuts** and another dialog opens. This shows a list of the current shortcuts in use for the trace edit dialog. Click on an item in the list to select it. You can then press any key or combination of keys to set a new shortcut. A few restricted combinations are not allowed e.g. **CTRL+ALT+Del**. The standard defaults can be restored using **Letter Defs** or a set using only the numeric keypad can be set using **Numeric Defs**. Remember to use **Num Lock** if you chose this option. Once you are happy with the shortcuts either close the dialog or click back on the data area to start using them. The short cut dialog can be left up to remind you which shortcuts you have chosen.



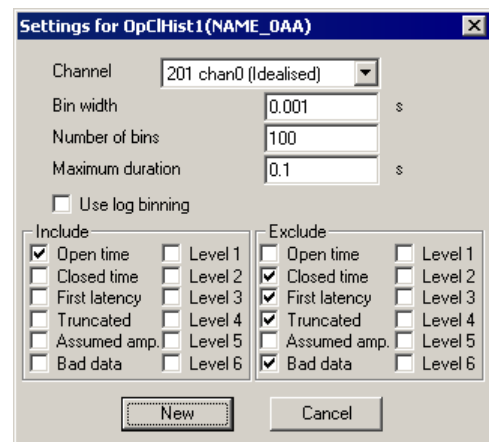
Open/Closed time histogram

Once an idealised trace has been formed it becomes possible to build histograms showing the duration of events of a particular type. Go to the **Analysis** menu again and in the **Open/Closed times** sub-menu select **Open/Closed time histogram...**

This is again a standard settings dialog just like all those generated from the **New memory view** sub-menu. The **New** button generates a new memory view with an X-axis from zero to **Maximum duration** and the number of events on the Y axis.

The **Channel** selected here is the virtual channel containing the idealised trace. It is not the original data trace as sampled by the 1401.

Bin width is simply the x-increment for each bin of the histogram. **Maximum** is the longest duration of an event to be included in the histogram. **Use log binning** is used to generate histograms with a geometric progression of bin sizes, if it is checked then the **Bin width** item becomes **Minimum duration** and the number of bins defines the logarithmic progression.



The **Include** and **Exclude** sets work in exactly the same way as they do in the **Patch and Voltage Clamp Software**. Each event in the idealised trace has a number of flags associated with it. In order to be included in the histogram an event must have at least one flag set which is in the **Include** set and none of the flags set which are in the **Exclude** set. In the above example we are wanting to build a histogram of open times but exclude those open times which are flagged as

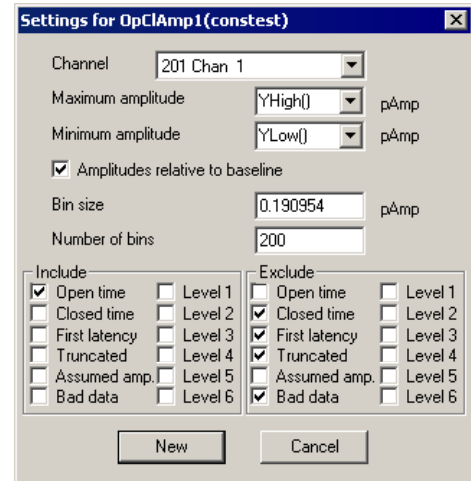
Closed time, First latency, Truncated or Bad data. Checking the Closed time checkbox in the exclude set is redundant here, but it makes no difference.

Open/Closed amplitude histogram

Amplitude histogram formation works in almost exactly the same way as do time histograms. The obvious difference is that we are looking at the level (Y axis value) of an event, rather than the duration. Also, the limits of the histogram are defined in terms of y-values on the idealised trace which translate to x-values of the histogram.

Amplitudes can be measured either as an absolute value using the channel's calibration settings or relative to the baseline.

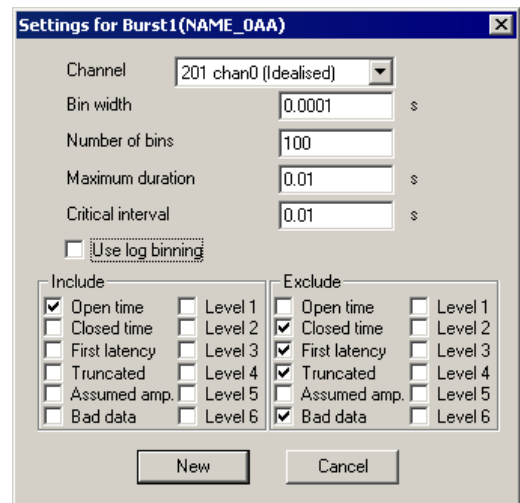
The settings dialog is virtually identical to the Amplitude Histogram settings dialog used for waveform data, the only difference being the Include and Exclude sets, which operate in the same way as for Open/closed time histograms.



Burst duration histogram

Use this option for building burst analysis of open/closed times. The Channel field selects a waveform channel from the source view. This channel must have an idealised trace fitted in order for the histogram to be built. The x-axis of the result starts at zero, so the Bin width and Maximum duration together define the number of bins in the histogram. Use log binning is used to generate histograms with a geometric progression of bin sizes, if it is checked then the Bin width item becomes Minimum duration and the number of bins defines the logarithmic progression.

The two regions labelled Include and Exclude represent the flags associated with each event. A burst duration begins with the start time of an event included (using the rules described for the Open/Closed time histogram) and ends at the start time of an excluded event having a duration longer than the critical interval.



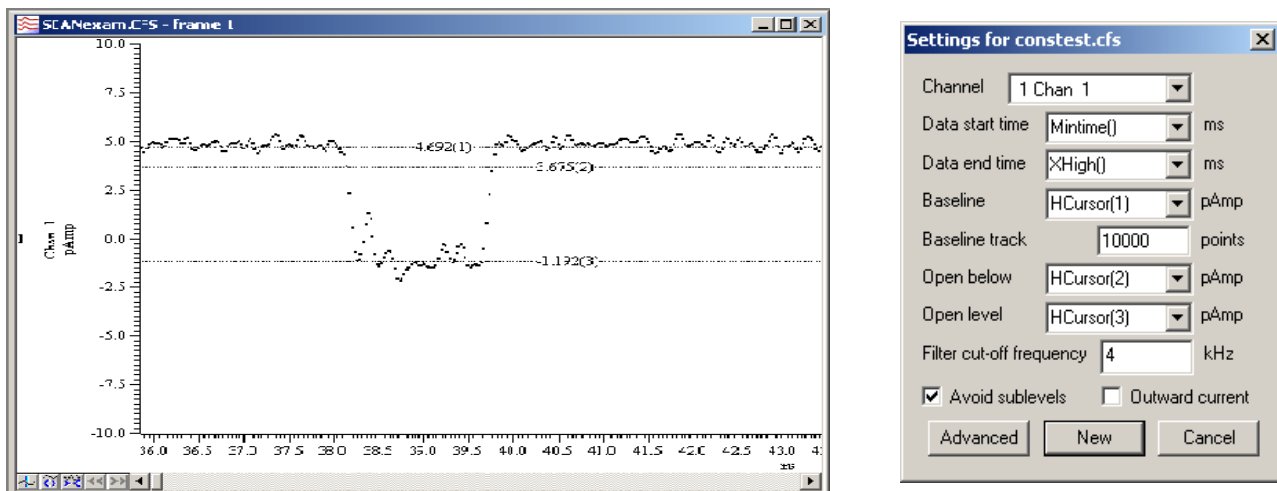
Baseline measurements

This option must be used before any SCAN analysis can be done. A dialog is presented asking for a time range over which the baseline is to be measured and the channel on which to measure it. Clicking OK will then produce a message window with 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 will also be written to the log window.

SCAN method of Idealised trace generation

This is a more complex method of generating an idealised trace. It makes use of an assumption that a Gaussian filter was used to remove noise from the signal to produce a guess of very high time resolution of what the original unfiltered, noise free waveform was. The description here assumes you are using version 4. Version 3 will contain most but not all of the features described.

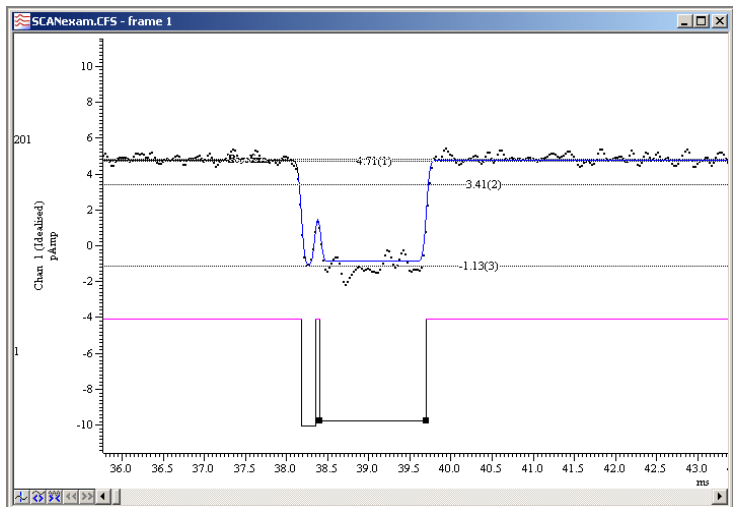
Open the data file SCANexam.cfs in the data sub-folder of your Signal folder. To visualise the quality of a fit you may like to set the draw mode of the data trace to large dots. This avoids the display looking cluttered and it is possible to see if the filtered idealised trace (the convolution) passes through each sampled data point. Next create three horizontal cursors. Set the first to be on the baseline and the second just below it beyond the reach of the noise. The third cursor needs to be placed at the full open level. Browse the openings close to the start of the file and place the cursor at the appropriate level. Using cursors in this way rather than just entering numeric values allows the analysis to modify the cursor positions as it tracks any gradual changes through the data. This in turn means that if you are analysing incrementally then the start conditions for each processing will match the end conditions of the previous process.



Now in the Analysis menu select the Open/Closed times sub-menu and then select New idealised trace (SCAN)... Since there is only one data channel in this file the Channel selection will have been done for you. If you had recorded another channel at the same time for whatever reason, you would need to make sure the channel containing the appropriate data was selected. Set the Data start time to "Mintime()" and the Data end time to "XHigh()". Using XHigh() in this way means that the analysis will stop at the end of the displayed data. This allows small sections of idealised trace to be checked as they are generated and corrections made. Since this data is an inward current you should uncheck the Outward current box. The Baseline, Open below and Open level fields should be set to "Cursor(1)", "Cursor(2)" and "Cursor(3)" respectively. Baseline drift is minimal in this file and a value of 10000 points is fine. Rapidly changing baselines would need a smaller value. Checking the Avoid sublevels box will cause the analysis to insert pairs of transitions in place of some sub-conductance levels. This is not a guarantee that sublevels will not be inserted but just shifts the emphasis away from them. The Filter cut-off frequency for this particular file should be set to 4 kHz. Note that this is the -3dB frequency. Many commercial filters will use a different definition of the cut-off frequency which is about double in value to the -3dB frequency. Halving the value on the front panel of the filter will normally give good results for this analysis. It is also possible to filter the data digitally after it has been sampled. Note that the "Corner" frequency used by Signal's IIR Bessel filter is actually the -3dB point.

We will not adjust the **Advanced** settings at this stage. Until we see how Signal performs with its estimates of transitions we won't know how to adjust these. They are best adjusted during the analysis based on what transitions the software is suggesting as real rather than noise.

Click on **New** then on **Process**; an idealised trace will appear below the data trace with a blue line drawn over the data. This blue line is the "convolution". That is: what the idealised trace would look like if it were filtered and noise free. The aim is to get the blue line to fit the data. The exact number of transitions will depend on the particular levels at which you place the horizontal cursors.



It is possible that there will be a number of spurious transitions that have been inserted to make the convolution follow the data more closely. At this point it is best to remove the transitions by editing the idealised trace. Once this has been done you are ready to do a least squares fit of the convolution to the data. Note that the filter cut-off frequency used to calculate the convolution is stored as part of the channel information and will be initialised to be the same as that used by the SCAN process when the idealised trace is first created. After this time the two copies of this value can be changed independently. The SCAN process version can be changed by using the **Process settings...** dialog and the convolution copy by using the **Channel information** dialog. Both of these options are available by right clicking on the data area. In practice, if you see that the filter cut-off frequency is clearly wrong at an early stage, then you may wish to delete the idealised trace and start again. Such an error will be apparent by the transitions in the convolution appearing too gradual (cut-off too low) or too steep (cut-off too high).

If you haven't already done so to edit the idealised trace, you will need to right click on the data and select: **View and modify events...** In this dialog you can now click **Fit Visible**. At this point the convolution should jump to fit the data and the line turn from blue to black to indicate that the fit was successful. If it does not then the idealised trace will need to be adjusted to allow the fit to take place. Sometimes part of the convolution will change colour but not another. The first unfitted event will be made the current event. Simply concentrate on getting the blue parts of the convolution to fit.

Check the **Resolution** field in the dialog. This is the duration of full amplitude opening needed for the filtered data to reach the trigger level. There is little point in trying to fit events of less than this duration as you may well be fitting noise at that point.

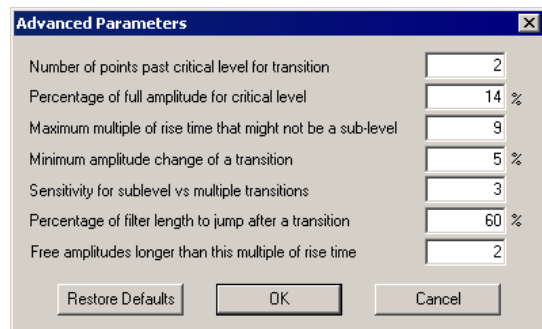
A successful fit will set the current event to the last event on the display. This means that when you click **Next** or **Scroll On** the display moves on to the next possible opening.

On reaching the next possible opening, check that the display does actually show an opening and not just an artefact or noise. If it does not contain an opening then you can click **Next** or **Scroll On** again to advance the display to the next possible opening and extend the closed period at the end of the idealised trace to cover the skipped section. You may also wish to set the flag for this closed period to indicate this is **Bad data** since artefacts may have obscured genuine transitions.

If on reaching the next possible opening you decide that this is a real event then you have two choices. Simply click **Fit Visible** to both perform the SCAN process and least squares fit simultaneously. Alternatively you can click **Process** just to perform the SCAN process part. This second option allows you to modify the initial guess, before doing the least squares fit by clicking **Fit Visible**. Generally speaking it is faster to do both operations is one go first and then go from there. You can click **Undo** at any time to work backwards through your actions.

You may like to backup your work from time to time. This can be done by using the **Backup [filename].sgr** option in the file menu. The sgr file contains various information about the display options of a particular file as well as the idealised trace.

If you are finding that the suggested idealised trace contains far too many extra transitions simply caused by noise then you may wish to consider altering the advanced parameters. The **Advanced** button in the settings dialog or the **Parameters** button in the edit dialog allows various parameters used in the trace formation to be changed. The first thing to try would be the **Minimum amplitude change of a transition**. This is just 5% of the range from baseline to full open level by default. This could be increased to say 15% to eliminate minor sub-level changes that are actually just noise.



Another problem that often occurs is that three rapid full height transitions in a row can be fitted as a pair of transitions separated by a sublevel. This can be minimised by decreasing the **Sensitivity for sublevel vs multiple transitions** value. These parameters are described in the main Signal manual as part of a more detailed description of the SCAN algorithm.

View event list

This will open a window rather like the cursor regions window except this will contain information about the events on view. Each column has a duration and an amplitude with the currently selected event shown highlighted. Closed events will be shown in their own colour. Click on a column and you will find that not only does the selection change in the list but this is also reflected in the idealised trace and in the event details dialog. The list is updated whenever the events around the selected event in the displayed range change

Export to HJCFit

HJCFit is an analysis program produced by Prof. David Colquhoun of University College London. His program allows you to define a possible model for the state changes in the ion channel and test the model by fitting the rate constants within the model to the data in the idealised trace. More information can be found at <http://www.ucl.ac.uk/Pharmacology/dcpr95.htm>.

Script Introduction

Script Introduction

This chapter has two sections. The first part illustrates how a basic knowledge of script language use may save a great deal of time and energy and can provide access to information from the data otherwise unavailable from the menus. Short scripts may save many hours of work by automating simple yet repetitive tasks. Even a single script command can be a useful tool, for example to find the sampling rate used for a data file or to re-title data channels.

The second part introduces you to the basics of Signal (and Spike2) scripts and the script language by examples and description. If you are experienced in programming, you may prefer to use the *Signal Script Language* manual for a more formal approach. You will need this manual (or the on-line Help) to look up the details of script commands.

It is likely that many users of Signal will have very little, if any need to use the script language as the wide range of sampling and analysis options available from the menus mean that the majority of requirements are fulfilled with no additional programming. However, scripts can save you a great deal of time, especially if your analysis means repeating the same list of tasks many times.

Although many people are intimidated by the idea of programming, Signal does provide some help, including recording your actions in script format as well as the provision of scripts written to help you to write your own scripts.

Reasons for writing a script

There are several reasons why you might want to write a script. These include:

- To do things you can't do using the application menus
- To automate repetitive processing of data
- To provide fast online control, a script is faster than you are
- To simplify a task for someone not familiar with the program

What is a script?

A script is a program in its own right that controls Signal. It is a list of instructions and functions that can control all aspects of sampling, display and built-in analysis as well as containing arithmetical functions that can be applied to data and results.

A function is basically a request to perform an operation, and may require additional information in the form of arguments, which can be numbers or text.

An example of a typical function is:

```
FileOpen(name$, type% {,mode% {,text$}});
```

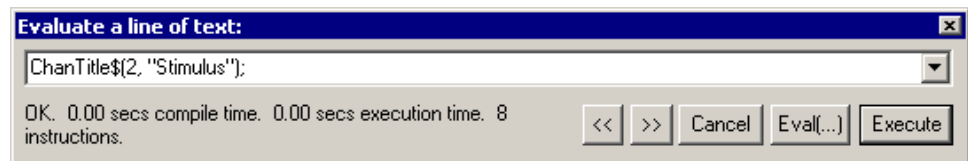
The items within the brackets are the requested arguments. In this case, `name$` requests the name of the file, `type%` determines which type of file it should be (for example a data file or a text file) and `mode%` determines the state of the window when opened (for example visible or maximised). Details of the required arguments are in the on-line help and the Script language manual. When editing a script or using the **Evaluate** window, place the text cursor in the function name and press **F1** to display the relevant help page.

Signal runs scripts in much the same way as you would read them. 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 it run round loops or do one operation as an alternative to another.

The level of detail involved in a script depends on your requirements. It can vary from a single line to a multi-page program to control the sampling and analysis for a complete experiment including user-defined toolbars dialogs and dedicated algorithms applied to the data.

The Evaluate... Window

The simplest possible way to use the script language is from the Evaluate window. This is available from the script menu or by typing `Ctrl+L`. It allows you to enter a single line of script for immediate execution and can be used for a variety of purposes.



For example, open the file `example.cfs` then enter the following into the evaluate window:

```
ChanTitle$(1, "Potatoes")
```

When you click on **Execute** you will see the title for channel 1 has now become "Potatoes". In order to return this to its original title simply replace "Potatoes" with "Sinewave". You can also use **Evaluate** to find out about script functions; position the text cursor within `ChanTitle$` in the window and press the F1 key to see the documentation for this function.

As well as **Execute**, there is an **Eval(...)** button in this window. This can be used to obtain information returned from the commands. For example try typing the following:

```
Binsize(1)
```

When you click on **Eval(...)** you will see the figure 0.01 which is the sampling interval for the channel in question. Similarly, if you type:

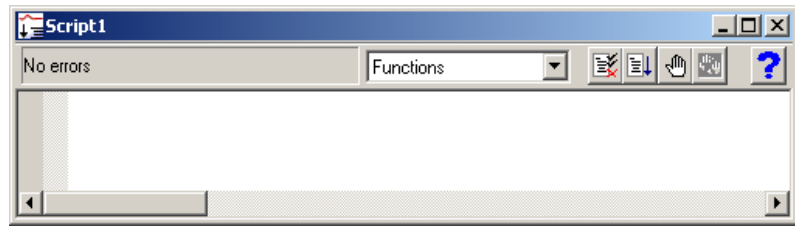
```
1/Binsize(1)
```

and click **Eval(...)**, the number will be 100, giving the sampling rate for the channel.

The **Evaluate** window stores the last 10 lines of script used. The arrow at the end of the command line allows access to previous lines, therefore a selection of frequently used functions can be stored and easily accessed by this method.

The script window

Although the Evaluate window is a useful way of executing a single line of script, for longer scripts a larger window is required that can contain several commands to be executed in sequence and saved to disk for later use.



To create a new script window go to the File menu and select New and then Script Document. You will notice that the window has a drop-down selector and 5 buttons in a toolbar. The selector provides a quick mechanism for finding a function in your script; select the function and the view adjusts to show it.

The button showing a page with ticks and crosses is the compile button. This will check for any errors and compile the script into a form that can be interpreted quickly by Signal. The button showing a page and arrow is the run button. This first compiles and then executes the script. The hand button adds a breakpoint (a point at which the script stops and allows you to see what has happened) at the cursor location; the crossed-out hand removes all breakpoints. The last remaining button with the question mark gives general help with script language functions.

Recording actions to a script

Signal can record actions and present them in script format. This can be extremely useful, especially if there is an action you know how to perform but do not know which function does the same thing in a script.

For example, close the `demo.smr` data file if it is open, then go to the Script menu and select Turn Recording On. Open the `demo.smr` data file and move the window to a new position using the mouse. Go back to the script menu and select "Turn recording off". You should then have a new script containing something like the following:

```
var v11%;
v11%:=FileOpen("C:\\Spike3\\Data\\demo.smr",0,3);
Window(43,0,99,92);
```

If you then close the data file and press the run button on the script window it should then re open the data file in the new position. The first part of the script opens the file with the second part positioning the window on the screen.

The majority of actions can be recorded in this way, however there are limitations. This method can help to get the correct functions for a script but you must bear in mind that in order to create a decent working script, some further work will be required.

For example, the `FileOpen()` function records the name of the file. To use the script on a number of different files, change `"C:\\Spike3\\Data\\demo.smr"` to `" "` and run the script again. This time, instead of loading the `demo.smr` file, you are prompted to select a data file. This behaviour is covered in the documentation for `FileOpen()`. Don't forget, if you click on any of the function names and press F1 the associated help page will be displayed containing information on what the function does and how it can be used.

Useful existing Signal scripts

The software is shipped with a selection of example scripts that may be of direct use to many users or at least provide a good basis upon which to develop their own routines. Some form a skeleton for a particular type of application. For example, the `sampling.s2s` script in the `Spike3\trainday\online` folder is a good basis for on-line scripts. It gives you a framework from which you can start and stop sampling with overall script control.

User-defined toolbars and dialogs can be created through scripts, allowing users to interact with the script by inputting information and linking directly to particular script areas. Although the code for creating these is relatively straightforward, for people interested in writing scripts containing these features it may be a good idea to look at the scripts titled `Toolmake.s2s` and `Dlgmake.s2s`. Using these scripts, you can easily define toolbar buttons and dialogs; the script code to produce them is written by the script itself. The sections produced by this method can then simply be pasted into larger scripts.

Script writing service

At CED we have a large selection of scripts written for a variety of purposes. As well as the demonstration scripts shipped with the software itself, more are available from our web site. If you wish to develop your own scripts, please feel free to contact us with details, we may be able to provide a script upon which to base your own, or even have an existing script that will do what you need. We also provide a script writing service, where we can write a script to your specification, supplied with documentation and tested and supported by CED.

Version 4 script window improvements

The text editor was completely revised for version 4 of Signal, and this allowed us to add features for automatic script formatting, automatic word completion and pop-up call tips to remind you of the arguments used by built-in and even user-defined functions.

The Script language

This document describes the basic processes involved in writing simple scripts (or programs) in Signal and Spike2. The ideas presented are common to many programming languages and the examples will run under either Spike2 or Signal.

Running a script

A script is a list of script language instructions in a file that you then 'run'. When you run a script each instruction is executed in turn. To begin writing a script use the *File: New* menu to open a new script document and then type in the script instructions to be executed. When you have done this you can click on the 'Run' icon at the top right of the script window to run the script.

A first example:

```
'Example: hello
Message("Hello world!");
Halt;
```

When run, this script displays a message "Hello world!" and waits for the user to press OK before ending.

Notes:

- The `'` character starts a 'comment' which is not executed but is merely a comment on the content of the script.

A more complicated example

When run, the following script displays the log window in three different positions on the screen. You must press an 'OK' button on the message box between each change of position.

```
'Example: window
View(LogHandle());
WindowVisible(1);
Window(0,0,30,30);
Message("Window now at top left. Press OK to continue...");

Window(30,30,60,60);
Message("Window now in centre. Press OK to continue...");

Window(60,60,90,90);
Message("Window now at bottom right. Press OK to continue...");
Halt;
```

Notes:

- The `View(LogHandle())` command makes the log window the current view (you needn't worry too much about this for now).
- `WindowVisible(1)` makes the current window visible.
- `Window()` positions the current window at the given coordinates. The first two numbers set the X and Y co-ordinates of the top-left corner of the window, the second two give co-ordinates of the bottom-right corner. All co-ordinates are in percentages of the application window.

Use of variables

Variables are used in a script to hold and calculate values. They can be thought of as 'boxes' whose contents vary but whose name remains the same:

```
'Example: vars
View(LogHandle());
WindowVisible(1);
Window(30,30,70,70);
PrintLog("Start:-\n");

var i%;

i% := 3; PrintLog(i%);
i% := 4; PrintLog(i%);
i% := i% + 1; PrintLog(i%);

var j%;
j% := 4; PrintLog(i% * j%);
j% := i% + j%; PrintLog(j%);

Halt;
```

This script shows how variables are defined and used. The values printed out when this script is run are 3,4,5,20 and 9.

Notes:

- The `PrintLog()` function prints a value to the log file.

- Before a variable is used it must be 'declared' using the `var` keyword. Variables can be declared as one of three types: integer, real and string. Integer variables can only hold whole numbers and always have `%` appended to their names; string variables hold a string of text and have `$` added (e.g. `str$`) and real variables hold a real number which can have a fractional part and have no suffix.

The `:=` symbol should be read as 'becomes' not 'equals'. For instance `i% := i% + 1` should be read as 'the value of `i%` becomes the old value of `i%` plus 1'.

The 'if' statement

The way the script chooses which is the next statement to execute is known as the 'flow of control'. The following is an example of 'conditional execution', that is directing the flow of control using the 'if' statement:

```
'Example: if
var num%;
num% := Input("Type in an integer number please", 0);
if num% < 0 then Message("It was negative!") endif;
if (num% mod 2) = 1 then
  Message("It was odd!");
else
  Message("It was even!");
endif;
Halt();
```

Notes:

- The `Input()` function gets an integer number from the user and stores it in the integer variable `num%`.
- The `if` statement directs the flow of control to the required place depending on whether the expression evaluates to 'true' or 'false'.
- The expression `(num% mod 2)` is evaluated as the remainder when `num%` is divided by 2.

Looping constructs

As well as redirecting program control using the 'if' statement, it is also possible to execute a sequence of statements a number of times by looping back to the beginning once the end is reached. There are three ways of doing this: `repeat ... until`; `while ... wend` and `for ... next`.

First an example of `repeat .. until`:

```
'Example: Mean1
var n, mean, total;
var count% := 0;

repeat
  n := Input("Please input a value", 0.0);
  if n <> -999 then
    total := total + n;
    count% := count% + 1;
  endif;
until n = -999;

if count% > 0 then
```

```
    mean := total / count%;
    PrintLog("Mean is %f\n", mean);
else
    PrintLog("No numbers entered...\n");
endif;

Halt();
```

When this script is run, you are prompted to enter real numbers again and again, until you enter -999. When -999 is entered the script calculates the mean of the numbers entered.

Notes:

- The variable `total` keeps a running total of the numbers entered; `count%` keeps a running total of how many numbers have been entered. The mean is formed by dividing the two.
- The `PrintLog()` statement needs some explanation. The `%f` means ‘print the value of a real variable here’. It is known as a format specifier: other format specifiers begin with `%` and include `%d` (‘print the value of an integer variable here’) and `%s` (‘print the value of a string variable here’). The variables to print are listed as further arguments to the `PrintLog()` function. In the above example, ‘`mean`’ is the variable to be printed.
- The `\n` in the `PrintLog()` statement is a code to tell the script to print a new-line character after printing the mean. A similar printing code is `\t` which tells the script to print a tab character.

Next, a similar example using `while ... wend`:

```
'Example: Mean2

var n, mean, total;
var count% := 0;

n := 0.0;

while n <> -999 do
    n := Input("Please input a value", 0.0);
    if n <> -999 then
        total := total + n;
        count% := count% + 1;
    endif;
wend;

if count% > 0 then
    mean := total / count%;
    PrintLog("Mean is %f\n", mean);
endif;

Halt();
```

Notes:

- Note that the value of `n` must be set to 0.0 initially in order to get into the loop. If `n` was initially set to -999 the loop would never be executed. Contrast this with `repeat...until` where the loop is always executed at least once.

Finally, an example using `for...next`:

```
'Example: Mean3

var n, mean, total;
var count%;

for count% := 1 to 4 do
  n := Input("Please input a value", 0.0);
  total := total + n;
next;

mean := total / 4;
PrintLog("Mean is %f\n", mean);

Halt();
```

Notes:

- This time we loop around the `Input()` statement four times as `count%` takes a value from 1 to 4.

Arrays

Often we would like to use data in a list rather than just single values. To declare a list of data we use the ‘array’ construct. An array is declared using the `var` keyword and can be a list of integers, real numbers or strings. The number of elements in the list is included in square brackets after the variable name in the `var` statement. Subsequently, an individual item from the list is denoted by the array name followed by square brackets enclosing its position in the list.

```
'Example: array

var data%[4];
data%[0] := 10;
data%[1] := 20;
data%[2] := 30;
data%[3] := 40;

var i%;
var total;

for i% := 0 to 3 do
  total := total + data%[i%];
next;

PrintLog("Mean is %f\n", total / 4.0);

Halt();
```

Notes:

- Note the use of `data%[i%]` to get the ‘ith’ element of the array.

Procedures and functions

Often we can simplify a script by enclosing parts of it as procedures or functions. Functions are essentially like the built-in Signal functions but are defined by the user. Procedures are similar but they don’t ‘return a value’.

When run, the following script gets you to open up a data file and prompts you to position a vertical cursor on the data in three different places. The position of the cursor each time is written to the log file.

```
'Example: func
var fh%;
var i%;
var value;

fh% := FileOpen("",0);

if fh% >= 0 then

    Window(10,10,80,80);
    WindowVisible(1);

    for i% := 1 to 3 do
        value := GetMeasurement();
        PrintResult(value);
    next;

    FileClose(0, -1);
endif;

Halt();

func GetMeasurement();
var ret;
CursorSet(1);
Interact("Place cursor at point to measure...", 0);
ret := Cursor(1);
CursorSet(0);
return ret;
end;

proc PrintResult(val)
PrintLog("Value is: %f\n", val);
end;
```

Notes:

- The function `GetMeasurement()` gets you to place a cursor and ‘returns’ the position of the cursor. This is the value which is taken by the variable `value` in the main part of the program. The function `Interact()` allows the user to interact with the data (eg by placing the cursor) before pressing an ‘OK’ button to resume the execution of the script.
- The procedure `PrintValue()` prints the value ‘passed to it’ into the log file.
- This sequence of events happens three time as the ‘for’ loop is executed.

Views and view handles

An important idea to understand in the Signal script language is the concept of a *view* and a *view handle*. Every window that can be manipulated by the script language is called a view. There is always a current view. Even if you close all the views you can find, the Log view is always present (it refuses to close and just hides itself instead). There are many functions which operate on the current view, so you need a way to make a window the current window. This means you need a way to identify a window.

Each view is identified by a number, called its view handle. All script functions that create windows make the new window the current window, and return the view handle of the window (or a negative error code if they fail). The following very simple example opens the example file, draws it and closes it again.

```
'view1.sgs
var vh%;                               'Variable to hold the view handle
vh% := FileOpen("example.cfs",0,1);    'Open the file and make it visible
if vh% <= 0 then message("failed to open window");halt; endif;
PrintLog("The view handle is %d\n", vh%); 'print the handle
Window(0,0,100,100);                  'make window fill the whole screen
Window(50,50,100,100);                'draw window at bottom right
FileClose();                           'close it
```

You will find this script in the training day examples, together with the example file. The `FileOpen` function opens the nominated file and returns the view handle, we store this in the `vh%` variable so we can print it. The `if vh% <= 0` line is checking that we managed to open the file correctly. The two `Window` functions change the screen position of the view and the `FileClose` function closes the file (and the window vanishes).

Those with fast eyes will notice that the contents of the new window are blank! Windows on the screen have two parts: an outer frame that always updates immediately when move or show it, and the inner region with the user data or text. When you run a script, the inner region only updates when the script tells it to with a `Draw` function or if the script is waiting for user input (and so has time to draw). Compare this with the next script:

```
'view2.sgs
var vh%;                               'Variable to hold the view handle
vh% := FileOpen("example.cfs",0,1);    'Open the file and make it visible
if vh% <= 0 then message("failed to open window");halt; endif;
printLog("The view handle is %d\n", vh%); 'print the handle
Window(0,0,100,100);Draw(0,1);        'whole screen, draw 1 second
Window(50,50,100,100);Draw();         'redraw window at bottom right
FileClose();                           'close it
```

This time, the contents of the window are drawn. Signal is very careful not to draw windows except when the script asks, otherwise the screen would tend to flash a great deal. Now suppose we want to position another window:

```
'view3.sgs
var vh%;                               'Variable to hold the view handle
vh% := FileOpen("example.cfs",0,1);    'Open the file and make it visible
if vh% <= 0 then message("failed to open window");halt; endif;
Window(0,0,100,100);                  'whole screen
View(LogHandle());                    'make log view the current window
Window(50,50,100,100);                'redraw window at bottom right
View(vh%);                             'swap back to the file
FileClose();                           'close it
```

The `View` function is used to change the current view. In this case we make the `Log` window the current window and position it, then we make the example file the current view and close it. The `LogHandle()` function returns the view handle of the log window, we can often use a function result just as if it was a variable.

If you want to swap to another view for one function call, then return to the original current view, you can use a different method:

```
'view4.sgs
var vh%;                               'Variable to hold the view handle
vh% := FileOpen("example.cfs",0,1);    'Open the file and make it visible
if vh% <= 0 then message("failed to open window");halt; endif;
Window(0,0,100,100);                  'whole screen
View(LogHandle()).Window(50,50,100,100); 'Place log window at bottom right
FileClose();                           'close it
```

This example is exactly equivalent to the previous one. The `view(x).Command(...)` syntax means save the current view, make the view with handle `x` the current view and run `Command(...)`, then restore the original current view, it provides a simple way of operating on a specified view.

You can set the current view in several ways:

1. The `view(x)` function or `view(x).function`.
2. Any function that creates a new view, like `FileOpen` or `FileNew` or the analysis functions that generate a memory view.
3. `FrontView(x)` makes the view with handle `x` the current view and also brings it to the top so that it is visible.

Script toolbox

Script toolbox

View manipulation

The underlying view system was introduced in the previous session. Perhaps the most important function in the whole system is `View(...)` together with the associated `View(...)` operator. The single most common error in a script is not having the correct current view for an operation, resulting in the "View is wrong type" error.

Positioning the view

The next little script moves the current view around the screen. Notice that areas wiped out by the window are not repainted until the script finishes. This is an important point. While a script runs, updates do not occur except in response to `Draw(...)` functions and when you give the system idle time (such as when a dialog box is displayed or when you use the `Interact(...)` or `Toolbar(...)` functions).

```
'RandWind.sgs
FrontView(LogHandle());           'Get a window at the front
Seconds(0);                       'Zero our time counter
while Seconds() < 10 do
  Window(Rand()*100, Rand()*100);
wend;
```

The `Rand()` function returns a random number from 0.0 up to, but not including 1.0. When you use the `Window(x,y)` function in this form, the `x` and `y` positions set the position of the top-left corner of the window as a percentage of the available area. You can also append two more arguments which are the `x` and `y` position of the bottom-right corner of the window. As an exercise, add two more arguments to randomise the size and run that. The full form of the function is: `Window(xL, yL{, xH, yH})`. Arguments in curly brackets are optional.

Drawing and updating

Another important function in the script when dealing with data views is the `Draw(...)` function. Signal is carefully written so that windows do not update until either you command them to, or there is idle time in which to do it. If this were not the case, scripts would run very slowly as any time that the script caused any portion of a window to become invalid, the program would have to stop and redraw that portion. Instead of this, each window remembers which portions have become invalid, and updates them when time becomes available. One way you make time available is with `Draw({from {,size}})`. If you omit the arguments, the window is redrawn with the same size or start and size as the last time. Draw is fairly smart: if nothing has changed, it doesn't waste time drawing and if it can achieve a move by scrolling the current view, it will.

This next example displays 10% of the current frame, opening a file if necessary, then scrolls through the remainder of the file 5% at a time.

```
'DrawView.sgs
var vh% := 0;
var w, t;
var mt;
if ViewKind() <> 0 then           'if current not a file view
  vh% := FileOpen("",0,0,"Select data file to display");
  if vh% < 0 then halt endif; 'Stop if no file
  Window(0,0,100,50);           'Set display position
endif;
ViewStandard();                 'All channels on view
mt := MinTime();                'For Signal users
'mt := 0;                        'For Signal users
w := MaxTime() - mt;            'Width of the frame
```

```
XRange(MinTime(), mt+(w/10));           'Display 10% of all data
WindowVisible(1);                       'Make visible
for t := 0 to 0.90*w step 0.05*w do
  Draw(t);
next;
```

The heart of this example is the last three lines. All the rest of the script checks to make sure we have a suitable view to work with. We start by checking that the current view is a file view; if it is not we prompt the user to select a suitable file from disk. Notice that the `FileOpen(...)` function makes an invisible window so that the functions to position the view do not cause the window frame to flash (which looks a mess and can force other windows to redraw).

`ViewStandard()` is a very useful function when you want to get a data view into a known state before starting to make changes to it.

`XRange(dispStart{,dispEnd})` sets the display range for the next time a window is drawn, but unlike `Draw(...)`, it does not force the window to draw. If you replace the `Draw(t)` in the script with `XRange(t)` you will only see a single update when the script stops and Signal has idle time in which to sort out the update.

`MaxTime()` is an important function to remember. When used in a data view with no arguments it returns the time of the last data item in the data frame. You can also include a channel number as an argument, in which case it returns the time of the last data item on the channel. `MinTime()` is just the same, but returns the time of the first item. This is often zero, but may not be, so note the calculation and use of the frame width to allow for this.

When we are dealing with a data view, there are useful functions for turning axes and grids on and off. These functions are self-explanatory, see the functions `Grid()`, `XAxis()` and `YAxis()` for details. You could add these to the previous script if you wanted to customise the display.

Cursor functions

You can use both horizontal and vertical cursors in data views. The `CursorXXX(...)` family controls vertical cursors; the `HCursorXXX(...)` family controls horizontal cursors. Cursors are created by `CursorNew()` and `HCursorNew()` which add one cursor if there are not already the maximum number of cursors, and `CursorSet()` which creates and or deletes up to 10 vertical cursors. There is no `HCursorSet()`.

Basic cursor functions

The following example prompts a user to position 4 cursors to enclose an area of a Waveform channel. We'll stick with the Example data file for this example:

```
'Cursor1.sgs
ToolbarText("Cursor functions");           'Stop screen jumping
if UCase$(FileName$(3)) <> "EXAMPLE" then Message("Wrong file!"); halt endif;
ViewStandard();FrontView();               'Get into a tidy state
ChanShow(-1,0);ChanShow(1,1);             'Show channel 1 only (waveform)
CursorSet(2);                              'Add two vertical cursors in the window
HCursorNew(1,1.0);HCursorNew(1,-1);       'These will be cursors 1 and 2
Interact("Position around the feature",4+32); 'let user modify things
CursorRenumber();HCursorRenumber();        'Get cursors in the right order
Message("%8g %8g seconds\n%8g %8g %s",Cursor(1), Cursor(2),
      ChanValue(1,Cursor(1)), ChanValue(1,Cursor(2)), ChanUnits$(1));
CursorSet(0);                              'Easier than calling CursorDelete() twice
while HCursorDelete() do wend;              'Remove all horizontal cursors
```

The main difference between vertical and horizontal cursors is that horizontal cursors are associated with a channel. You can set a horizontal cursor on any channel, but they are useful only for channels that have a y-axis. You can set a horizontal cursor on an event channel drawn in dots mode, but it is not very useful!

Renumber cursors

A very common situation occurs where you want the user to select an area of data for analysis with two or more cursors. The problem is that the user may well put the cursors in the wrong order. You can avoid this by labelling the cursors, but by far the simplest solution is to use `CursorRenumber()`:

```
'cursor2.sgs
ToolbarText("CursorRenumber() demonstration");
If ViewKind()<>0 and ViewKind()<>4 then
  message("Needs time or memory view selected to run");
  halt;
endif;
var sVis%;
sVis% := View(App(3)).WindowVisible(0);      'Hide script, and save state
FrontView();ViewStandard();                  'Make it visible
CursorSet(2);CursorLabel(2);                 'show two vertical cursors (numbered)
Interact("Swap cursors over and click OK",4+32);
CursorRenumber();                             'Get cursors in correct order
Interact("Now they are back in order",0);
View(App(3)).WindowVisible(sVis%);           'restore script to previous state
```

In addition to demonstrating how to renumber cursors (there is also a `HCursorRenumber` which renumbers from the top of the screen to the bottom), this script shows you how to hide the current script (which otherwise tends to get in the way unless you have hidden it yourself). The script that is running is usually hidden from the program to prevent your deleting it accidentally... However, you can get the script handle, and some other useful handles with the `App()` function. You might also consider looking at `SampleHandle()` if you want handles to exotic windows.

Interact

`Interact` is an easy way to let the user manipulate data, usually with the cursors, with the script paused. When `Interact` is active, the Signal system is in an idle state, and so will update any screen area that has been made invalid by a script activity, or by data sampling.

```
Func Interact(msg$, allow% {,help {, lb1$ {,lb2$ {,lb3$...}}});
```

You can use the message and the various labels to create buttons that the user can use to exit from `Interact`.

Central to the `Interact` command is the `allow%` parameter. As its name implies the `allow%` parameter determines what the user can do while `Interact` is running. Setting `allow%` to 0 would restrict the user to inspecting data and positioning cursors in a single, unmoveable window. The `help` parameter 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. Set 0 to accept the default help.

```
'interact.sgs
var btn%, msg$;
btn% := Interact("Choose a button", 0 ,0 , "one", "two", "three");
docase
  case btn% = 1 then msg$ := "one";
  case btn% = 2 then msg$ := "two";
  case btn% = 3 then msg$ := "three";
  else msg$ := "none";
endcase;
Message(msg$);
```

The above example also shows the use of the case statement. This can be handy if you want to test for several different values of a variable. As you can see Interact returns a value which corresponds to the button which was pressed.

The Toolbar family of functions

This is a most important family of functions, particularly if you want to run an on-line script. The toolbar has many of the features of Interact, except that instead of returning to the script each time you press a button, you have the option of linking a button to a user-defined function that is run once each time the button is pressed. What happens after a function runs depends on the return value of the function. You can also nominate a function that is called during idle time whenever there is no button pressed and nothing else for Signal to do.

To make using the toolbar family easier, there is a script, ToolMake.sgs that can be used to write the skeleton of a Toolbar-based script for you. This script can also simulate your toolbar for testing purposes. The script below was generated using Toolbar. This script is not on the disk. To create it, run `ToolMake` and follow these steps:

1. Click "Add button" and then OK (to add button 1). Edit the label to "Quit", leave the function name blank (we do not want to run a function when we quit), and leave the "This button closes toolbar" box checked.
2. Click "Add button" and then edit the button number to 3. By not using button 2 we create a gap on the toolbar to provide a visual break between buttons. Click OK, then in the next dialog set the label to "Action1" and the function name to `DoAction1` and leave the checkbox unchecked as this button does not close the toolbar.
3. Click Idle to add an idle time function and give it the name `MyIdle`.
4. Click Test to check that the result is as intended. Click Quit once it is OK. You can edit buttons if you have made any errors.
5. Click Write and then set check-boxes to determine what can be done while the toolbar is active. I usually set "Can move and resize" and "Can use View menu" unless I have good reasons for using other check boxes.
6. Click Quit and then test the new script.

```
'Generated toolbar code
DoToolbar();  'Try it out
Halt;

Func DoToolbar()                                'Set your own name...
ToolbarClear();                                'Remove any old buttons
ToolbarSet(0, "", MyIdle%);                     'Idle routine
ToolbarSet(1, "Quit");                          'This button returns its number
ToolbarSet(2, "Action1", DoAction1%);          'Link to function
  return Toolbar("Your prompt", 36);
end;

Func MyIdle%()                                  'Idle routine - repeatedly called
'Your code in here...
return 1;  'This leaves toolbar active
end;

Func DoAction1%()                              'Button 2 routine
'Your code in here...
return 1;  'This leaves toolbar active
end;
```

We now have the skeleton of an application. You can enable and disable the buttons using `ToolBarEnable()`. For example, we can add the following to the `MyIdle()` function:

```
var enable%;
enable% := Trunc(Seconds()) mod 2;           'either 0 or 1
ToolBarEnable(1, enable%);
ToolBarEnable(2, 1-enable%);
```

If you try this you will find that the two action buttons are alternately enabled and disabled once a second.

Input, Input\$ and Query

These functions are usually used for “quick and dirty” scripts to get a rapid response from the user to a single question. If you want more than one piece of information at a time, you are far better off using the `Dlg...` family of commands discussed below. `Input()` reads a number with optional limits, `Input$()` reads a string with optional character filtering, and `Query()` gets the user response to a Yes/No type question. Here is a script that could make you a multi-millionaire...

```
'UserIO.sgs
var name$, lucky%, i%, nums%[7], j%, t%, x%;
ToolBarText("Mystic Greg's Magic Lottery Predictor");
repeat
  name$ := Input$("Your name here please","",12,"a-zA-Z");
  if Len(name$)=0 then Message("Oh, don't be so shy...") endif;
until Len(name$);

if Query("Do you have a lucky number?",
        "Yes and I'll tell you",
        "None of your business") then
  lucky% := Input("What is your lucky number?",7,0,255);
endif;

for i%:=1 to Len(name$) do           'use name to generate the seed
  lucky% := lucky% + Asc(Mid$(name$,i%,1));
next;
Rand(lucky% / (Len(name$)+1.0));    'Seed random number generator
nums%[0] := Rand()*49 + 1;          'make 1 to 49 as first number
for i% := 1 to 6 do                 'loop round for next 6
  repeat
    t% := Rand()*49 + 1;            'generate 1 to 49
    for j% := 0 to i%-1 do          'now check not already used
      if t% = nums%[j%] then t% := 0 endif;
    next;
  until t%;                          't% is 0 if number already used
  nums%[i%] := t%;                  'save a number
next;

'This is a very crude sort, but ok for small number of items
for i%:=0 to 4 do                   'sort them into order (not bonus ball)
  t% := nums%[i%];                  ' get first number
  for j% := i%+1 to 5 do             ' see if smallest number
    if nums%[j%]<t% then              ' if number is smaller
      x%:=nums%[j%];                 ' swap it with test number
      nums%[j%]:=t%;
      t% := x%;
    endif;
  next;
  nums%[i%] := t%;
next;

Message("Your numbers: %d\nBonus ball: %d", nums%[:6], nums%[6]);
```

The Dlg... family of functions

The above script is rather tedious to run because it keeps putting up new dialogs, which is distracting when you want several items of information at the same time. If you run UserIOD.sgs you will see that we have combined the separate prompts into a dialog box, plus new features. The bulk of the changed code is:

```
DlgCreate("Mystic Greg's Lottery predictor"); 'Start new dialog
DlgString(1,"Your name please, oh mighty one!",20,"a-zA-Z");
DlgInteger(2,"Your lucky number, valued friend",0,255);
DlgCheck(3,"Include lucky number in calculation");
var option${3};
option${0}:="Use above data";
option${1}:="Look into future";
option${2}:="Run again";
DlgList(4,"Extra mystic passes",option${});
if not DlgShow(name$,lucky%,useLuck%,option%) then return -1 endif;
```

This code looks a little fearsome to write yourself, but fortunately, you don't need to! I generated this code by running the script DlgMake.sgs and pressing buttons and typing prompts. Then I pasted the result into the original script, changed a few variables and the job was done.

The steps in making a dialog box yourself are:

Use DlgCreate(...) to start the creating process, give the dialog a name, and optionally position and size the dialog.

1. Use DlgChan(...), DlgCheck(...), DlgInteger(...), DlgLabel(...), DlgList(...), DlgReal(...), DlgString(...) and DlgText(...) to define fields in your dialog
2. Use DlgShow(...) with a variable for each field to be filled to display the dialog and collect the values.

The DlgMake script only generates simple dialogs with the fields stacked vertically. If you are prepared to try a bit harder, you can create all sorts of fancy dialogs (but we leave this as an exercise for the student). You can use the DlgMake script itself as an example of more complicated dialog creation.

Maths and Array arithmetic functions

Signal provides a basic set of maths functions, and built-in functions to manipulate arrays. You can derive most other reasonably common functions from the built-in set. However, if you find that the lack of Bessel functions, or something else similar, is a real problem, then let us know as it is relatively easy to add new ones. In addition to the array functions there are also functions to directly modify channel data and to use the frame buffer.

In addition to using maths functions on a single value to return a single result, you can also apply them to an array. It is usually MUCH FASTER to use the array method than using a program loop:

```
'array1.sgs
var data[10000];
var i%,t1,t2;
seconds(0); 'zero the timer
for i%:=0 to 9999 do
  data[i%] := 5.0*cos(i%/1000.0); 'beware i%/1000!!!
next;
t1 := Seconds(); 'see how long it took
Seconds(0); 'zero the timer
```

```

data[0]:=0;                               'first point is 0
ArrConst(data[1:],0.001);                 'set the same except first
ArrIntgl(data[]);                         'form a ramp
Cos(data[]);                              'take cosine
ArrMul(data[],5.0);                       'form product
t2 := Seconds();

```

```
Message("By hand %g seconds\nArrays %g seconds", t1, t2);
```

On my computer, the hand written loop took 2.87 seconds. The same calculation using the array arithmetic took 0.028 seconds (I had to time 100,000 bins to get an accurate time).

If you need access to standard constants, π is `4*ATan(1)` and e is `Exp(1)`.

String functions

Signal scripts can use the usual string handling functions that you would expect:

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

Most of these are easy to use, and `Print$` has been mentioned before. `ReadStr(...)` is very useful when you need to parse a line of text containing several data fields separated by spaces and/or commas or tabs.

Many script writers forget that you can use `+` to add two strings together, `+=` to append a string and the comparison operators `<`, `<=`, `>`, `>=`, `=` and `<>` to compare strings. Remember that the comparison operators and `InStr()` use case sensitive comparisons. If you require case insensitive comparisons, use the `LCase$()` or `UCase$()` functions before the comparisons:

```

if jim$ > sam$ then DoSomething() endif;           'Case sensitive
if LCase$(jim$) > LCase$(sam$) then DoIt() endif; 'Case insensitive

```

When strings are compared, the comparison is left to right, character by character. Characters later in the alphabet are larger. The length of two compared strings only matters if both strings are the same to the length of the shorter string, in which case the longer string is considered greater.

File functions for text and binary files

These functions let you read and write external data files without the overhead of attaching them to a window. Binary file input and output is usually used to import or export a foreign data format, suffice it to say that you open or create an external binary file using `FileOpen(...)` with mode 9, you move to a particular offset in a file with `BSeek(...)`, and you read data into variables or arrays using `BRead(...)` or `BReadSize(...)` and write data with `BWrite(...)` and `BWriteSize(...)`. You close an external file using `FileClose()`, just as for any other type of file. External files have a view handle in exactly the same way as any other file, but these views are always invisible. External files are closed automatically when a script ends, should you forget to close them yourself.

XY views

In its most basic sense, an XY view is a way of plotting any value against any other value. The points can be plotted using a variety of symbols and colours. The data is still divided into channels and the data points within each channel may be joined or unjoined.

Creating an XY view from a script

An XY view always has at least one data channel, so when you create a view, you also create a channel. The following script code shows you how to make an XY view:

```
var xy%;                               'handle for the XY view

xy% := FileNew(12,1);                  'type 12=XY, 1=make visible now
```

Then if you want to add additional channels you can do this using the `XYSetChan()` command. You can also use this command to set a channel to a particular state. The following sets channel 1 (the first channel) to show data points joined by lines with no limit on the number of data points, drawn in the standard colour:

```
XYSetChan(1, 0, 0, 1);                 'chan 1, no size limit, no sort, joined

XYDrawMode(1, 2, 0);                   'set a marker size of 0 (invisible)
```

To add data points to a channel you use the `XYAddData()` command. You can add single points, or pass an array of x and y co-ordinates. The following code adds three points to draw a triangle:

```
XYAddData(1, 0, 0);                    'add a point to channel 1 at (0, 0)

XYAddData(1, 1, 0);                    'add a point at (1,0)

XYAddData(1, 0.5, 1);                  'add a point at (0.5, 1)
```

You will notice that the result of this draws only two sides of a triangle. We could complete the figure by adding an additional data point at (0,0), but it is just as easy to change the line joining mode to "Looped", and the figure is completed for you:

```
XYJoin(1,2);                            'set looped mode
```

In addition to the above commands there are a host of other commands which may be used with XY view. These are listed below:

<code>XYColour()</code>	Gets or sets the colour of a channel.
<code>XYCount()</code>	Gets the number of data points in a channel.
<code>XYDelete()</code>	Deletes a range of data points or all data points from one channel.
<code>XYGetData()</code>	Gets data points between two indices from a channel.
<code>XYInCircle()</code>	Gets the number of data points inside a circle.
<code>XYInRect()</code>	Returns the number of data points in a channel.
<code>XYKey()</code>	Gets or sets the display mode and positions of the channel key.
<code>XYRange()</code>	Gets the range of data values of a channel or channels.
<code>XYSize()</code>	Gets and sets the limits on the number of data points of a channel.
<code>XYSort()</code>	Gets or sets the sorting method of a channel.

A good example of an XY view is the clock script which is shipped with both Spike2 and Signal. In this script the clock face is drawn using a single channel of 12 data points which are not joined and drawn as large solid blocks. Each of the clock hands are drawn using a channel each and are drawn using invisible points which are looped.

Scripts and Signal data

Scripts and Signal data

Up to this point the scripting talks given have been non-specific in the application they refer to. The way in which data is accessed and processed in Signal is, however, very different from Spike2.

A review of arrays

Almost all data you will be manipulating in Signal will be in the form of an array. For this reason it is essential to have a good understanding of what arrays are and how to use them.

Put simply an array is a collection of variables (elements) all having the same name and referred to individually by using a number (the index). If you have an array called “dat” with say ten elements of floating point numbers then this would be declared using:

```
Var dat[10];
```

The first element of any array always has index 0, so in this case the first element would be `dat[0]`. This of course means the last element would be `dat[9]`. Many functions both built in and user defined can take arrays as arguments. For instance: `Cos(dat[])` will convert each element of array `dat` to its cosine. `Cos(dat[3:4])` will convert just 4 consecutive elements starting with index 3. Here `dat[3:4]` is what is known as a sub-array. Other examples of sub-arrays would be `dat[6:]` which would be the latter part of the array starting at index 6 and `dat[:7]` which would be the first 7 elements of the array. If you need to find the size of an array you can use the `Len()` function, so `Len(dat[])` would return 10.

It is also possible to have arrays of arrays. These are known as two-dimensional arrays. They are declared in a very similar manner but with an extra index:

```
Var moreDat[20][30];
```

Individual elements now require two indices to specify. It is also possible to refer to one-dimensional sub-arrays by eg. `moreDat[11][]`.

Data views as arrays

The script language sees a channel in a data view as an array attached to a view. To access an individual array element use `View(v%,c%).[index]` where `v%` is the view handle, `c%` is the channel number and `index` is the bin number, starting from 0. You can pass a channel from a data view to a function as an array using `View(v%,c%).[]` for the entire array and `View(v%,c%).[a:b]` for a sub-array starting at element `a` of length `b`. `View(0,1).[0:3]` for the first three values in channel 1 of the current view.

Marker channels act as arrays holding time values, they are read-only data because marker data times must be kept sorted. The only way to change a marker time is by using the `MarkTime()` function.

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

```
'chandata.sgs
var v%, index%;
v% := FileOpen("",0,1);
if v% > 0 then
for index% := 0 to ChanPoints(1)-1 do
  View(v%,1).[index%] := index%; 'set all the points in channel 1
  Draw();
next;
endif;
```

Run this script and you will see how the data in the view can be accessed directly. At this point, only the data in memory has been changed. The data on disk is untouched. Step to the next frame and you will be prompted with what to do next. This prompt will determine whether the changes are to be written back to the file or discarded. If you check the “Adjust data file update mode to match” then instead of prompting you there after, it will do whatever you choose this time. The data file update mode can also be set in the preferences dialog next to “Save changed data”.

This sort of direct access to the channel data is particularly useful in script programming and is the standard mechanism used to manipulate or access file data. Though channel data manipulation is possible using specialised functions such as `ChanRectify()` and other functions that match the channel modification options available from the Analysis menu, it is often easier to work on the data directly.

Creating memory views

Creating new data from old is something almost all script writers are going to want to do at some stage. Hopefully you are familiar with creating memory views. Creating them from a script is very much the same in that there is a script command corresponding to each of the dialogs. To see how this works let's try creating an average interactively with recording on and look at the script it produces. Open `example.cfs` and create an average by using the analysis menu and accepting the default settings. You should get a script like the following:

```
var v6%;
var v7%;
v6%:=ViewFind("example");
FrontView(v6%);
v7%:=SetAverage(-1,0.04,0,0,0);
WindowVisible(1);
ProcessFrames(1,-1,-1,0,1,1);
```

As you can see; the script first makes sure that the data view is current then calls `SetAverage()`. This does the actual creating of the memory view. All functions which create memory views have the form: `SetXXX()` and to-date, there are the following functions: `SetAutoAv()`; `SetAverage()`; `SetPower()` and `SetLeak()` these correspond to the New memory view options in the analysis menu. In addition to these specialised functions there are also `SetCopy()` and `SetMemory()` which we will come back to later; they are used to create memory views for direct manipulation.

Let us look more closely at `SetAverage()`. It has the form:

```
Func SetAverage(ch%|list%[]|ch${, width, offs {, sum%{ ,xzero%}}});
```

ch% A waveform channel to analyse from the current view. Use a channel number (1 to n), or -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.

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

list%[] As an alternative to **ch%** or **ch\$**, you can pass in a channel list (as constructed by `ChanList()`). This must be an array of waveform channels in the current data view, with the first element of the array holding the number of channels in the list.

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.

Like all functions that create a view this one returns the view handle and creates the view invisibly. That's why the recorded script has `WindowVisible(1)` just after the call to `SetAverage()`. It is important to realise that the `SetAverage()` call does not actually do the average. This is done by the `ProcessFrames()` call. The syntax of this being:

```
Func ProcessFrames(sF% {,eF% {,mode%{,clear% {,opt% {,optx%}}}}});
```

```
Func ProcessFrames(frm$|frm%[]{|,mode%{,clear% {,opt% {,optx%}}}}});
```

sF% First frame to process. This option processes a range of frames. **sF%** can also be a negative code as follows:

```
-1 All frames in the file are included
-2 The current frame
-3 Frames must be tagged
-6 Frames must be untagged
```

eF% Last frame to process. If this is -1 the last frame in the data view is used. This argument is ignored if **sF%** is a negative code.

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

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

`mode%` If `mode%` is present it is used to supply an additional criterion for including each frame in the range, list or specification. If `mode%` is absent all frames are included. The modes are:

```
0-n   Frames must have a state matching the value of mode%
-1    All frames in the specification are processed
-2    Only the current frame, if in the list, will be processed
-3    Frames must also be tagged
-6    Frames must also be untagged
```

`clear%` If present, and non-zero, the memory view bins are cleared before the results of processing all the frames are added to the view and `Sweeps()` result is reset.

`opt%` If present, and non-zero, the display of data in the memory view is optimised after processing the data.

`optx%` For XY views only, if present, and non-zero, the X axis in the XY view is optimised after processing the data.

The function returns zero if there are no errors or a negative error code.

Now consider this script:

```
var v6%;
var v7%;
v6%:=ViewFind("example");
FrontView(v6%);
v7%:=SetAverage(-1,0.02,0,0,0);
WindowVisible(1);
Process(-0.01);
Process(0.01);
```

The difference here is the `Process()` command. There is also a change to the second parameter of the `SetAverage()` command to half the time width of the average. The `Process()` command takes a start time for processing and acts only on the current frame of the data view. In the example, we are averaging together the two halves of the current frame of `example.cfs`.

There are two other processing commands worth mentioning. One is `ProcessAll()` which is like `ProcessFrames()` but as you might expect, acts on all processes attached to a particular view. The other is `ProcessOnline()`, which is just an on-line equivalent of `ProcessFrames()`.

SetCopy() and SetMemory()

If we just wanted to create a memory view by hand in which to display and/or store data from our own calculations then these would be the commands to use. Here is an actual example, which was requested by a Signal user who wanted to subtract one channel from another and store the result. Since there is no way to add a channel to a CFS file once it has been created a memory view with three channels needed to be created; the two original data channels copied into the memory view together with a third channel made up of the subtracted data.

```
' chansub.sgs
' Script to subtract one channel from another in a CFS file
' Make the data view current and run the script: a second view will be created with
' the old and new data together. Save this to create a new file.

const chanToSubFrom% := 1;
const chanToSub% := 2;

var frames%;
var chans%;
var pts%;
var timeview%, resView%;
var binsz;
var title$;
var chan%;
var frame%;

title$ := WindowTitle$();
binsz := BinSize(1);
timeView% := View();
chans% := ChanCount(-4);
frames% := FrameCount();
pts% := ChanPoints(1);
resView% := SetMemory(chans%+1,pts%,binsz,0,0,0,0,title$,"s"); 'create new view
for chan% := 1 to chans% do 'copy chan details
    View(resView%).ChanTitle$(chan%, View(timeView%).ChanTitle$(chan%));
    View(resView%).ChanUnits$(chan%, View(timeView%).ChanUnits$(chan%));
next;
View(resView%).ChanTitle$(chans%+1, "Subtracted");
View(resView%).ChanUnits$(chans%+1, View(timeView%).ChanUnits$(chanToSubFrom%));

for frame% := 1 to frames% do
    View(timeView%).Frame(frame%);
    if frame% <> 1 then
        View(resView%).AppendFrame();
        View(resView%).Frame(frame%);
    endif;
    for chan% := 1 to chans% do 'copy old data to new view
        ArrConst(View(resView%,chan%).[], View(timeView%, chan%).[]);
    next;
    'Now place subtracted data in new channel
    ArrConst(View(resView%,chans%+1).[], View(timeView%, chanToSubFrom%).[]);
    ArrSub(View(resView%,chans%+1).[], View(timeView%, chanToSub%).[]);
next;
View(resView%).Optimise(-4); ' optimise and display the new view
View(resView%).WindowVisible(1);
```

`SetCopy()` is very similar to `SetMemory()` except that it takes its settings from an existing data view with the option to copy data from it as well.

Appending frames

Memory data can also be added to the end of an existing data file using the `AppendFrame()` function. This adds a frame of memory data to the end of the current view. The data in this new frame is either all zeroes or a copy of the current frame's data.

You can use `AppendFrame()` for any purpose necessary, possible uses include building a composite file containing data from multiple files, appending averages to the end of a data file or just generating and saving synthesised data.

Appended frames will be written to the disk file when the view is changed. Appended frames that have not been written to disk can be removed using `DeleteFrame()`.

Creating Trend plots

You may well have noticed in the above descriptions of the process commands that they can also be used to process XY views. In just the same way as memory views can be created, there is also a `SetTrend()` which can be used to create trend plots. It is even possible to have multi-channel plots by using `SetTrendChan()`. To create an example, try turning recording on and generating a trend plot.

```
var v5%;
var v11%;
v5%=ViewFind("example");
FrontView(v5%);
v11%=SetTrend("Channel 1",4,1,"MaxTime()", "0",0,0,1,"Cursor(1)", "0",0,0);
WindowVisible(1);
ProcessFrames(1,-1,-1,0,1,1);
```

The trend plot dialog has a lot of information in it and consequently the corresponding `SetXXX()` command has lots of parameters. Here is the syntax:

```
Func SetTrend(na$, xt%, xc%, xp|xp$, xb|xb$, xw, yt%, yc%, yp|yp$, yb|yb$, yw {, pts%});
```

na\$ The name of the channel. The channel name is shown in the XY key area.

xt% The type of measurement to take for the X part of each point. The possible values are:

0	Value at a point
1	Value difference between points
2	Time at a point
3	Time difference between points
4	Frame number
5	Absolute time of frame
6	Frame state value
7	Fit coefficient
8	User entered value
9	Area between points
10	Mean between points
11	Slope between points
12	Area over zero between points
13	Sum between points
14	Modulus between points
15	Maximum value between points
16	Minimum value between points
17	Amplitude value between points
18	RMS Amplitude value between points
19	Standard dev. value between points
20	Absolute maximum value between points
21	Peak found between points
22	Trough found between points

-
- `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).
- `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.
- `yt%` The type of measurement to take for the Y part of each point. The possible values are the same as for `xt%`.
- `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).
- `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.

Extra channels may be added using `SetTrendChan()` which takes exactly the same parameters as `SetTrend()` but does not return a view handle.

Signal scripts online

Signal scripts online

This session assumes that you are familiar with the essentials of using scripts with Signal and have access to either *The Signal script language* manual, or the *Signal for Windows* on-line help system so you have the descriptions of the functions we will use. For reasons of space, I am not going to repeat the full function descriptions here and we assume that you can easily look up argument definitions.

Signal scripts can be used to configure sampling, to control the sampling process or to perform specialised on-line analyses. The DoEmg script shown in appendix 1 at the end of this manual uses many of the techniques discussed in this section, other scripts worth looking at for online techniques are Artefact and SOnline.

Configuring sampling

The `Sample` family of functions can be used within a script to set the sampling configuration and to control the sampling from start to finish. They are used in on-line scripts so that the user does not have to manually load a configuration for the experiment or to tweak a configuration by, for example, disabling writing to disk.

The following functions (and many, many others) set up the configuration to be used for sampling:

<code>SampleBurst()</code>	<code>SampleClear()</code>	<code>SampleKeyMark()</code>	<code>SampleDigMark()</code>
<code>SampleLimitFrames()</code>	<code>SampleLimitSize()</code>	<code>SampleLimitTime()</code>	<code>SamplePeriXXXX()</code>
<code>SamplePoints()</code>	<code>SamplePortFull()</code>	<code>SamplePortName\$()</code>	<code>SamplePortZero()</code>
<code>SamplePorts()</code>	<code>SampleRate()</code>	<code>SampleTrigger()</code>	

These functions can be used to set up a fixed precise sampling configuration, to modify a preloaded sampling configuration or a dialog can be used to enter parameters from which the sampling configuration could be built. There are also copious `Pulsexxx` functions that can be used to set up output pulses. A simple example of synthesising a sampling configuration from scratch is taken from `Sonline.sgs`:

```
SampleClear();           ` Initialise the configuration
SampleMode(0);          ` Basic mode
SampleBurst(1);         ` Burst mode
SamplePorts(2);         ` 2 ADC ports (0 and 1)
SamplePortUnits$(0,"Volt"); ` Set up port 0
SamplePortFull(0,5);
SamplePortZero(0,0);
SamplePortName$(0,"ADC 0");
SamplePortUnits$(1,"Volt"); ` Set up port 1
SamplePortFull(1,5);
SamplePortZero(1,0);
SamplePortName$(1,"ADC 1");
SampleKeyMark(1);      ` Enable keyboard markers
SampleLimitFrames(-1); ` Turn off all sampling limits
SampleLimitSize(-1);
SampleLimitTime(-1);
SamplePoints(10000);  ` 10000 points per sweep
SampleRate(1000);     ` sampled at 1 KHz
SampleTrigger(0);     ` No sweep trigger
SamplePause(1);       ` Pause at sweep end for script access
SampleWrite(0);       ` Don't write automatically to disk
```

The `PulseXXX` functions allow the script to control the pulse generation system. The functions allow pulses to be added, deleted, moved or modified. Here is an example of generating a simple dual-pulse protocol:

```
PulseClear(0,0);           ` Clear out all pulses for state 0, DAC 0
PulseDataSet(0,0,0,0);    ` Set initial level to zero
PulseAdd(0,0,1,"pulse1",0.05,0.01); ` Add first pulse at 50 ms
PulseDataSet(0,0,"pulse1",5); ` Set first pulse amplitude to 5 V
PulseAdd(0,0,1,"pulse2",0.1,0.01); ` Add second pulse at 100 ms
PulseDataSet(0,0,"pulse2",5); ` Set second pulse amplitude to 5 V
```

You may have wondered why all the function parameters start with two zeroes. The first argument selects the pulses for a given sampling state, zero selects the default pulse set. The second argument selects the 1401 output; 0 to 3 for DACs 0 to 3, 4 for the digital outputs. There is also a mechanism for naming pulses and accessing them by name or number, which may seem unduly complex. The reason for this is that, though the pulses are numbered and can be accessed by number, the numbering is in time order and can change when a pulse is added or moved. In addition the arbitrary waveform output, if used, is always attached to the DAC0 outputs which can confuse a script relying upon numbers. Therefore I recommend that, in non-trivial scripts, you set names for your pulses and use the names to access them.

The third argument in `PulseAdd()` is a code for the pulse type, codes include 1 for a square pulse, 2 for a square pulse with varying amplitude, 4 for a ramp and 7 for a pulse train. For each type of pulse, the meaning of the arguments in `PulseDataSet()` will vary appropriately.

The arbitrary waveform pulse item (code 6) is very useful for generating very short pulses or pulses with a customised shape or a shape taken from a waveform in a data file. For example, suppose you required square pulses that were only 10 or 20 microseconds long. This would require a sequencer resolution of 10 us, which is too fast, but using arbitrary waveform output at 100 KHz we can achieve the desired effect:

```
var wave%[10000];         ` An array to hold the waveform data

PulseClear(0,0);         ` Clear out all pulses for state 0, DAC 0
PulseDataSet(0,0,0,0);   ` Set initial level to zero
PulseAdd(0,0,6,"wave",0.01,0.01); ` Start waveform at 10 ms
PulseWaveSet(0,1,10000,10000); ` Set waveform to 10000 points at 100 KHz
ArrConst(wave%[], 0);    ` Initialise waveform to zero
wave%[1000] := 32767;     ` 10 us pulse 10 ms into waveform
wave%[2000] := 32767;    ` and 20 us pulse 20 ms in
wave%[2001] := 32767;
PulseWaveformSet(0,0,wave%[]); ` Load up the waveform data
```

This script sets up a waveform item, starting at 10 ms, which runs for 100 ms (10000 points at 100 KHz). Running at 100 KHz, each point in the array 'lasts' for 10 us. We then set up a 10000 point array to hold the waveform data that we wanted. Note that I used an integer array `wave%[]`. I could have used an array of floating-point data instead, in which case instead of setting up the DAC output values as the underlying -32768 to 32767 that is actually written to the DACs, I would have used calibrated values; -5 to +5 if the DAC is calibrated in volts, otherwise whatever data range is implied by the calibration. So if you are stimulating a cell membrane and your DAC is calibrated in mV at the cell membrane, you can set up your wave array with the millivolt values you actually want. Anyway I am a regenerate programmer and so I naturally use the raw binary values! Having

set up the array with the data values, it is straightforward to load the array into the pulse information using `PulseWaveformSet()`.

Another technique that I have found useful is to use the sampling configuration commands to interrogate (and modify if necessary) an existing sampling configuration so that I get a script that works with whatever sampling parameters have been set up previously. For example you can find out how many waveform channels are being sampled, and how many points there are in each channel:

```
nch% := SamplePorts();           ' Save points & channels for later
npt% := SamplePoints();
SampleWrite(0);                  ' Disable write, pause at sweep end
SamplePause(1);                  ' so that the script can control frames
```

Control of sampling

Another group of functions control sampling, or get information about the current state of sampling:

```
SampleAbort()      SampleAccept()      SamplePause()      SampleReset()
SampleStart()      SampleStop()         SampleSweep()      SampleWrite()
SampleHandle()     SampleKey()          SampleStatus()
```

Note that most of the sampling configuration commands are not useful during sampling; they modify the sampling configuration that will be used next time, but do not affect the sampling that is in progress. Exceptions to this are `SampleWrite()`, `SampleTrigger()`, `SamplePause()` and all the pulse configuration commands.

When running a script, the sampling control panel is normally hidden. The obvious way to control sampling is by calling sample control functions, usually from within the toolbar functions to allow some user control. It is generally unsatisfactory to show the sampling control panel and let the user control sampling as usual, because the script never knows when the user has pressed Start, or Restart, or Abort, all of which would require the script to make some adjustments or at least be aware of what has happened. A trivial example of a script that substitutes the toolbar for the sampling control panel is:

```
DoToolbar();                    'Call DoToolbar function
Halt;

Func DoToolbar()
ToolbarClear();                 'Remove any old buttons
ToolbarSet(1, "Quit", Quit%);   'Quits the script
ToolbarSet(2, "Start", Start%); 'Link to Start function
ToolbarSet(3, "Stop", Stop%);   'Link to Stop function
ToolbarSet(4, "Reset", Reset%); 'Link to Reset function
ToolbarSet(5, "Abort", Abort%); 'Link to Abort function
return Toolbar("Your prompt", 0);
end;

Func Quit%()                    ' Button 1 to quit script
return 0;                       ' Return zero to exit Toolbar()
end;

Func Start%()                   'Button 2 START SAMPLING routine
fv% := FileNew(0, 1);
SampleStart();
return 1;                        'This leaves toolbar active
end;
```

```

Func Stop%()
SampleStop();
return 1;
end;
'Button 3 STOP SAMPLING routine
'This leaves toolbar active

Func Reset%()
SampleReset();
return 1;
end;
'Button 4 RESET SAMPLING routine
'This leaves toolbar active

Func Abort%()
SampleAbort();
fv% := -1;
return 1;
end;
'Button 5 ABORT SAMPLING routine
' Flag we have no sampling view
'This leaves toolbar active

```

This is not particularly elegant, but the script does work. To tidy up the script, disable the toolbar buttons when they would have no affect. The script section above will enable 5 buttons, 4 of which would mimic the floating sampling control window and the 5th quits the script. To control writing of data to disk, it would be necessary to use the `SampleWrite()` function to switch writing to disk on and off. If this was a script written for student use we might not allow the student to terminate sampling without saving the data (just in case). We would therefore disable the reset and abort functions. Another aspect of using a toolbar, which we will cover below, is setting an idle function that is called repeatedly when nothing else is happening.

Many scripts use a button marked 'Disk Write On' or 'Disk Write Off' alternating between the two states depending on what the sample write status is at that time. This section of script allows the user to switch writing to disk on or off:

```

ToolbarSet(6, "Disk Write ??", record%);
'Button for access to the function
...
Func Record%()
if samplewrite()=0 then
SampleWrite(1);
ToolbarSet(6, "Disk Write On", record%);
else
SampleWrite(0);
ToolbarSet(6, "Disk Write Off", record%);
endif;
return 1;
end;
'Button 6 function
'if writing to disk is not on...
'...Switch disk write on and...
'...change the label
'If writing to disk is ON then switch off
'and change the label
'This leaves toolbar active

```

Monitoring sampling and manipulating data

While sampling is in progress, the script 'sees' a data file that has a frame zero which continually fills up with data. Depending upon the sampling settings, a full frame zero might just disappear and start to fill again, be written to the end of the file so that `FrameCount()` increases, or just sit there.

If the script wants to interact with the data in frame zero, it needs to get at the full frame zero before it disappears or is written to disk. The simplest way of doing this is to disable automatic writing to disk at the sweep end and to enable pausing at the end of a sweep:

```

SampleWrite(0);
SamplePause(1);

```

Once this is done, sampling will progress automatically until frame zero is full and then stop and wait until the script does something. This is quite important! It would be easy to write a script that let sampling free-run and (for example) watched `FrameCount()` until it reached some value, and then changed the pulses or the sampling state. This seems straightforward, but essentially you then have a race between the script execution and the automatic sampling process. This means that when the script sees `FrameCount()` reach 10, you cannot be sure that frame 11 has or has not started being sampled. You must set sampling to pause at the end of a sweep if you want the script to be able to control things or carry out actions on a frame-by-frame basis.

We can use the toolbar Idle function to monitor the progress of sampling and do what we want with new sweeps (for example customised artefact rejection):

```

ToolbarSet(0, "", Idle%);           ' Define an idle function

func Idle%()
var s%;
s% := SampleStatus();              ' Find out what is happening
if (s% = 3) then                    ' If we have a sweep of data
    if (code here to test the sweep data) then ' If we want this sweep
        SampleAccept(1);           ' Write this sweep to disk
    endif;
    SampleSweep();                  ' Either way start another sweep off
endif;
return 1;                            ' Return 1 to keep toolbar running
end;

```

This sort of interaction with sampling (a toolbar idle function that monitors sample status) is by far the simplest and most powerful way of giving a script control of sampling. As long as you remember to turn writing to disk off and pausing at sweep end on, your script will have complete control of sampling on a frame-by-frame basis. Using this technique it is possible to:

- Modify the data in frame zero before writing it to disk as in SOnline.
- Carry out script-based processing of the newly sampled data, with or without writing it to disk
- Implement custom artefact rejection mechanisms as in Artefact.
- Change what happens in the next sampling sweep (for example the pulses that are output) based upon the data sampled in this sweep.

Other methods of monitoring sampling

We have seen how the `SampleStatus()` function can be used to check for a sweep of data available in frame zero. In the example below, we use `FrameCount()` to determine how many sweeps of data have been written to the new data file and, if there are more new frames than some limit, carry out some processing task and update the 'last frame processed' value:

```

func idle%()
var f%;

view (rawData%);                    'make sampled data window current
f% := FrameCount();                  ' How many frames are in the file
If ((f% - last%) >= 10) then         'if we have >= 10 unprocessed sweeps
    view(av%).ProcessFrames(last%+1, last%+10, -1, 1, 1); ' Average 10 frames
    last% := last% + 10;             ' keep track of last frame done
Endif;
Return 1;                             'return to active toolbar
end;

```

Instead of simply using the `ProcessFrames()` function to provide an average of the last ten frames updated every ten frames (something that can be perfectly easily done using the built-in on-line processing system), the processing could consist of any code that worked through the new data, say averaging every third frame. When your interaction with the sampling is limited to working with frames already written to disk, it is not necessary to pause at the sweep end, sampling can be allowed to free-run.

The Yield() command

Any script which performs heavy, time consuming calculations should consider using the `Yield()` command. Added to Signal in version 2.11, this allows Windows to perform some background processing tasks which might otherwise not get a chance to run. In particular, if you are doing these calculations during sampling then sampling may fall behind if not given an opportunity to catch up.

Direct 1401 access

Version 4 of Signal allows direct access to the 1401 via a series of commands prefixed with `U1401`. For the most part you will not need to use these commands unless you have a specialist 1401 command you wish to use. For most people the most likely use would be to determine the type of the 1401 being used. This can be done using `U1401Open` as follows

```
var type%;           'Stores the 1401 type
type% := U1401Open(); 'Returns the 1401 type
U1401Close();       'We should close the 1401 again when not needed

if type% < 2 then
    Message("Gap free sampling is not available for old 1401 types.");
endif
```

The return values of `U1401Open` are as follows:

0	Standard 1401
1	<i>1401plus</i>
2	micro1401
3	Power1401
4	Micro1401 mk II
5	Power1401 mk II

Appendix

Appendix - DOEMG - a worked example

The DOEMG script was written as an example script that does something useful and that could be relatively easily modified by users to match their own requirements. I don't expect that we will have time to look at it during the training day but I have included it as an interesting worked example of a (hopefully) useful script.

Roughly speaking, the script provides two options; Sample and Analyse, via buttons in the main toolbar. A certain amount of care taken in maintaining view handles means that the script will analyse the most recently sampled file, or ask for a new file if there isn't one, and can take care of closing data files nicely. The analysis function is a relatively trivial loop based around `Interact()`, with results written to a text file. Sampling uses its own toolbar, toggles writing on and off using a toolbar button and monitors sampling in the idle routine so that it can maintain a display of rectified newly sampled data using a memory view.

```
.....  
' This script provides facilities for capture and analysis of emg data
```

First we define a few variables used throughout the script. `sv%` is the handle for the data file being sampled or analysed. `rv%` is the handle of the memory view displaying rectified data during sampling. `nch%` and `npt%` hold the number of waveform channels and points per frame for the data file being sampled or analysed. `wr%` is the 'write data to disk' flag.

```
var sv%, rv%;           ' The file view and memory view handles  
var nch%, npt%;        ' The number of channels and points  
var wr%;
```

This is the start of the script, which basically hides the script view, initialises a few variables, sets up the toolbar for use and runs the toolbar by calling `ToolBar()`. When this eventually returns because the user has pressed the Quit button on the toolbar the script window is un-hidden and the script stops. The `SetMainToolBar` function below just sets up the toolbar for use.

```
View(App(3)).WindowVisible(0);           ' Hide the script  
sv% := -1;                                ' Flag we have no views  
rv% := -1;  
SetMainToolBar();                         ' Set up the toolbar  
SetClose();                               ' Set up Close File toolbar button  
ToolBar("Sampling and analysis of EMG waveforms", 8+32); ' Do it  
View(App(3)).WindowVisible(1);           ' Show the script again  
halt;
```

```
' Sets up the toolbar for the sample / analysis / quit selection  
proc SetMainToolBar()  
ToolBarClear();  
ToolBarSet(4, "Close file", DoClose%);  
ToolBarSet(3, "Sample", DoSample%);  
ToolBarSet(2, "Analyse", DoAnalyse%);  
ToolBarSet(1, "Quit", Quit%);  
return;  
end;
```

The `DoClose%` function is called from the toolbar, which is why it always returns 1 to keep the script running.

```
' DoClose - close any active file  
func DoClose%()  
SaveFile%();                               ' Use this function to do the work
```

```
SetClose();           ' Get CloseFile button disabled
return 1;             ' Return 1 to keep script running
end;
```

The `DoAnalyse%` function does the entire analysis function - it is called from the toolbar. First of all it ensures we have a data file by testing `sv%` and loading a file if necessary. Then, after creating a text file for the results and a bit of setup, it falls into an `Interact()` loop where the user can position cursors and the software will take measurements and write them into the text file.

```
' DoAnalyse - go through the analysis process
func DoAnalyse%()
var cv%, tv%;
var choice%, some%, i%;
var a, st, en;

if (sv% < 0) then           ' If we havn't got a data file
  DoLoad%();               ' Try to load up a data file
  if (sv% < 0) then        ' If sv% is still negative
    return 1;              ' then just give up now
  endif;
endif;

' Here we have got our file sorted out, so we need to get ready for analysis.
' We need a pair of cursors for measurement, and a text view to log into, plus
' we need some information about the data file for general use.
some% := -1;                ' Flag that we havn't logged anything yet
nch% := View(sv%).ChanCount(1); ' Number of waveform channels
tv% := FileNew(1, 1);       ' Create a new text view
View(sv%).Window(0,0,100,50);
View(tv%).Window(0,50,100,100);
View(tv%).WindowVisible(1);
View(sv%).CursorSet(2);    ' Get a couple of vertical cursors
View(tv%).WindowTitle$("Measurements");
View(tv%).Print("Rectified area measurements\n\n");
View(tv%).Print("Frame Start      Width      Areas ....\n");

repeat
  View(sv%);
  choice% := Interact("Position the cursors, please", 2, 0, "Done", "Measure");
  if (choice% = 2) then
    cv% := View(sv%).SetCopy(-4, "Work", 1); ' Invisible copy view
    st := View(sv%).Cursor(1);               ' Time range for measurement
    en := View(sv%).Cursor(2);
    if (st > en) then
      st := en;
      en := View(sv%).Cursor(1);           ' Swap times if needed
    endif;
    View(tv%).Print("%4d   %8g   %8g   ", View(sv%).Frame(), st, en-st);
    for i% := 1 to nch% do
      abs(View(cv%,i%).[]);                ' rectify the data
      a := View(cv%).ChanMean(i%, st, en); ' Get mean value
      a := a * (en-st);                     ' Convert to area
      View(tv%).Print("%8g   ", a);
    next;
    View(tv%).Print("\n");                  ' Finish off line of logged text
    some% := 0;                             ' Flag that we have got some data
    View(cv%);
    FileClose(0, -1);                       ' Kill off the copy view
  endif
until choice% = 1;
View(tv%);                                  ' Close the text view, some% will try
FileClose(0, some%);                        ' to save if we logged something
View(sv%).Window(0,0,100,100);              ' File view again fills the screen
SetClose();                                 ' Set up Close File
'ToolbarText("Sampling and analysis of EMG waveforms");
return 1;                                   ' Return 1 to keep script running
end;
```

The `DoSample%` function does the entire analysis function - it is called from the toolbar. First of all it ensures we are clear for sampling by testing `sv%` and saving & closing the current file if necessary. Then, after setting up sampling parameters and a new toolbar, it falls into a `ToolBar()` loop where the idle routine monitors the incoming data and displays new sweeps (rectified) in the memory view.

```
' DoSample - go through the sampling process
func DoSample%()
if (SaveFile%() > 0) then           ' Check we are OK for another file
    return 1;                       ' Give up if we are not
endif;
SetParameters();                   ' Tidy up sampling parameters
SetSampToolBar();                   ' Set up toolbar as we require
Start%();                           ' Start off the sampling
ToolBar("Sampling of EMG waveforms", 0); ' Give control to the user
SetMainToolBar();                   ' Sampling done, restore the toolbar
SetClose();                          ' Set up Close File
ToolBarText("Sampling and analysis of EMG waveforms"); ' Tidy up text
return 1;
end;

' Set up the sampling parameters. Mostly we use whatever parameters are currently
' loaded.
proc SetParameters()
SampleLimitFrames(0);               ' No frame limit
SampleWrite(0);                      ' Sweeps not written to disk
wr% := 0;                             ' Flag we are not writing
SamplePause(1);                      ' Pause after each sweep
return;
end;

' Set up the toolbar for the sampling process
proc SetSampToolBar()
ToolBarClear();
ToolBarSet(0, "", Idle%);
ToolBarSet(3, "Save data", Write%);
ToolBarSet(2, "Stop", Stop%);
ToolBarSet(1, "Abort", Abort%);
return;
end;
```

The `Idle%` routine checks to see if there is a new sweep of data available. If so, it copies the data into the result view using `ArrConst()`, rectifies it and displays the rectified data. Finally a new sweep is initiated. The `write%` function toggles the disk write state.

```
' Idle does most of the work. If there is a new sweep available we copy
' the data into the result view, rectify the data, and display it.
func Idle%()
var s%, i%, v;

s% := SampleStatus();
if (s% > 0) then
    if (s% = 3) then                 ' If we have got a sweep available
        for i% := 1 to nch% do
            ArrConst(view(rv%,i%).[], view(sv%,i%).[]);
            Abs(view(rv%,i%).[]);
            view(rv%).Optimise(i%);
            view(rv%).YRange(i%, 0, view(rv%).yHigh(i%));
        next;
        view(rv%).Draw();
        SampleSweep();               ' Enable a new sweep
    endif;
endif;
return 1;
end;

' Write - toggle the write state and update the toolbar button
```

```

func Write%()
if (wr% = 0) then
  wr% := 1;
  ToolbarSet(3, "Discard data", Write%);
else
  wr% := 0;
  ToolbarSet(3, "Save data", Write%);
endif;
SampleWrite(wr%);
return 1;
end;

```

start% is called from the sampling toolbar and creates both the new file view and the result view used to display the rectified data. It then positions the views nicely and kicks sampling off.

```

' Start - create the file and memory views, initialise the views
'           and position them on the screen and initiate the
'           sampling process.
func Start%()
var er$, sh%;
var sp, st;

sv% := FileNew(0, 0);           ' Create a file view for sampling
if (sv% < 0) then               ' and check for errors
  er$ := Error$(sv%);
  Message("File new failed :\n\n%s", er$);
  sv% := -1;                    ' Flag we have no views
  return 0;
endif;
sh% := SampleHandle(0);
if (sh% <> sv%) then
  Message("Handle %d should be %d\n", sh%, sv%);
  return 0;
endif;
nch% := SamplePorts();         ' Save points & channels for later
npt% := SamplePoints();
rv% := SetCopy(-4, "Rectified EMG", 0);
if (rv% < 0) then               ' again check for errors
  er$ := Error$(rv%);
  Message("Result create failed:\n\n%s", er$);
  View(sv%).FileClose(0,-1);
  sv% := -1;                    ' Flag we have no views
  rv% := -1;
  return 0;
endif;
View(sv%).WindowTitle$("Raw EMG"); ' Change file view name
View(sv%).Frame(0);           ' Fix file view on frame 0
View(sv%).Window(0,0,100,50); ' File view in top half of screen
View(rv%).Window(0,50,100,100); ' and rectified data in the bottom half
View(sv%).WindowVisible(1);  ' Allow us to see the views
View(rv%).WindowVisible(1);
SampleStart();                 ' and start sampling
FrontView(sv%);
return 1;
end;

```

stop% kills off sampling and then either displays the new data file or, if no frames were written to disk, deletes it.

```

' Stop - stop the sampling, kill off the memory view and leave the
'           file view filling the screen.
func Stop%()
SampleStop();                 ' Stop all sampling
View(rv%);
FileClose(0,-1);             ' Kill off the memory view
rv% := -1;                    ' Flag we have no memory view
View(sv%);                    ' Switch to our new data file
if (FrameCount() > 0) then    ' Have we got any data

```

Appendix - DOEMG - a worked example

```
    View(sv%).Window(0,0,100,100);          ' File view now fills the screen
else
    FileClose(1, -1);                      ' Kill off the useless file
    sv% := -1;
endif
return 0;                                  ' Return 0 to exit from sampling toolbar
end;

' Abort - abort sampling, which will kill off both views
func Abort%()
SampleAbort();
View(rv%);
FileClose(0,-1);                          ' Kill off the memory view
sv% := -1;                                 ' Flag we have no file view
rv% := -1;                                 ' and no memory view
return 0;
end;

' Quit - return zero to terminate ToolBar function
func Quit%()
return 0;
end;
```

The utility routines are fairly straightforward. DoLoad takes care of loading in a new file (saving & closing the previous file if necessary). SaveFile takes care of closing & saving a data file that is in use.

```
.....
' Utility routines start here
.....

' DoLoad - load in a new data file for use in analysis
func DoLoad%()
if (SaveFile%() > 0) then                  ' Check we are OK for another file
    return 1;                             ' Give up if we are not
endif;
sv% := FileOpen("*.cfs", 0, 3);          ' Open a new data file
if (sv% >= 0) then
    View(sv%).Window(0,0,100,100);        ' File view now fills the screen
endif;
return 1;                                 ' Return 1 to keep script running
end;

' SaveFile - save previous data so that we can load or
'           sample a new data file. Sets sv% -ve if all
'           is OK, also returns zero.
func SaveFile%()
if (sv% >= 0) then                        ' Have we got a current file?
    view(sv%);
    if (FileClose(1, 0) > 0) then        ' Close and save the previous file
        return 1;                       ' If user cancelled then quit
    endif;
endif;
sv% := -1;                                ' Flag that we have no file loaded
return 0;
end;

' Gets the toolbar CloseFile item enabled\disabled
func SetClose()
if (sv% >= 0) then
    ToolbarEnable(4, 1);                 ' Enable Close File if we have a file
else
    ToolbarEnable(4, 0);                 ' Disable Close File as no file
endif
end;
```