

Spike2

Training Course Manual

November 2013

Copyright © Cambridge Electronic Design Limited 1996-2013

The information in this book was prepared and collated by the staff of Cambridge Electronic Design Limited as teaching material for the Spike2 Training Days held in Cambridge and worldwide between 1996 and 2011. It has been updated to take account of new features in Spike2 versions 3, 4, 5, 6, 7 and 8.

First version	July 1996
Revised	September 1997
Revised	January 1998
Revised	October 2000
Revised	January 2001
Revised	October 2002
Revised	October 2004
Revised	June 2006
Revised	April 2007
Revised	May 2009
Revised	May 2011
Revised	November 2013

Published by:

Cambridge Electronic Design Limited
Science Park
Milton Road
Cambridge
CB4 4FE
UK

Telephone: Cambridge +44 (0)1223 420186
Fax: Cambridge +44 (0)1223 420488
Email: info@ced.co.uk
Web site: <http://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.

Spike2 Training Days

Introduction This book is a compendium of some of the material presented at the Spike2 Training Days held in Cambridge, England and around the world. The material here, especially in the scripting sections, expands on the information in the standard Spike2 manuals. We hope that you find this book useful and instructive.

If you are reading this book after attending the Training Days, it should remind you of the topics covered and give you some extra reading; it was impossible to cover everything in this book in the time available. If you are reading this independently of the Training Days, then if you follow it through in the order of the chapters, you will get a fairly detailed account of the Spike2 program.

Some sections of this book rely on material covered in other chapters. For example, the *Using scripts on-line* chapter assumes that the reader has followed the *Script toolbox* chapter, and this, in turn assumes knowledge of the *Script Introduction* chapter.

For the most part, this book covers material that is in common for all versions of Spike2 from 2 to 8. Occasionally, there are sections that refer to features that are not in all versions. These are indicated in the text and users of older versions should skip them (or consider if an upgrade would be a useful investment!) The screen images from a range of versions of Spike2. You should consult the documentation for your version of Spike2 for the details of your particular version.

Example files The output sequencer and script chapters have many examples. To save you a lot of typing these examples are available on disk. These examples may not be identical to the listings as some of the comments were shortened to fit in with the format of this manual. Users of Spike2 version 3 and later will find that the examples are copied into folders in the TrainDay folder in the Spike2 installation folder. The folders are:

Control	Examples from the <i>Sampling, control and the output sequencer</i> chapter
Script1	Examples from the <i>Script Introduction</i> chapter
Script2	Examples from the <i>Script toolbox</i> chapter
Spk2Data	Examples from the <i>Scripts and Spike2 data</i> chapter
Memchan	Examples from the <i>Advanced topics</i> chapter
Online	Examples from the <i>Using scripts on-line</i> chapter

Table of Contents	Introduction to Spike2.....	2
	Sampling data.....	15
	The graphical output sequence editor.....	28
	Sampling, control and the output sequencer	36
	Spike shapes.....	47
	Script introduction.....	56
	Script toolbox.....	66
	Scripts and Spike2 data.....	74
	Using scripts on-line	93
	Advanced topics.....	108
	Sonogram display mode.....	127

Introduction to Spike2

Introduction to Spike2 The purpose of this chapter is to give you an overview of what Spike2 is, what it can do, how to get started with it and where to find items in the menus. Armed with this knowledge you should be able to read through the more specialised topics in the remainder of this book and see how they fit into the generalised framework presented here.

What is Spike2? Spike2 is different things to different people. To some it is a simple data capture and review program, acting as a computerised chart recorder. 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. Some people use it to produce complex stimulus patterns; others have no interest in the output capabilities.

We assume that you want to use Spike2 to solve a problem, so you already have some idea of what the program is going to be for you. We hope that the contents of this book will show you enough of the capabilities of the program so that you can decide how much (or how little) of it you must use in your work.

Installation Detailed installation instructions for each system are given in the manuals supplied with your copy of Spike2. The following is a summary of installation:

Windows The software is easy to install and guides you through the process of installation. The installation program will suggest a folder called `SpikeN` for version N, for example `Spike8` for version 8. If you are installing to a system that already has a previous version, do not install the newer version on top of the old one unless you never want to run the old one again. Most users have no problems installing the software but if you do, don't panic; you can always contact CED and we will then talk you through the procedure.

The installation program will also install 1401 support, if you wish. This means that if you are setting up a new system with a 1401 and Spike2, you only need use the Spike2 installation disk. The installation disk that comes with the 1401 does include extra programs, but you would only need them if you were having hardware problems. If you install 1401 support from the Spike2 disk, it is possible that you may be asked to restart your computer to make sure that the most up-to-date device driver is used.

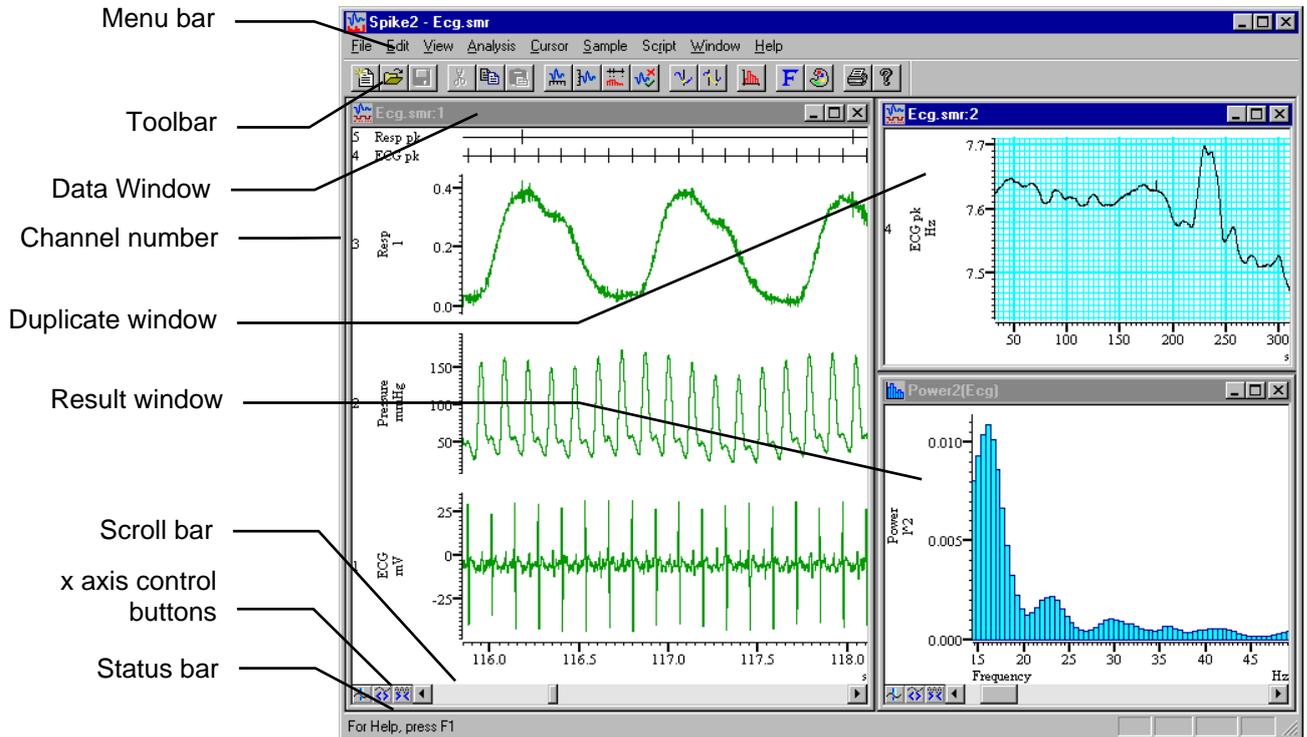
Once you have installed the Spike2 and 1401 driver software you can run Spike2 by either clicking on Spike2 within the Spike2 program group or if you are running Windows 95 or later, you can use the Start menu to locate Programs and then Spike2.

Multi-Tasking Spike2 data acquisition is a 'real time' process and requires the host computer to react on demand to a call for resources. It is therefore inadvisable to run other applications when sampling data as this can cause the sampling to fail if another application does not release required system resources in a reasonable time. Many applications do run happily at the same time as Spike2, just remember, you have been warned. From Spike2 version 5, the time-critical parts of the data acquisition run in a separate thread with a higher priority, so it is more robust when competing with other programs for system time.

From version 5 there is also a Scheduler tab in the Edit menu Preferences that gives you some control over how much CPU time Spike2 uses.

Basic screen layout

When you launch the application you will see a blank screen except for the menu (and toolbar and status bar in Windows). 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 gives you short cuts to Menu bar commands. The Status bar gives you feedback (for example it gives the function of any toolbar button under the mouse pointer).

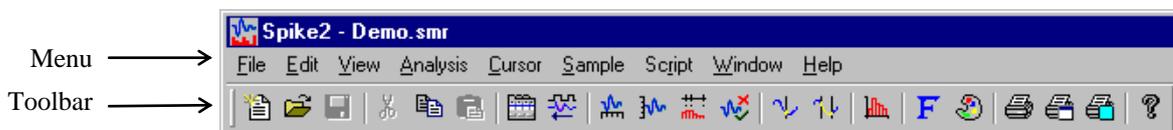
The Data window shows unprocessed sampled data. We refer to such a window in Spike2 as a “Time view” because the window shows how the data changes with time. You can have multiple channels of data in a time view. Each channel has an associated Channel number that uniquely identifies the channel (this number is used by the script system).

The Result window shows a single array of data that is usually the result of an analysis of data from a time view. In the Spike2 manuals we call this a “Result view”.

At the bottom of each time and result view is a Scroll bar and the x axis control buttons. You use these to choose the section of data that is displayed in the window. You can hide the x axis and the controls from the View menu Show/Hide channels command.

A Duplicate window is a copy of an existing time window created with the Window menu Duplicate command. You can duplicate a window several times if you wish. A duplicate lets you display the same data in a different format and display different areas of a file simultaneously.

Tour of the menu bar The menu bar holds the list of commands available in the application. From the menu bar you can sample data, open files and analyse, display and manipulate the data within them.



The illustration above shows the menu bar and its associated toolbar (icons). The toolbar is a short cut to the more common functions available from the menus. It can be hidden (from the View menu) to give more room for data and result views.

Dimmed icons in the toolbar correspond to functions that do not operate on the active window.

Spike2 versions 3 onward also have edit and debug toolbars and you can show and hide toolbars by clicking the right mouse button on a vacant area of a toolbar.

File	From here you can load, save and print documents and create new ones. This menu also lets you save and restore the system configuration. Under Windows you also get a recently used file list, and if you are attached to a suitable Mail server you can send the current document as mail.
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 command which sets where new data files are stored until you save or delete them.
View	View manipulation commands which control the “look” of the data windows; many of these commands can be duplicated by clicking with the mouse on a window. You control the font and colour of the active window from here.
Analysis	The main built in analysis functions live here. They range from averages and stimulus histograms through to deriving new channels based on peak and trough detection. From version 4, this also holds commands to make measurements and create XY trend plots based on cursor positions and do FIR digital filtering. Version 5 adds commands to create virtual channels through channel expressions. Version 6 extends the virtual channels and adds IIR digital filtering.
Cursor	Cursor values, areas, means and slopes can be accessed here. From version 4 the controls for the active cursors can be found here.
Sample	Setting up of the sampling configuration is done here. The menu also duplicates the sampling control box and access to the template parameters. The “Create TextMark” function is also here, allowing the user to ‘stick’ a short note on to a data file whilst sampling. You can also configure the CED 1401-18 discriminator and serial-line controlled CED1902 and CyberAmp signal conditioners from here for systems that support them.
Script	Use this to evaluate one line of script, record your actions and run a script. You are allowed to load as many scripts as you can fit in memory in the File menu. From here you can select one to run.
Window	This contains commands for laying out the screen area and hiding or showing the views.
Help	One method of accessing the help index. This also includes information about the version of Spike2 that you are using.

Using Help

Spike2 has a comprehensive help system installed alongside the main software. You can use the help icon  in some documents to access it directly or use the context sensitive help from the script and evaluate windows. Place the cursor somewhere in the command and press F1. If Spike2 recognises the text surrounding the cursor it will jump straight to the help section related to the text. Pressing F1 whilst a dialog is active will access the appropriate help page for that dialog.

The on-line Help includes an easy to use index; full installations of Spike2 also include PDF files of much the same information organised as a printable manual.

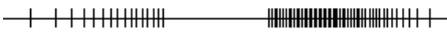
Document types

Spike2 can open and display several document types. They are opened from the File pull down menu. Files of different types are distinguished by the file extension. The file extension is a period followed by from one to three characters at the end of the file name.

Type	Extension	Purpose
Data	.smr .smrx	.smr files are used by all version of Spike2 to hold sampled data collected using Spike2 and a 1401 These files hold 32-bit times, and are up to 1 TB in size. Spike2 version 8 introduces .smrx files that use 64-bit times and that are of virtually unlimited size.
Result	.srf	Once data has been analysed and a result view has been created then the view can be saved in this format. This allows the user to reload the file for further analysis or print it for inclusion in a paper.
Text	.txt	The standard text file can be useful for storage of results and notes or for exporting data to other applications.
Sequence	.pls	The pulse output file. It is often necessary to control stimuli whilst sampling and this file type is used for that purpose. The file is a set of output commands that can be initiated via the mouse, keyboard or from an on-line script.
Script	.s2s	This is the Spike2 macro (script) file type. It enables the user to write or record a set of actions which may include further customised analysis. Repetitive or complex tasks can then be performed as often as the user wishes. With Spike2 for Windows you can make a script run when the program loads. This can be used to run an on-line script to control data sampling and analysis with little user intervention.
Resource	.s2r .s2rx	These are resource files that holds information such as screen arrangements of data windows. The .s2rx format was introduced in version 7.11 and is the preferred format.
XY	.sxy	Available from Spike2 version 3. XY view disk format. The same format is used by Signal version 1.5 and later.
Sampling setup	.s2c .s2cx	This file type holds the sampling configuration as displayed by the Sampling Configuration dialog. The .s2cx format was introduced in version 7.11 and is the preferred format.
Filter bank	.cfb .cfbx	The file <code>filterbank.cfb</code> holds digital filters that you edit and use from the Analysis menu Digital filter command. The .cfbx format was introduced in version 7.11 and is the preferred format.
Video	.avi	Spike2 version 5 onwards can display multimedia files linked to data files. As you scroll through the time view, the linked multi-media files also scroll to the same position.

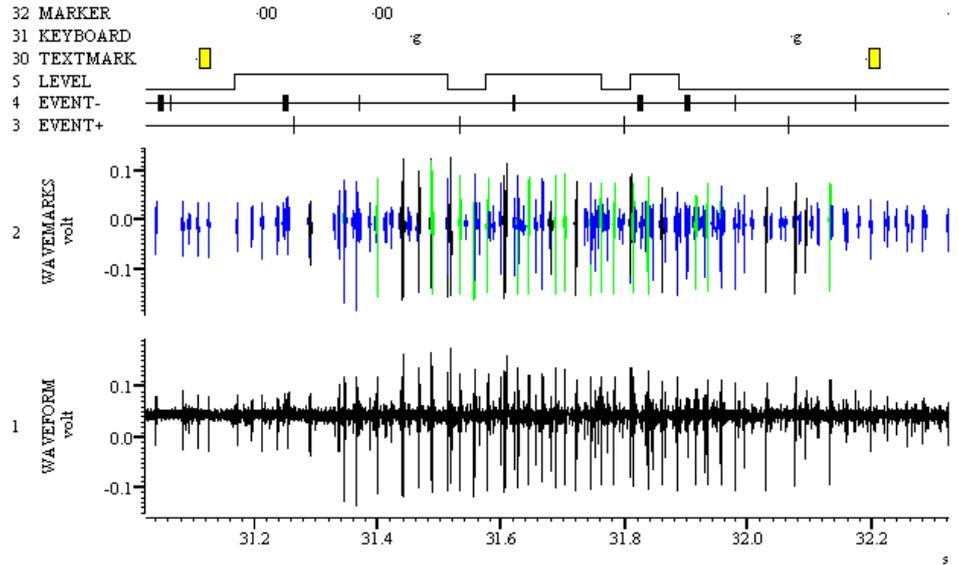
Concept of channels and channel types

These are the different channel types that Spike2 can sample and derive from other channels. These channel types can only exist in a time view.

Waveform	Waveforms are stored as a list  of data points. The points are equally spaced in time (but we also allow gaps in the waveforms). The data is stored on disk as 16-bit integers (range -32768 to 32767), but they are presented to the user as real numbers in user units. Each channel has an associated scale and offset that converts between the integer values on disk (and in memory) and the values the user sees.
RealWave	This is very similar to a Waveform, except that the data is stored as 32-bit floating point numbers. This channel type can be used as if it were a 16-bit Waveform. Each channel has an associated scale and offset that are used to convert the data to 16-bit integer if this is required.
WaveMark	This type of data is a short fragment of a waveform  plus four codes (in the range 0-255) that label the type of the data. This data type is often used to store discriminated spike shapes. From version 4.10 onwards, WaveMark data can have multiple traces, making this type suitable for recording Stereotrode and Tetrode data.
Event+	This data type is a time marker.  The time is taken from the rising edge of a TTL compatible input signal. This type of data is used to store externally discriminated spikes and general event times. There is also Event- data, which is identical, except the falling edge of a TTL compatible signal is used.
Level	A level channel is like the combination  of Event+ and Event- data. It records both the rising and falling edges of data. Unless you really need to record both edges it is far more efficient to use Event+ or Event- data types.
TextMark	A TextMark is a short text note linked to a time in the file.  Like the WaveMark type, it has four codes in the range 0-255 that identify types of TextMark data. During sampling you can add these manually or automatically log text strings from a serial line input.
Keyboard	A Keyboard marker is a time stamp. It has 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 time window is active, the key value and the time are saved in the file.
Marker	The digital marker channel is stored in the same way as the keyboard marker. The difference is that the first of the four codes holds the digital data read by the 1401 when a trigger line was pulled low. This type of channel is used to collect information from other digital equipment during data capture.
RealMark	A realmark channel is a marker channel with the added ability to store a list of real numbers. It can be created using the active cursor measurement process or by the Spike2 script language.
Memory	A memory channel can be of any of the above types. It is a channel that is created either by analysis functions or by the Spike2 script language.
Virtual	Virtual channels contain RealWave data derived by a user-supplied expression. The derived data can be based on existing data channels or functions such as sine, square or triangle waves.

A simple data file might contain a channel of waveform data (voltage information) and a keyboard marker channel containing key codes indicating various stages in the experiment. A rather more complex file might contain a number of waveform channels for eye movement and blood pressure along side a number of multi-unit WaveMark channels and digital input/output. This would enable Spike2 to keep track of (and perhaps produce) the stimuli for the subject.

Example data file

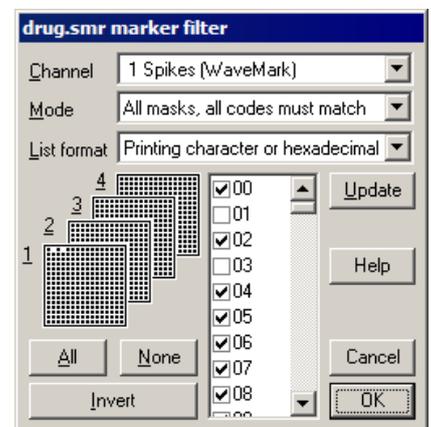


Marker data and marker filter

The marker filter selects data from a keyboard marker, digital marker, TextMark, RealMark or WaveMark channel based on the four codes stored with each data item. We will henceforth refer to all such data channels as “marker” channels. In terms of the keyboard channel, it might be necessary to filter a particular key or keys that indicate a drug dose is being applied. This allows you to scan quickly through the data file identifying regions of interest. When using a script, the marker filter can help to jump directly to an area marked by a character or code and present a pre-determined time around that point. It would then be the users decision whether it was a valid region for analysis.

For multi-unit recording the marker filter is invaluable. It enables the user to extract different units from the channel and perform analysis on just one type or cross analysis between several types of unit both on-line and off-line.

The view to the right shows the marker filter. At the top, you can see which channel the masking will affect. The Mode field is not present in version 2; the picture shows it set to be compatible with version 2. The vertical bar indicates the code(s) that will be masked; in this case 01 and 03 would be hidden. There are in fact 4 code “levels” each with 256 different values, although you tend to only use the first level for most applications.

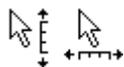


The small squares to the left of the scrollable region give you a visible indication of the codes that are included (black dot) or excluded (white space) for each of the four levels of codes. For a data item to be included, each of the four codes must be included in the corresponding level mask. As most people only need to use level 1 (the first code), they usually leave the filter for the remaining three levels set to accept all codes.

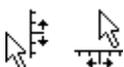
The List format field was added at version 5 to control the display format of the scrolling list of codes.

Control of time views You can scroll through the data and result views using the mouse to move rapidly to interesting areas. There are also functions to optimise channel displays and to duplicate views so you can view different time ranges simultaneously.

Double-click the X or Y axis to type in a display range. Drag functions also allow you to zoom the channel by clicking and holding down the mouse button on the data areas of the screen. Dragging the new pointer  in any direction will zoom into the data area. Zooming out requires the Ctrl key (Option key on the Macintosh) to be pressed while dragging and the pointer will appear with a negative symbol .



From version 4, moving the pointer over the X or Y axis ticks allows you to click and drag the axis range displayed, with the scale remaining constant. The window will update when the mouse button is released. If Ctrl is held down whilst doing this the window will update continuously



Moving the pointer beyond the ticks to the number area allows you to click and drag to change the axis scaling in the same way. If the zero point is visible, the scaling is done around this, with the zero point remaining fixed. If not visible the fixed point is the middle of the axis.

Quick measurements Pressing the Alt key then clicking and holding down the left mouse button will change the pointer to display the current mouse position in X and Y axis units. Subsequently dragging the pointer within the channel area will update this with the X and/or Y axis difference from the start position to the current one. While the mouse button is depressed (the Alt key can be released at this point) pressing C or L on the keyboard will copy the current values to the clipboard or write them to the log window.

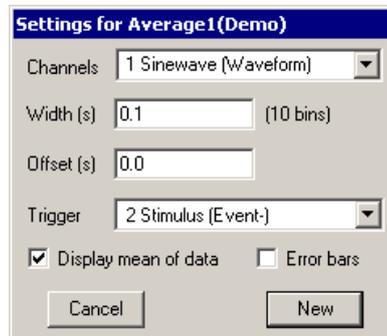
Use of cursors You can add up to 10 vertical cursors to a data or result view with the cursor icon  or the cursor pull down menu. This then allows you to take quick visual measurements with the cursor label or by using the Display Y Values and Cursor Regions again from the cursor pull down you can measure slopes, means and areas for each analogue channel. Cursors also allow us to obtain a count of events between cursors.

Vertical cursors are numbered 0 to 9. Pressing Ctrl + n (where n is a number key between 0 and 9) will fetch or create that numbered vertical cursor within the visible data area.

Built in analysis

The view to the right is the result view menu originating from the analysis pull down. You can select any of these on or off-line but if you are sampling at high rates, the priority must be to get data safely to disk so analysis must become a secondary consideration. However, if you are using a modern, fast computer, this is not usually a problem, especially if you use a PCI or USB interface to the 1401.

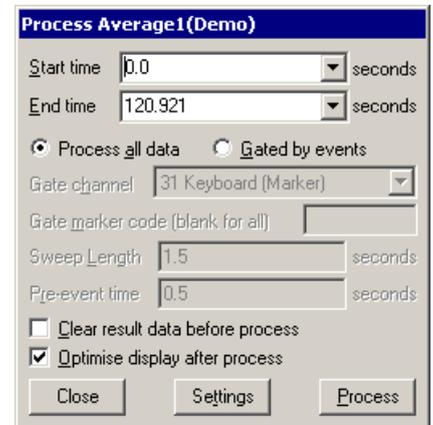
Interval Histogram...
Waveform Average...
Stimulus Histogram...
Event Correlation...
Phase Histogram...
Power Spectrum...
Waveform Correlation...



This is a typical example of the first stage of the analysis setup both on and off-line. You choose the channels to analyse in the first field at the top of this dialog. The next field sets the width of the result view; the dialog also displays this as the number of bins in the result. For a waveform average this is the number of analogue data points to use.

When making histograms from event channels there is also a *Bin width* field. It is not present for a waveform as the bin width is set by the interval between waveform samples. The time width of the histogram is the number of bins multiplied by the bin width. The **Offset** field sets the pre-trigger time to display in the result.

Off-line, once you have clicked the OK button you are presented with this new dialog. Here you set the area of the data file to include in the analysis sweep. You can give start and end times in seconds or if cursors are present these can be used to determine the area to use. If **Clear result view before process** is unchecked as shown then the new result will be added to the old. Version 5 allows you to restrict the analysis to time ranges that surround event markers; you might use this if you used markers to indicate different treatments.



Simple sampling

The well-known Nyquist theorem states that waveform data should be sampled at a rate that is at least twice the bandwidth of the signal. If you don't sample fast enough, frequencies in the input data above half the sampling rate appear to be below half the sampling rate in the sampled data. Most users sample quite a bit faster than the absolute minimum rate so that high speed signals don't look jagged. A rule of thumb for the absolute minimum sample rate you should use is 2.5 times the maximum frequency in the signal. This would produce a reasonable 'image' of the original signal in digitised form. If in doubt, sample the incoming data at a high rate and then perform a power spectrum upon it. This will identify the maximum frequency present.

Maximum rates of continuous acquisition to disk range from 80 kHz (with a standard 1401) up to the maximum ADC rate supported by your 1401. However, the maximum rate is also dependant on the 1401 type, connection (USB 1, USB 2, PCI), speed of the computer system and disk in use. Although you can buffer information inside the 1401 (up to 1 GB with a Power1401), you do have to get rid of it as the memory is filled. If you have a choice of interface for the 1401, a Power 1401 or Micro1401 mk II or 3 with USB 2 are the fastest, the PCI card is the next fastest, USB 1 is the slowest.

Don't sample a waveform such as respiration at 1000 Hz when you can get a good image of the signal at only 10 to 20 Hz. The space used by a channel soon mounts up. For example:

data sampled at 500Hz (say ECG) = 1000 bytes/sec which equates to 60,000 bytes/minute and therefore 3.6MB/Hour and that's just one channel.

Use of excessively high sampling rates wastes disk space and increases the time required for analysis. It also makes the data files inconveniently large when backing them up or archiving them. If you sample at a rate that is at least 2.5 times the highest frequency present in your data, Spike2 can reconstruct the data in the gaps between the samples.

Backing up your data

With very large data files, it is prudent to store data on a backup media. Although the probability of a disk error is small, with files of many megabytes, the chances are that if there is a disk error, it will be in one of your large files. It is also important to remember that hard disks are not 100% reliable. They are mechanical with precision components that are sensitive to mechanical shock and wear. As hard disks age they become more likely to fail or to develop hard (unrecoverable) and soft (recoverable) errors.

The type of backup media can include:

- Removable hard disks
- Writeable CD ROMs
- Writeable DVDs
- Magnetic tape
- Network backup drives

It is usually not a good idea to sample directly to network drives, or to work with files across a network. Network traffic can lead to unpredictable access times.

Setting up Spike2 to sample

From the **Sample** menu you can open the sampling configuration dialog. Here you set up the number and types of channels to use together with their independent sampling rates.

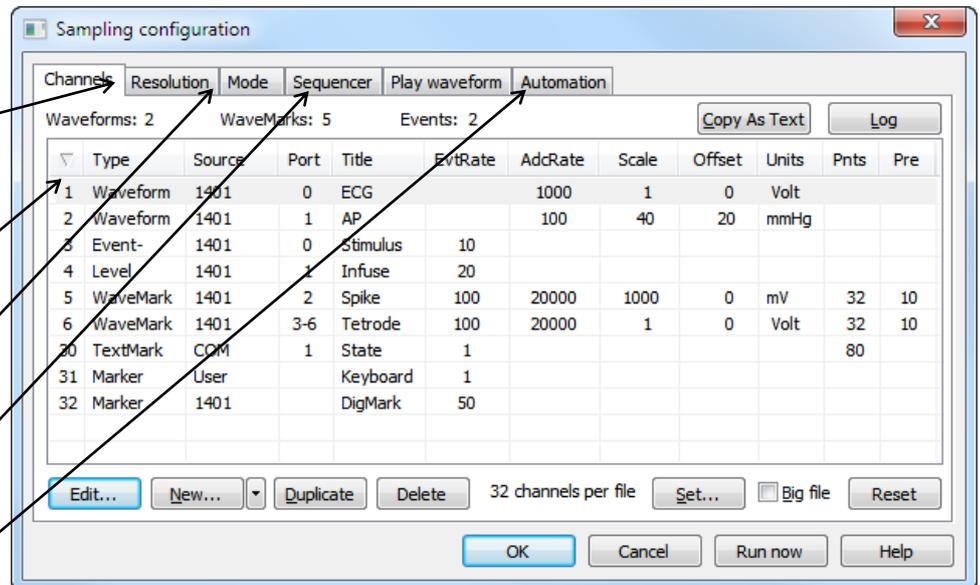
This tab sets the time resolution, which sets the maximum run time and the 1401 type

These are the channels to sample, their sample rates, titles, comments and waveform scaling

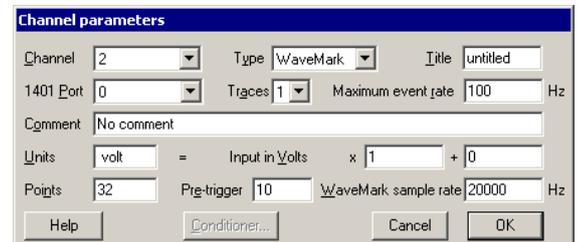
This sets the sampling mode (continuous, timed, triggered)

This sets a sequencer file to run during sampling

Maximum file size and sample time and automatic file naming



Double click on a channel or click Edit on the highlighted channel to access a further configuration menu. Use the New or Duplicate buttons to add a channel. The WaveMark channel configuration to the right is a combination of event and waveform data.



Channel The channel that is being edited, or the channel number you wish to set.

Type The type of input you wish to capture.

Title The title for the channel (this is displayed in the time view).

1401 Port Which physical input port on the 1401 this channel will receive its data from.

Maximum sustained event rate The mean number of events that you expect per second. The setting here dictates the size of memory allocated to the channel inside the 1401. The maximum performance will be increased if the rate is set realistically.

Ideal ADC sampling rate In place of Maximum sustained event rate for a waveform channel. This sets the desired analogue to digital rate for the particular channel. It is necessary to check that the resolution is set high enough to achieve the desired sample speed.

Comment The channel comment

Units Units to be displayed on the y axis (mm Hg, Litres etc.)

Input in volts The scaling factor e.g. 1 volt = 10 mm Hg + offset

Points For WaveMark channels; the number of data points per event (from 6 to 126)

Pre-trigger For WaveMark channels; the number of pre-trigger points to capture.

Traces For WaveMark channels in version 5; the number of interleaved traces for each spike.

TextMark channels have additional options that allow you to record text strings from a serial line as part of the data file.

Saving the Default Configuration

Once you have checked that the sampling works correctly, you can save it as your default sampling configuration in place of the standard 100 Hz continuous sampling version supplied with Spike2. You can also modify the colours (and the application window position in Windows). This will then be the new configuration, which will load every time Spike2 is executed.

To set a default configuration hold down the control key, open the **File** menu and use the **Save Default Configuration** command. You can also write configuration files with your choice of name if you use the **File** menu without holding down the control key and select the **Save Configuration** command.

Moving data to the clipboard

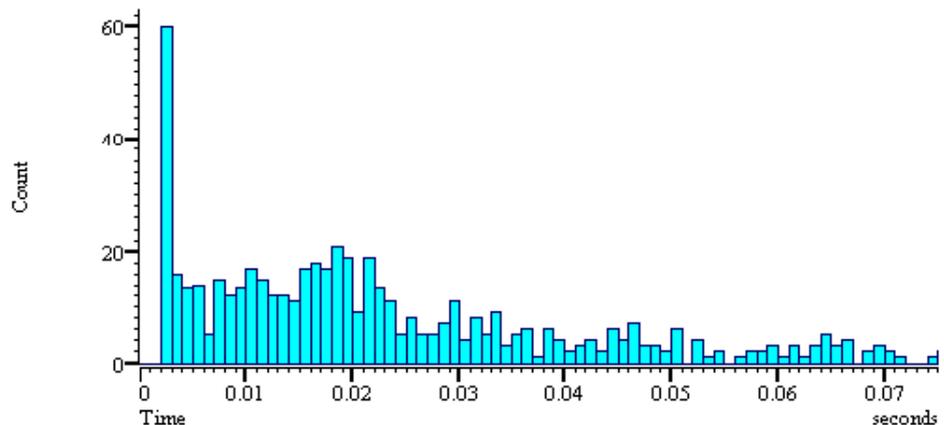
Using the **Edit** pull down you can access the **Copy** and **Copy as Text** function to place data into the clipboard. This is a holding area for data that will be *pasted* into other applications, such as a word processing package or spreadsheet.

Exporting graphics and text

Using the results acquired from the built in analysis section or from a script, you can export data as graphics and text to another application, in the case of this document, Microsoft Word. An example can be seen below. All of the following were produced using **Copy** or **Copy as text**.

Bitmap images

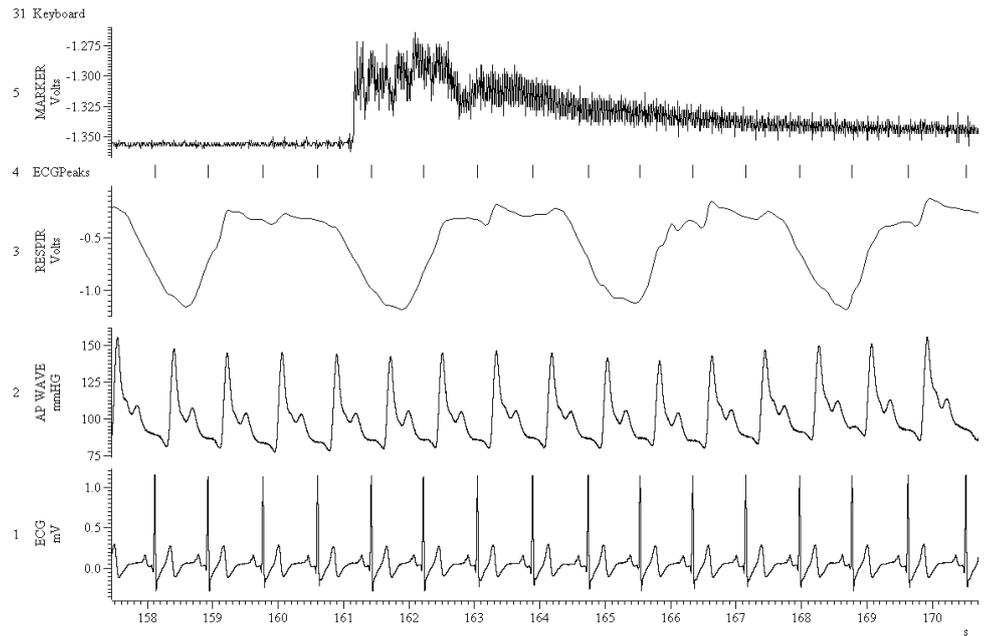
The image below was pasted into this Word document using **Copy** in Spike2, then **Paste Special** in Word, and selecting a bitmap format. Bitmaps are an exact copy of the image on the screen and are limited to the screen resolution, even when displayed on a device with a much higher resolution (such as a printer). In particular, they suffer when scaled and lines that are not horizontal or vertical look jagged when printed.



Vector images

The next image was produced using **Copy** and then **Paste** into this Word document. In this case, this is a vector image, and can be scaled to suit the document without losing resolution. When you print this 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. However, beware that many drawing programs allow a rather limited number of vectors. A Spike2 data file can have many millions data points, and although we restrict the number of lines we draw this can still break some drawing programs.



From Spike2 version 3 onward, you can adjust the picture resolution from the **Edit menu Preferences** option. If you have problems with importing time or result views into drawing programs, lowering the resolution may solve the problem.

Text output I used Copy as Text to produce this output of the raw waveform data as text.

```
"INFORMATION"
"Coldresp.smr"
""

"SUMMARY"
"1" "Waveform" "ECG" " volt" 250 250 1 0
"2" "Waveform" "Rate" " volt" 50 50 1 0
"3" "Waveform" "Volume" " Lit" 50 50 0.5 0
"4" "Waveform" "CO2" "%" 50 50 0.5 0
"5" "-" "untitled" 100
"6" "Evt+-" "Memory" 1
"31" "Marker" "Keyboard"

"CHANNEL" "1"
"Waveform"
"No comment"
"ECG"
" volt" 250

"START" 159.04500 0.00400

-0.23682
-0.01953
0.37109
0.98877
1.78467
2.55371
3.06885
2.98828
2.05078
0.69824
-0.49072
```

Here you see the text output information.

You also see the channels and their sampling rate, scaling factors and titles. This should help to reconstruct the data in other applications.

Channel 1 synopsis.

Here is the data start time and the time between points

-The data values

-ECG Peak

Here is the text output of a histogram. For this I used Copy as text in Spike2 and Paste in Word.

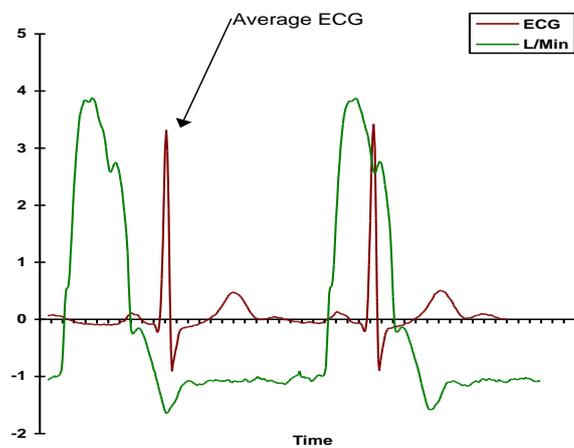
```

150          -Number of bins to follow
0           0          -and their x position and value
0.01       0
0.02       0
0.03       0
0.04       0
0.05       168
0.06       57
0.07       50
0.08       49
0.09       31
0.1        33
0.11       21
0.12       48
1.49      075
etc.

```

Spreadsheet output

The image below is an example of Copy as Text. It is, in fact, a section of the ECG waveform from the data above and an extra respiration channel from a similar file. This was created using Word Graph, but it can also be produced by other spreadsheet packages.



From version 5, you can ask to export time view data in Spreadsheet format. In this mode, the data from each channel has its own column, and Spike2 resamples all the channels to the same rate so that the exported data forms a rectangular table. This makes it very easy to import the data into spreadsheets and most other programs that accept tabulated data.

Sampling data

Overview This chapter investigates how to sample various data types with Spike2. The data types fall into four main categories:

- Waveform data
- Event data
- Marker data
- WaveMark data (not discussed in detail here, see the *Spike shapes* chapter)

Each data type can be processed during the experiment and we will consider the various means of visualising and analysing each data type. Certain topics falling in the domain of data sampling, including spike sorting, generating output sequences and controlling data capture with a script, are covered in other chapters and are not studied in detail here.

Data sampling is the process by which Spike2 captures external signals, displays and analyses these signals in real time, and records the signals on the hard disk of the host computer. To sample data, first use the Sampling configuration to describe the data channels you wish to sample, then create a new data file to hold the data and finally start sampling. We will consider how the sampling configuration dialog relates to each data type in turn, rather than study the configuration independently of sampling.

Starting a new data file The File...New...Data document menu command creates a new document (window) in which data will be sampled. In versions 3 and 4, you can also open a new data file from the Sampling configuration dialog and from the Sample bar. Control over the sampling process is provided by a floating window, or from the Sample option in the Spike2 menu.

Floating command window The floating command window starts and stops data capture. Click the Start button to begin sampling, or if the Trigger box is checked, supply a trigger pulse to the Event 3 input of the standard 1401 and 1401*plus* or to the Trigger input of the micro1401 and Power1401.



When data capture starts, the options to Stop data capture (end the experiment) or Reset the new file (discard all data accumulated so far and prepare for a new recording) appear in this window. Check the Write to disk box to save data to disk. An option to Abort the sampling process is always available in the command window. This option abandons any sampling that is in progress and discards the new file.



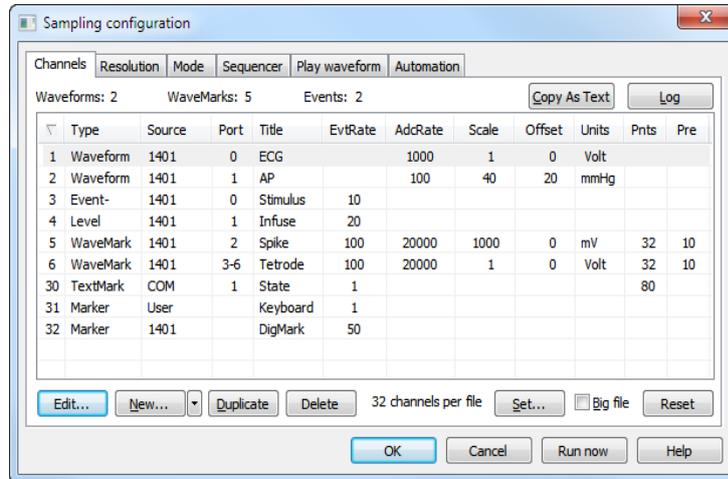
Sampling waveform data A waveform is any signal that varies continuously in time, such as an electrocardiogram, or a temperature recording. Continuous (analogue) data is not suitable for processing in a computer and so a waveform must be converted into a digital format. Spike2 records the amplitude of a waveform at fixed intervals. This information can then be used to reconstruct the waveform. The reciprocal of this sample interval is the sampling rate.

If the original signal is to be faithfully reconstructed, sufficient amplitude samples must be taken per second. A waveform must be sampled at a rate that is at least twice the frequency of the maximum frequency component of the raw signal.

Each waveform sample occupies two bytes of hard disk space. It is therefore necessary to select a sampling rate that is high enough to represent the highest frequency in the signal, but not be so high that unnecessarily large data files are created. With Spike2 you can record waveforms on different channels at different rates. High bandwidth signals, such as EMG can be sampled at high rates whilst lower frequency signals, such as blood pressure can be sampled more slowly.

Configuring Spike2 to record a waveform

The sampling configuration dialog sets the sampling parameters for a new data file. Once you create a new data file, the configuration is fixed. If you change the sampling configuration after creating the file, the changes apply to the next file you create; the current file does not change.



The list of channels to record is edited using the Edit, New Channel, Duplicate and Delete buttons. Reset and Run now were added at version 2. The current channel has a highlight and can be selected using the mouse, or the cursor keys.

Edit The same as double clicking a channel. Edit the current channel.

New channelTo add a new channel of a particular data type to the channel list.

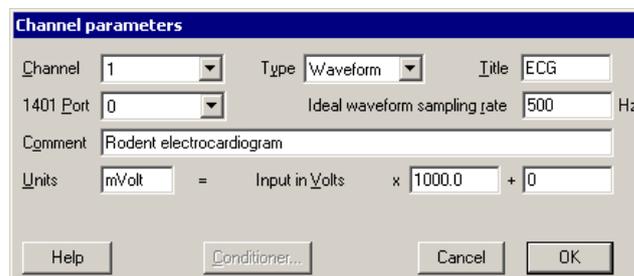
Duplicate Add a copy of the current channel at the next available channel number.

Delete To remove the current channel from the channel list.

Reset Delete all user-defined channels and sets a standard sampling state.

Run now This button closes the dialog and opens a new data file.

You can create up to 64 waveform channels in the channel list (16 in version 3). Click the New button for a new channel or Edit an existing channel and use the Type field to set the channel type to Waveform. The table, below, describes the other fields in the dialog:



Channel This is the channel number. This number is drawn to the left of the data in a time view and is used by the script language to identify a data channel.

Type This sets the type of data to record on this channel (in this case Waveform)

A title for the new channel, displayed in the Y axis for the channel.

Title

The 1401 hardware port to sample the waveform from. There are 16 ports as standard on all 1401s except the micro, which starts with 4. ADC expansion units are available for all 1401s to increase the number of inputs.

1401 port

The desired sampling rate for the waveform in Hz (samples per second). This may not be the actual rate used for reasons given below.

Ideal ADC sampling rate

A user comment for this channel entered here is stored with the data file.

Comment

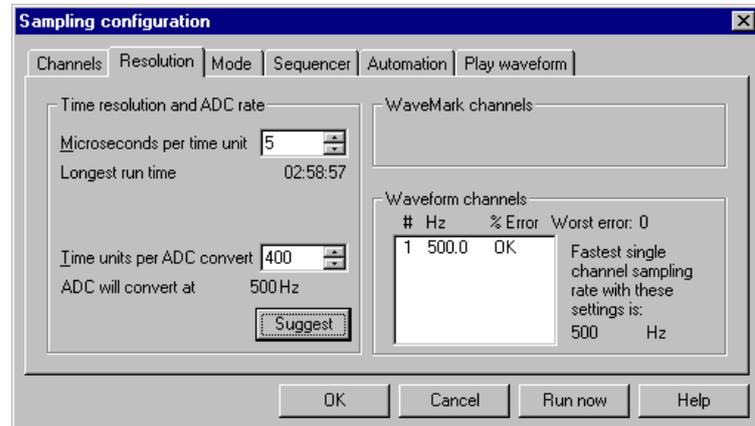
The amplitude units for the waveform (the standard setting is "Volts").

Input in Volts These fields scale and offset the input signal from Volts to user-defined units. The standard values are scale = x1.0 and offset = 0.0

Once the fields are set, click OK to add the new channel to the channel list. Note that the actual sampling rate is displayed rather than the requested ideal rate (see below).

Waveform sampling rates

The sampling rate of a waveform channel depends on the base waveform sampling resolution. This is set in the Resolution tab of the sampling configuration dialog.



Times measured by Spike2 during sampling are relative to a master clock in the 1401. The Microseconds per time unit field sets the clock tick period in the range 2-1000 μ s. There is a maximum of 2,147,483,647 ticks per data file ($2^{31}-1$), which sets the maximum possible recording time per file. With a time period of 10 μ s this is about 6 hours. The Longest run time field displays the maximum sample time.

The Time units per ADC convert field sets the time interval in units of the Microseconds per time unit field between waveform samples from the 1401 Analogue to Digital Converter (ADC). The example to the right shows a simple case with one channel sampled at the same rate as the ADC converts.

Microseconds per time unit (μ s)	5
Time units per ADC convert:	400
ADC conversion interval (μ s)	2000
Total sample rate available (Hz)	500
Number of channels	1
Sample rate per channel (Hz)	500

Versions 2 and 3

With more than one waveform, the channels share the total sampling rate. The example to the right shows the result of adding 4 more channels. The maximum rate for each channel under these conditions is 100 Hz; slower rates are possible by down-sampling.

Microseconds per time unit (μ s)	5
Time units per ADC convert:	400
ADC conversion interval (μ s)	2000
Total sample rate available (Hz)	500
Number of channels	5
Sample rate per channel (Hz)	100

“Down-sampling” means we keep every n^{th} data point for a channel; n is an integer in the range 1 to 65535. The possible sampling rates for the 5 channels in the second example above would be $100/n$. Spike2 chooses n to make the actual rate as close to the ideal rate as possible. In the example to the right, channel 1 can achieve exactly the requested rate, but channel 2 can only approximate it.

Microseconds per time unit (μ s)	5
Time units per ADC convert:	400
ADC conversion interval (μ s)	2000
Total sample rate available (Hz)	500
Number of channels	2
Sample rate per channel (Hz)	250
Ideal rate channel 1	50
Actual rate (250/5)	50
Ideal rate channel 2	35
Actual rate (250/7)	35.71

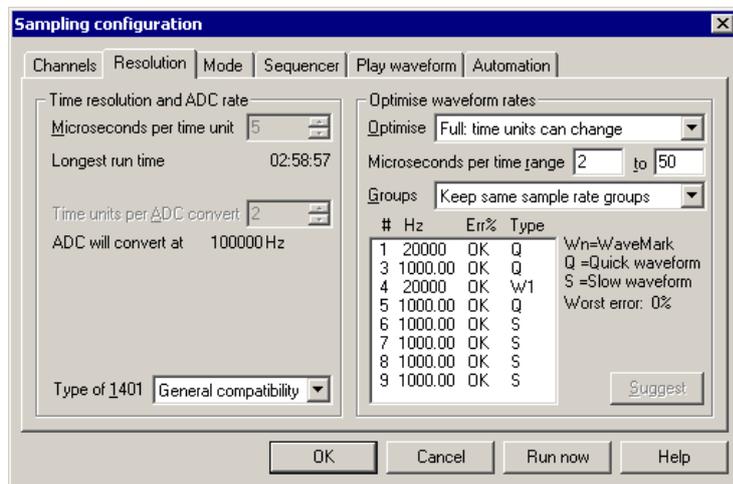
If it was really important to you to get 50 and 35 Hz, we can get pretty close. Observing that the lowest common multiple of 35 and 50 is 350 Hz, if we could get a maximum channel rate of 350, the problem would be solved. Unfortunately, 350 Hz on two channels is a total rate of 700 Hz. This is a sampling interval of 1428.57 microseconds. The nearest whole number of microseconds is 1429, which is prime. The next best choice is 1428, which has factors, so we choose 4 and 357 to give a reasonable resolution.

Microseconds per time unit (μs)	4
Time units per ADC convert:	357
ADC conversion interval (μs)	1428
Total sample rate available (Hz)	700.28
Number of channels	2
Sample rate per channel (Hz)	50.14
Ideal rate channel 1	50
Actual rate (350.14/7)	50.02
Ideal rate channel 2	35
Actual rate (350.14/10)	35.01

Suggest button The Suggest button can be used to ask Spike2 to adjust the Time units per ADC convert field to minimise the error in sampling rates.

Special case If *m* channels of the spike sorting data type (WaveMark) are set, these channels automatically get $m/(m+1)$ of the total sampling rate, irrespective of the number of waveform channels. The waveform channels share the remaining total sampling rate.

Versions 4, 5 6 and 7 From version 4, the Resolution tab has more options and Spike2 has a better strategy for matching the actual sampling rates to the ideal rates set in the Channels tab. If you need exact compatibility with version 3, set the Optimise field to None and the Groups field to Version 3 compatible; the system will then behave as described above for version 3.



If you always use the same 1401 type for sampling, optimise the settings by selecting the appropriate value in the Type of 1401 field. If work with a range of 1401s, select the slowest type you will use. Set the Groups field to Keep same sample rate groups.

For automatic optimisation of sampling rates, set the dialog as shown above. If you need to sample for more than 71 minutes, or with a time resolution that is better than 50 μs, then edit the Microseconds per time range field to limit the time resolution to a range that is useful for you. Spike2 optimises for the minimum error in sample rate. If there are multiple solutions, Spike2 chooses the slowest ADC rate and the longest run time.

To run with a fixed time resolution, set the Optimise to Partial and set Microseconds per time unit manually; Spike2 will optimise the Time units per ADC convert.

Version 8 If you select a 64-bit data file, you can then run with 1 us resolution (or whatever you choose) for as long as you like, so this simplifies the choice of sampling rates.

Waveform analysis options

There are three waveform analysis tools built in to Spike2: Power spectrum, Waveform correlation and Waveform average. Each tool produces a new window containing the results. Additional analyses are possible using a Spike2 script, but these possibilities will not be discussed here.

Analysis result windows are created with the Analysis...New Result View option in the Spike2 menu. An analysis can be created at any time from opening the new data file, even during data acquisition.

All analyses are performed on a particular block of raw data (e.g., the first 10 seconds of the data file, or the whole of the data file so far). The region of data that is processed is determined by a Process dialog, which follows the analysis parameters dialog.

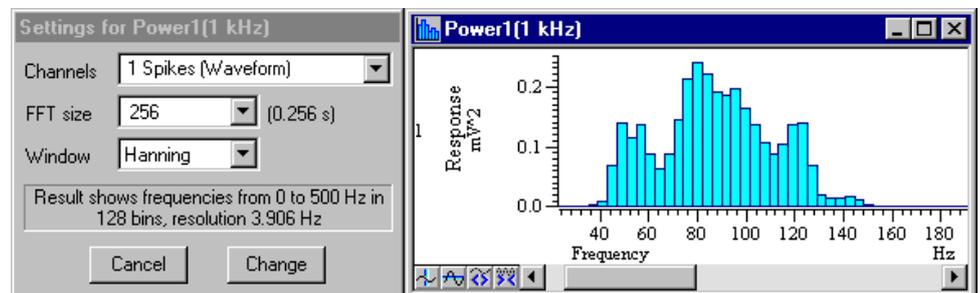
In versions 2 and 3, you set the width of the result views in terms of the number of bins. From version 4, apart from the Power spectrum, you set the width as a time and Spike2 calculates the number of bins for you. Version 4 also allows you to analyse multiple channels per result view, however the channels must all have the same sample rate.

Power spectrum

A power spectrum displays the frequency components of a waveform signal. The components are measured in units of root-mean-square power. The analysis is performed using a mathematical operation called a Fast Fourier Transform (FFT) that resolves a block of waveform data points into corresponding cosine (single frequency) components. The FFT transforms blocks of data points that are a power of 2 in size, from 32 to 4096 points. The data that is processed by the FFT must contain at least as many data points as the FFT block size.

The size of the block determines the resolution of the power spectrum; the higher the block size, the better the frequency resolution of the result.

The analysis parameters dialog for a power spectrum is illustrated below, together with a power spectrum. The user need only set the FFT block size. Before version 4 there is no option to set the window function.



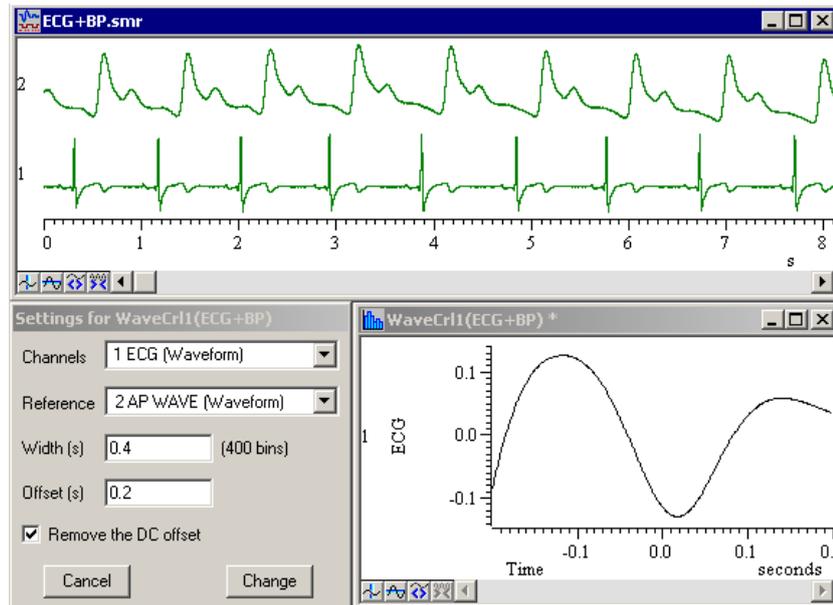
The full x axis range runs from 0 Hz to half of the sampling rate used for the waveform (the maximum frequency in the waveform is expected to be less than half of the sampling rate to avoid the problem of aliasing, mentioned above). The number of bins (elements) in the result is one half of the FFT block size selected.

Example: Power spectrum with a block size of 256 of a waveform sampled at 1000Hz.

The resulting spectrum will have an x axis that runs from 0 to 500 Hz. There will be 128 bins in the result. The spectral resolution (frequency width of each bin) will therefore be $500/128 = 3.9$ Hz. A better resolution will be achieved by supplying a bigger block size, but at the cost of time resolution in the original waveform (because a larger block size spans a longer time period).

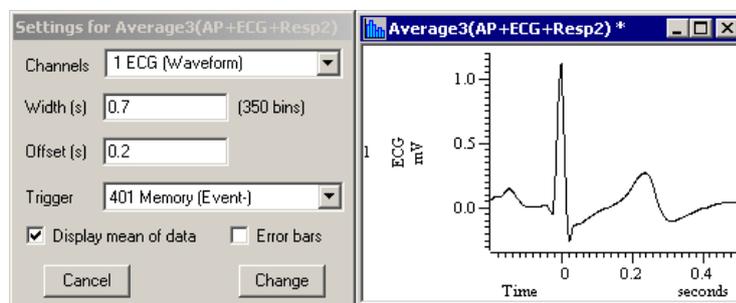
Waveform correlation

This analysis measures the similarity of two waveforms with the same sample interval. The first result point is calculated by multiplying all of the data points in one waveform, the “reference”, by all of the data points in the second waveform and summing the results. The reference waveform is moved one sample point to the right and the process repeated for the next result point until all of the result view bins are filled. The result is scaled so the y axis of a waveform correlation runs from -1 (correlated but inverted), through 0 (uncorrelated), to +1 (full correlation). There is an option to remove the DC offsets from the waveforms so offsets do not change the result. The picture shows two waveforms, the waveform correlation settings dialog and the resulting correlation.



Waveform average

This analysis averages a raw waveform with respect to a trigger event. The triggering events may be recorded on an event or marker channel (see later), or manually set by the user. The analysis parameters dialog for a waveform average is illustrated below.



The Channel field sets the waveform channel. Number of bins sets the number of waveform points in the average. The Offset field sets the pre-trigger time in the result. The Display mean of data field should be checked to generate an average, or unchecked to generate a sum of all of the waveform sweeps.

From version 4 there is an extra option to display error bars. If you use this you can choose to display 1 or 2 Standard Errors of the Mean (SEM) or the standard deviation for each bin in the average.

Sampling event data

A Spike2 event channel holds time stamps. For the 1401 to record the time stamps, something must convert the event into a TTL compatible voltage pulse. TTL (Transistor-Transistor-Logic) is a standard used to define voltage levels in digital equipment. A voltage signal is TTL high if it lies between 3 and 5 Volts and TTL low if it is between 0 and 0.8 Volts. Event inputs are normally connected to the 1401 digital inputs.

TTL signals are commonly found in experiment conditions. Pulses used to trigger data capture or trigger the accumulation of an waveform average are often TTL-compatible. A waveform discriminator usually generates TTL-compatible pulses when it detects an action potential.

Spike2 records up to 8 channels of events. Signals are connected to the digital input on the standard 1401 and *1401plus*. The micro1401 and Power1401 use the back panel digital input with 2 channels also available as front panel BNC connectors. The Spike2 top box makes all 8 inputs available on BNC connectors for micro1401 and Power1401.

An event is recorded as a time in the data file. The concept of sampling rate is not relevant to the capture of events. Each event occupies 4 bytes of hard disk space. The timing resolution of event data is set by the Microseconds per time unit field of the sampling configuration.

Configuring Spike2 to record an event channel

An event channel is configured in the sampling configuration dialog in a similar way to waveform channels (above). The following parameters must be supplied:

Channel The channel number to use. Spike2 uses this number to refer to the data that is recorded in this channel.

Type The data type. This field can be set to **Event+** to record a positive-going level change, **Event-** to record a negative going level change, or **Level** to record both transitions.

Title A title for the channel

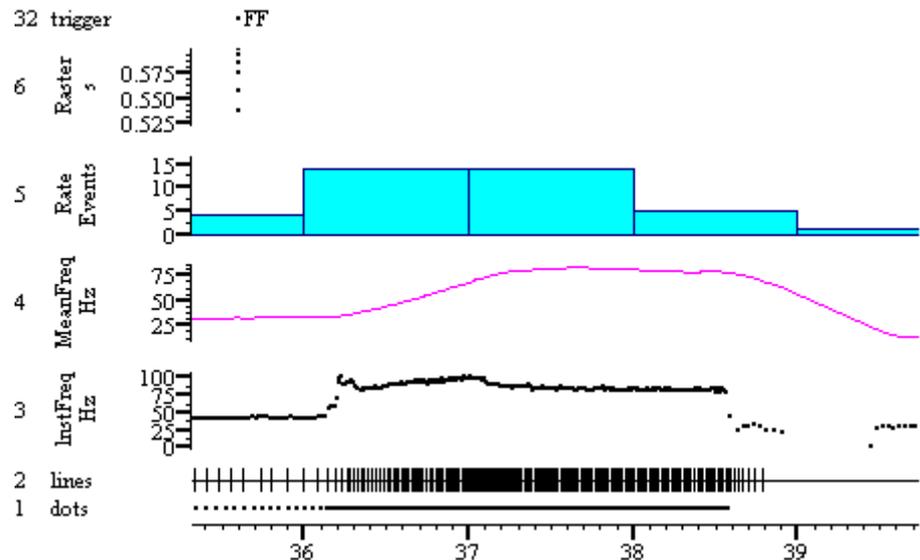
1401 port Ports 0 to 8 are available on any 1401. Note that it is possible to use port 0 for a waveform channel and port 0 for an event channel because the physical ports for each data type are different on the 1401.

Maximum sustained event rate An estimate of the maximum number of events to record each second, used to allocate resources during recording. A good estimate of the value of the **Maximum sustained event rate** field will improve sampling performance. Spike2 allocates resources to channels based on the data flow rate through them.

If you give a channel a higher estimated rate that it needs, you will starve other channels of resources. This is not important at low data rates, but as the total data throughput approaches the maximum rate that your system can handle, setting good values can make the difference between a machine that feels sluggish as it is writing blocks to disk all the time and one that feels responsive.

Displaying event data during sampling

Each event data channel can be drawn in different ways. The drawing modes are available from the View...Event Draw mode menu option in Spike2. The modes are:



Dots Events are drawn as large or small dots (varying sizes are available with a script).

Lines Events are drawn as vertical lines at the time they occur.

Rate The number of events occurring in a user-defined period is plotted as a rate histogram, the width of each histogram bin being the selected period.

Instantaneous frequency The event is plotted as a dot. The x axis dot position is the time of the event. The y axis dot position is the instantaneous frequency of that event in Hz with respect to the previous event.

Mean frequency The events are plotted as a line. The frequency of each event is calculated over a fixed period that is set by the user in the Bin size field for this event draw mode. The frequency calculation depends on the interval between the first and last events in the time period. If the interval is less than half of the period:

$$\text{Frequency} = \text{number of events in period} / \text{period}$$

If the interval between first and last event is greater than half of the period:

$$\text{Frequency} = (\text{number of events}-1)/\text{time between first and last event}$$

Raster This mode displays the pattern of event logging following a stimulus event. Each stimulus event is plotted on the x axis at its occurrence, with a y value of zero. The latency of events in the response channel that fall around the stimulus with respect to that stimulus are plotted on the y axis. Only events falling between the previous trigger time and the following trigger time will be plotted for each trigger event.

Event analysis options

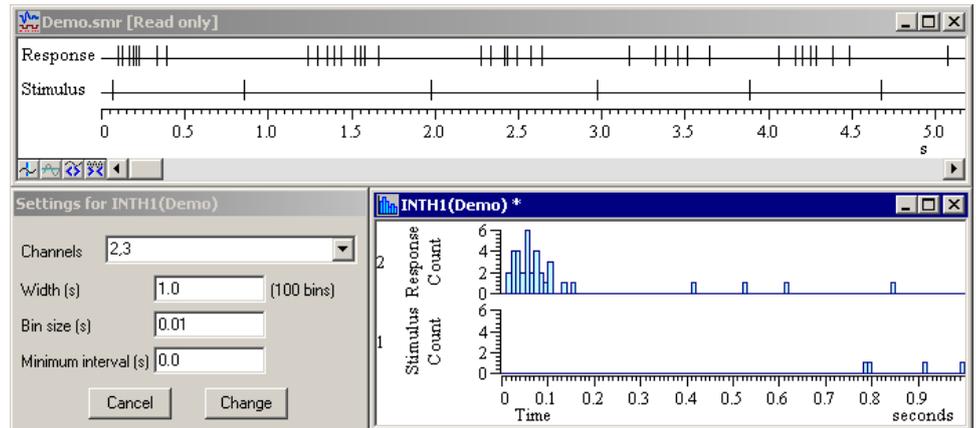
Event analyses are available from the Analysis...New Result view. These analyses process raw event data according to the Process... criteria that apply to waveform data.

In versions 2 and 3, you set the width of the result views in terms of the number of bins. From version 4, you set the width as a time and Spike2 calculates the number of bins for you.

From version 4, all event analyses except the Interval histogram have the option of a Raster display. This shows the events that form the histograms sweep by sweep. You can also analyse multiple channels per result view.

Interval histogram

This analysis evaluates the frequency of event intervals. The analysis parameters dialog is illustrated below, together with the source events and the result.



Channel The event channel (from version 4 you are allowed multiple channels) to analyse

Number of bins The number of bins in the result view, limited only by available memory

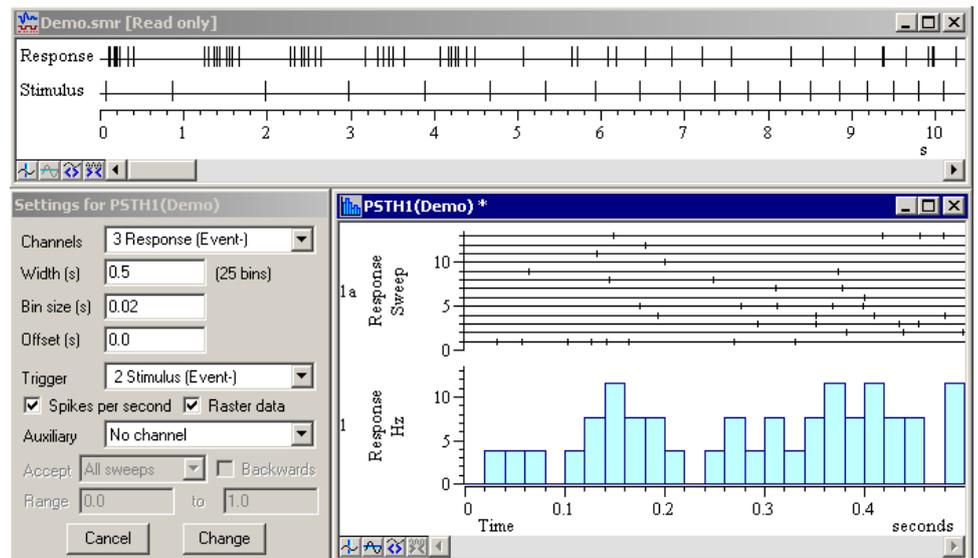
Bin width The width of each bin, in seconds.

Bin size The shortest interval to display in the result. All event times are multiples of the Microseconds per time unit field (2 - 1000 microseconds) set in the sampling configuration. To analyse small intervals, you must record data with a time unit that gives sufficient resolution for your analysis.

Minimum interval

Stimulus histogram

This analysis maps the occurrence of events around a trigger. The analysis parameters dialog is shown below, together with the trigger and response channels and the result.



Channel The event channel (or channels from version 4) to analyse

Number of bins and bin size These fields set the resolution (bin size) and time base (bin size * bins) of the result

Applies an offset for mapping pre-trigger response events

Offset The channel containing the trigger events. This could be another event channel, or a marker channel (see later). If no channel is selected, the trigger time is supplied manually in the Process dialog. If a trigger occurs within a preceding analysis sweep, the trigger is ignored.

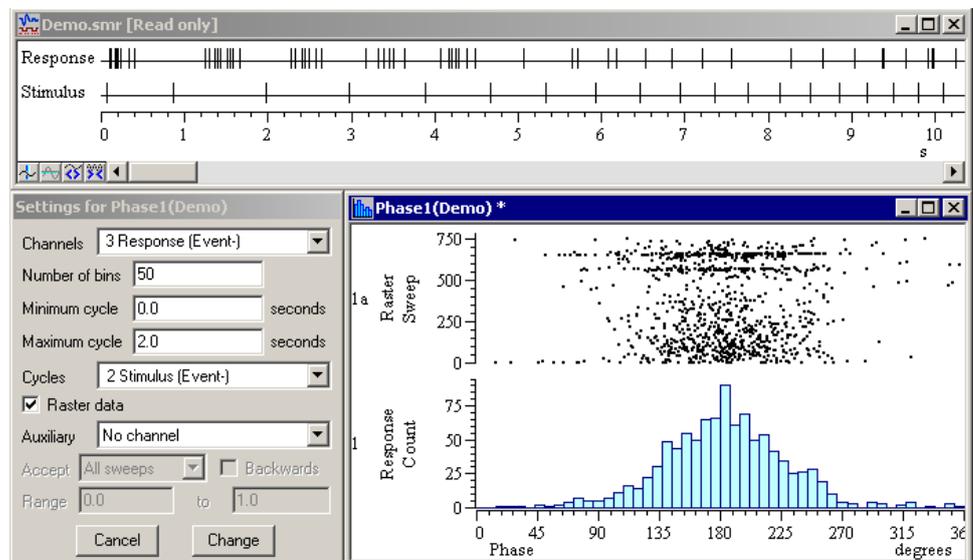
Trigger

- Spikes per second** You can display the result as event count per bin, or check this box to normalise the result by dividing the event count per bin by the number of sweeps and the bin width.
- Raster data** Check this box to save the times of each event in each sweep. Each result view channel is then duplicated and the duplicated displays the raster for each sweep. The screen image shows the raster data drawn in **Raster** line mode with the centre line enabled.
- Auxiliary** You can nominate an addition channel to use as a data source for measurements, and then use the measurement to sort the rasters, to include and exclude sweeps and, if the measurement is a time, to display a symbol. If the auxiliary channel is a waveform, the measurement is the waveform value at time 0 in the sweep. Otherwise, the measurement is the first/ last event time after/before the sweep trigger point on the auxiliary channel.

Event correlation This analysis is very similar to the stimulus histogram analysis. There is, however, an important distinction. Any trigger event falling within a preceding analysis sweep is included and used to build the result.

It is possible to generate an auto-correlation by setting the stimulus channel the same as the response channel. In the auto-correlation, the correlation of an event with itself at time 0 is ignored.

Phase histogram This analysis maps events according to their position in a cycle. A cycle is defined by another event or marker channel and can have varying cycle times. Each event in the cycles channel marks the end of the preceding cycle and the start of the next cycle. The Number of bins field in the analysis parameters dialog defines the number of time elements that each cycle will be divided into for mapping the responses. The width of each element (bin) thus varies according to the width of the cycle. A minimum and maximum cycle width can be set to exclude unwanted cycles.



You can also include raster data in this analysis. The screen image shows the raster data drawn in **Raster** mode.

Sampling marker data A marker is a time stamp plus four numeric codes (usually only the first code is used). Markers are often used to annotate a data file with information about the experimental conditions. The markers in a channel can be filtered into groups of a particular type based on the codes for conditional processing of data.

Marker data can be processed in the same way as event data (they can be used as trigger sources or analysed in the same fashion as event channels).

There are three types of marker data that can be recorded in a Spike2 data file.

Keyboard markers Spike2 always reserves channel 31 for keyboard marker information. This channel cannot be deleted from the Sampling configuration dialog. You can add a keyboard marker to the file by pressing a key, but only if the sampling window is the currently active window. If a text window is active, or another application, your keys will not be logged. You can also set keyboard markers from a script, and particular keyboard markers can be linked to an output sequence (see the next chapter) and used to trigger sampling.

Keyboard markers are logged to an accuracy of about 1 second. It is important to ensure that the raw data window is current for the key press to be logged. Keyboard markers each occupy 8 bytes of hard disk space.

Digital markers A digital marker is recorded from the Digital In port of a 1401 whenever a TTL compatible pulse is applied to the E1 BNC input on a 1401 (strobe input on a micro1401 or Power1401). The E1/strobe pulse can be simulated by the `REPORT` and `MARK` instructions in an output sequencer file (see the next chapter).

Channel 32 is reserved for digital markers. Each digital marker is a byte value (0-255) which represents the pattern of 8 digital (on/off) inputs read from the Digital In port. Because of the large number of possible patterns (256), digital markers can be used whenever a large number of different experimental conditions need to be recorded. Digital markers are logged with the same timing accuracy as events and occupy 8 bytes of hard disk space. The digital markers can be selectively filtered in the same way as the keyboard markers, and may also be processed in the same fashion as event data.

Text markers These markers add timed comments (up to 100 characters in length) to the data file. Channel 30 is reserved for TextMark data. The `Sample` menu `Create text mark` command generates text markers during data capture.

The comment includes user defined marker codes, which can be used to selectively filter the markers. The amount of disk space occupied depends on the length of the comment. The text mark is recorded visually in the data file as a small document icon. You can view and edit the comment by double-clicking the icon. The time cannot be edited.

A special case: WaveMark data WaveMark data is discussed elsewhere. This is a special data type that is the result of discriminating action potentials from a raw waveform. The data consists of short waveforms, each describing the shape of the action potential. Each WaveMark is accompanied with a code number that is assigned according to the shape of the unit. The data can be treated as event and waveform data for analysis purposes.

Additional data sampling tools

Further tools exist for optimising your data sampling configuration. These can be divided into three categories:

- Display tools
- Measurement tools
- Processing tools

Display tools

The display tools are available from the **View** menu in Spike2. You can also activate some by double-clicking on the data window with the mouse. Many controls have short-cut key equivalents (these are listed in the **View** menu). Commonly used controls are available from the system toolbar.

Mouse control

You can zoom in on an area of interest in the data, result or XY (not version 2) view with the mouse. Click and drag a rectangle around the area you wish to see in more detail, and the rectangle will expand to fill the available space.

The x axis can be doubled or halved in length by clicking the zoom-in and zoom-out icons in the bottom left corner of a data or result window.

Double-click the x axis or y axis to open a dialog that sets the axis range. You can set the start and end of the axis range. For the x axis you can also set the width.

The scrollbar at the bottom of a window indicates the current display position with respect to all of the data in the file (or result). In a raw data file, if the scroll-bar is right-most, the incoming data will cause the data in the window to scroll off the left side. The scroll-bar can be moved to any position to access any portion of the data file recorded so far (the scroll-bar will then move as incoming data affects the relative position of the currently viewed data).

In version 4 there are additional mouse controls. Click and drag axis ticks to scroll an axis, click and drag the axis numbers to scale the axis. If you hold down **Ctrl**, the display updates during the drag. If you click in the data area with **Shift** down, you can change the size of the channel areas. You can also change the channel order by clicking and dragging the channel numbers. See the version 4 documentation for details.

Toolbar, Status bar

The Spike2 toolbar includes short-cut buttons for commonly used functions. The status bar provides information about current activity in Spike2. Both of these bars can be hidden to optimise space for viewing data.

Commonly used View menu commands

The following commands are located in the **View** menu and you will find that you use them frequently. Because of their frequency use, most of them have short-cut keys and are duplicated on the Spike2 tool bar.

View...Show/hide channel

Channels can be turned on or off, and a grid can be superimposed on the data.

Standard display

This option returns the window to a standard state. For a raw data window, all channels are displayed in their default state, with x and y axes visible and the grid off. In a result view, the data is drawn in a standard mode with both axes visible.

Result draw mode

This option is available if the current window is a result. The data can be drawn as a histogram, line, dots, or skyline.

Font and colour options

The **Font**, **Use colour**, **Use black and white** and **Change colours** options are used to set the text font and to control the colours used in Spike2 for drawing different objects.

Measurement tools

Measurements are taken from time and result views with cursors. You can add up to 9 (4 before version 4) cursors with the **Cursor** menu **New cursor** command, or by clicking the cursor icon in the bottom left corner of a time or result window. Cursors can be labelled in different ways using the **Cursor** menu **Label mode** option: number, position and position and number. The Spike2 script supports user-defined labels.

The **Cursor** menu **Display Y values** option displays measurements at the cursor position in a new window. For waveforms, the result is the amplitude of the nearest data point. For event data, the measurement depends on the channel display mode. 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 be made between pairs of cursors with the **Cursor** menu **Cursor regions** command. A new cursor measurements window is created in which one of three types of measurement can be made between adjacent cursors. 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.

	Result view	Time view
Mean	Sum of bin contents between cursors divided by the number of bins	Mean value of all waveform samples or mean event rate
Area	The mean value between the cursors multiplied by the distance between them	Area under the waveform between the cursors or number of events between cursors
Slope	The slope of the least squares best fit line for the bin values between the cursors	Slope of the least squares best fit line to the waveform between cursors (no measurement for event data)

Version 4 introduced active cursors that can seek data features such as peaks, troughs and threshold crossings. With the new **Analysis** menu **Measurements** you can create **XY** view graphs from cursor measurements of a range of data in a time window.

Processing tools

The marker filter

You can open the marker filter dialog from the **Analysis** menu **Marker filter** command. It is used to select groups of one or more particular markers from a marker data channel.

Duplicating channels

To duplicate a channel, select it by clicking the mouse pointer over the channel number, then use the **Analysis** menu **Duplicate channel** command. This useful feature allows you to display the same data in different ways. For example, you can duplicate an event channel to create two channels, and then draw one as lines and the other as a mean frequency. This also allows you to duplicate a marker channel and perform different filters on any number of multiple channels.

Duplicating windows

The **Window** menu **Duplicate window** command creates a copy of the current time view. This is useful for creating multiple views of the same data with varying time bases.

Additional sampling parameters

Sampling mode

The sampling configuration dialog has additional fields that give you further control over sampling:

Spike2 can be used to sample data continuously, in timed epochs (e.g. sample for 1 minute every 5 minutes), or in response to a trigger event (sample for a given time after the trigger event). The trigger event may occur on an event or marker channel that has been described in the channel list.

File size and time limit

The maximum file size and maximum sample period are set in the sampling configuration.

The graphical output sequence editor

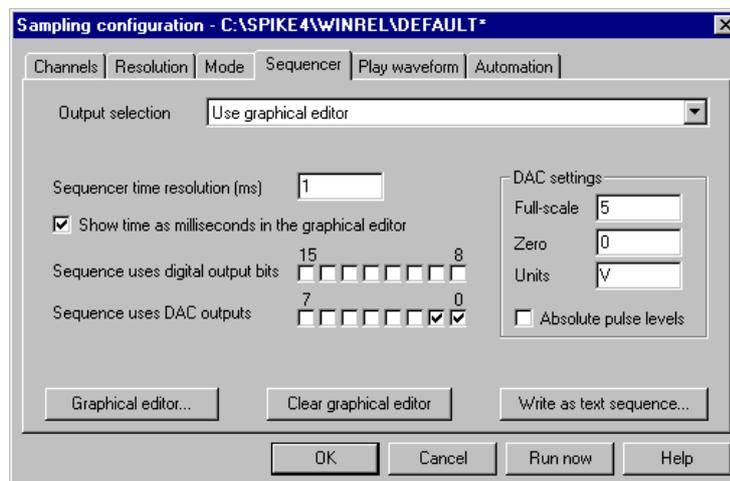
Overview While sampling data, Spike2 can generate precisely timed digital pulses and analogue voltages, monitor your experiment, and respond to input data in real time. This is achieved with the Spike2 output sequencer, which executes a list of sequencer instructions at a constant, user-defined rate. The sequencer can generate pulses, ramps and cosine wave outputs, trigger waveform replay, test digital inputs or recently sampled values and generate delays, as well as other more complex effects.

Before Spike2 version 4.04, output sequences were generated with a text editor where each line of text generated one instruction. Many users found this difficult to learn to use, so CED have supplemented this with a graphical sequence editor where output actions are represented by graphical items that can be selected, dragged and edited.

	Graphical sequence	Text sequence
Edited with	Built-in graphical editor	Built-in text editor
Visualise output	Yes	No
Stored	In sampling configuration	.PLS sequence file
Implemented by	Drag and drop editing	Machine code like language
Ease of use	Very easy to learn and use	Takes time to learn
Flexibility	Uses pre-set building blocks	All features available
Timing	Several instructions per item	One instruction per text line

If your requirements can be met by the graphical sequence editor, you will find it much easier to use than the text sequence editor. However, it has limitations and you can write more complex sequences using all sequencer features by using the text editor. You can save a graphical sequence as a .PLS file to allow you to view or edit it; but you cannot convert a .PLS file into a graphical sequence.

Editor settings To view the graphical editor settings, open the Sampling configuration dialog and select the Sequencer tab. Then select Use graphical editor from the Output selection drop down list. The editable fields in this dialog set values that apply to the entire sequence:



Set time resolution The sequencer time resolution sets the time resolution of your sequence (the clock interval of the sequencer clock), this is also the minimum duration of any pulse. You can set values in the range 0.01 to 3000 milliseconds. All actions in the sequence occur at integer multiples of the time you set here, some take more than one interval. The table below shows the minimum clock intervals, the approximate time per step and the extra time used for cosine output for each type of 1401, all in microseconds.

	Power	Micro mk II	micro1401	1401plus
Minimum resolution (us)	10	10	50	50
Time used per tick (us)	<1	~1	<8	<10
Cosine penalty/tick (us)	<1	~1	~4	~10

The Minimum tick is the shortest interval we allow you to set. The Time used per tick is how long it takes to process a typical instruction. The Cosine penalty/tick is the extra time taken per cosine output point. Time used by the sequencer is time that is not available for sampling, spike sorting or arbitrary waveform output. To make best use of the capabilities of your 1401 you should set the slowest sequencer step rate that is fast enough for your purposes. For our purposes the default resolution of 1 millisecond is fine.

Show time as milliseconds Check this box to display and edit time in the graphical editor as milliseconds and not seconds. This is purely for your convenience; if your sequence sections are all less than a second you will probably find it more convenient to use milliseconds, which is what we shall do.

Select used digital outputs Check the boxes for the dedicated digital outputs that you will use. Only these outputs will appear in the editor, reducing visual clutter. If you do not require any digital outputs, clearing all the check boxes will save an instruction at the start of each sequencer section.

Select used DACs Check the boxes for the Digital to Analogue Converters (voltage output devices) that you will use in your sequence. Unused DACs are not included in the graphical editor (to reduce visual clutter) and are no sequence code is generated for them, which saves instructions at the start of each sequence section. We will use DACs 0 and 1 only.

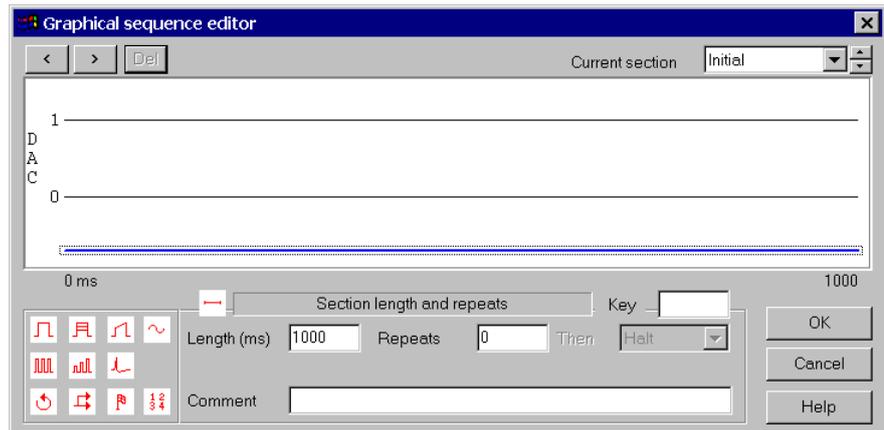
DAC full-scale, zero and units You can define the DAC outputs in units of your choice. 1401 DACs normally have a range of ± 5 Volts, but ± 10 Volts systems exist. Set the full-scale value to the value in the units that you want to use that corresponds to the maximum output from the DAC. Set the zero value to the value in your units that corresponds to a DAC output of 0 Volts. Set the units to the units you want to use.

Absolute pulse levels The DAC pulses take their starting level as the current DAC value at the pulse start time. The DAC then changes to another value, then back to the original level. Normally you define pulses in terms of the pulse amplitude relative to the starting level and all pulses add. If you check this box, then you set the absolute level for the DAC to change to.

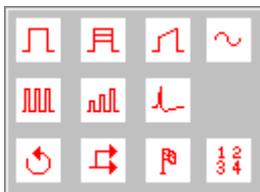
Graphical editor and paired pulses Click the Graphical editor button to open the editor. The graphical editor display is shown below. It has controls at the top for selecting pulses and sections, a display area in the middle where outputs and actions are shown, and more controls at the bottom. We will demonstrate the basic capabilities of the graphical editor by generating outputs suitable for an imaginary paired-pulse experiment.

The display area always contains a *control track* drawn as a thick blue line at the bottom. We chose output on DACs 0 and 1 only, so there are also two DAC output traces filling the remainder of the space. If you stretch the dialog by dragging the bottom-right corner, this area gets bigger so you can see the traces better, you can also zoom a trace to fill the area by double-clicking it. There is always one item selected in this area; the selected item has a grey rectangle around it. Click on the control track now.

Initial display

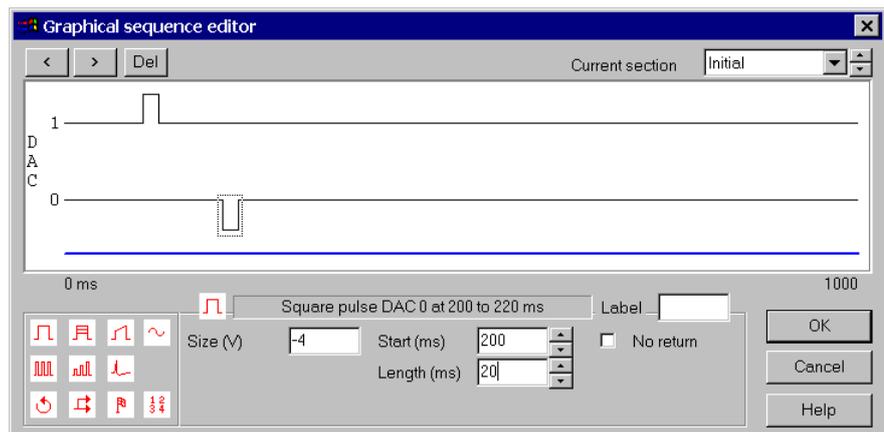


The central area below the display shows the parameters for the selected item, selecting the whole control track gives the parameters for the output section as a whole (the outputs are handled as one or more sections – we will look at sections later). We set the section length to 1000 milliseconds, and set the repeat count to zero, meaning repeat forever.



The next task is to create the pulse outputs on the DACs, we want one pulse on each DAC, each 20 milliseconds long, with a 100 millisecond delay from the first to the second. The area at the bottom left of the editor is a palette holding icons for the available pulses and actions; you can add an item by dragging it to the required trace. We just want a simple pulse, which is the item at the top left of the palette (□), so we drag one onto the DAC 1 trace. While we are dragging it along the trace a numerical display shows approximately where we are, but we can edit the position later to get it exactly right.

Display after adding 2 pulses



Once the pulse is dropped into position it appears, already selected, in the DAC trace and we can edit the pulse parameters, setting the correct pulse length and amplitude. While we are doing this, experiment with the < and > buttons at the top left, which toggle the selection forwards and backwards through the items available for that output. Usually it's easier to click on an item directly but if it is very small, or there are two items very close together, this can be difficult and these buttons can be very useful.

We can also click on the rest of the DAC trace to view and edit the initial DAC level, but we don't need to change this. Once the DAC 1 pulse is done we can add a second pulse, this time to DAC 0, and set it up as required. And that's all; if we run the sampling configuration we can see the result. Note that the sequencer control panel appears, with this sequence it doesn't do much but you can see the sequencer step change as it executes. This sampling configuration is saved on disk as PAIRED1.S2C.

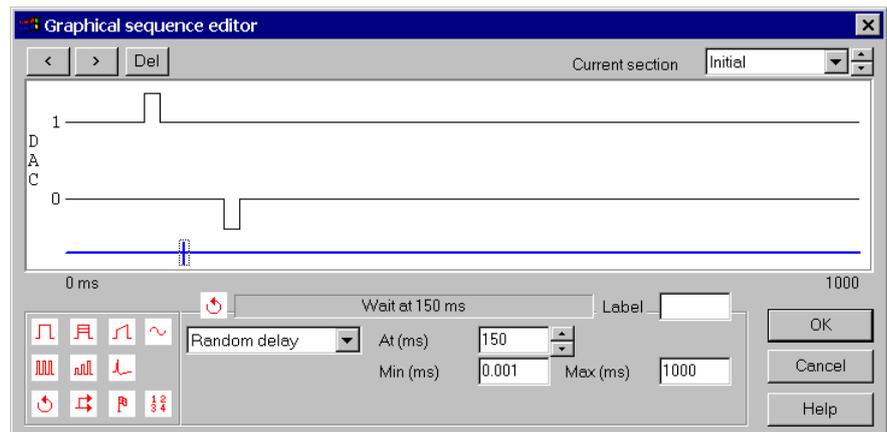
Varying amplitude pulses

We can demonstrate a few more features of the graphical editor by making our outputs more complex. Let us start by varying the amplitude of the second pulse, the one on DAC 0. First delete the current pulse by selecting it and pressing the Del button at the top, then drag a varying amplitude pulse from the palette (⌘) to replace it. It's basically the same as the simple pulse, with an extra parameter to specify the change in the amplitude each time it is used.

That's all very well, but with continuous incrementing, we will get a range of pulse amplitudes running all the way from -4 volts to $+4$ volts then wrapping round to -5 and continuing on. Let's say we only want pulse amplitudes from -4 to -1 Volts, so we need to restart the initial section after four cycles to reset the pulse amplitude. We do this by adjusting the way the section repeats, as the pulse amplitude is reset to the initial value whenever the section restarts. So we select the control track and set the section to repeat 4 times and then switch to itself again. When we run that we get the correct outputs.

Wait items

We can also modify the inter-pulse delay. Currently there is a fixed 100 millisecond delay, we can change this to a random delay of between 100 and 1100 milliseconds – this may be rather large for research purposes but it makes it easy to see what is going on. The items in the bottom row of the palette are all control items, the one on the bottom left (⏸) is a wait. Drag a wait item into the control track (it won't go anywhere else) between the two pulses, when you drop it, it appears as a blue marker, with a selector for the type of wait among the parameters. As you can see, many types of delay and wait are available, we will select Random, and set the range to 0 to 100 milliseconds. As you see, we cannot quite reach zero, if we need precise timings we should move the second pulse forwards by one millisecond. This one is on disk too (PAIRED2.S2C).



We could use a number of other types of wait to achieve different effects, for example a Poisson delay, to wait for a specified number of events, to be logged or to wait for a waveform value on a channel to cross a specified limit limits.

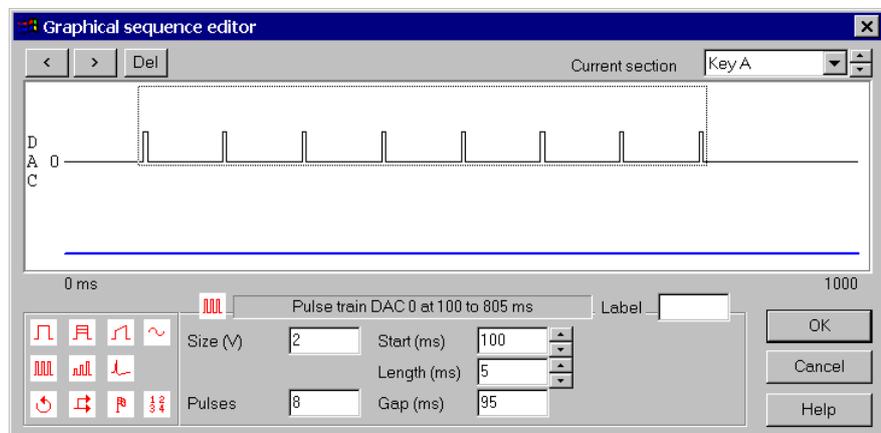
If you leave the graphical editor and use the **Write as text sequence** option you can save the sequence used as a .PLS file and open it in the pulse sequence editor to examine the generated sequence. Dissecting the operation of a generated script is a good way to learn about the sequencer, or you can edit the sequence text directly to customise its behaviour.

Graphical editor Pulse trains

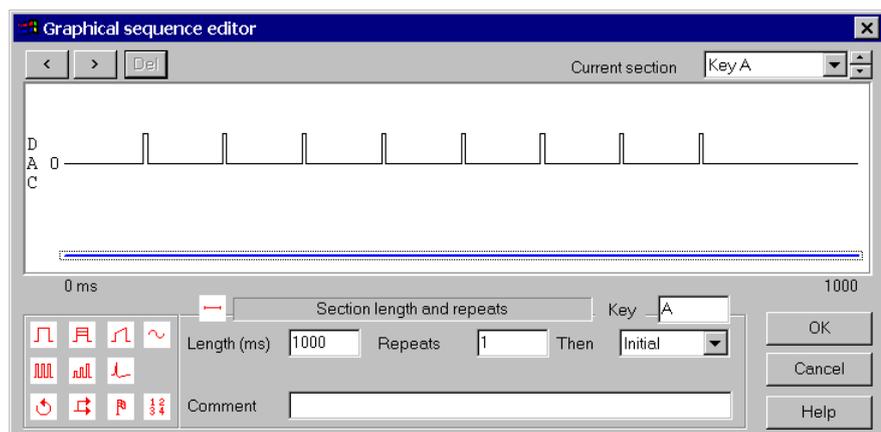
We will now examine some more graphical editor facilities by using them to generate another set of outputs, this one generating pulse trains of varying frequency on DAC 0. We want 10, 20, 50 and 100 Hz pulse trains, each 800 milliseconds long. We could do this by placing the pulse train outputs one after another into our initial outputs section, but we get much more flexibility if we use a different section for each frequency.

Sections There are 27 sections in the graphical editor, called Initial and Key A through Key Z. The Current section selector at the top sets the section to display and edit. The Initial section runs when the sequence starts; in many cases (such as our paired pulses) this may be the only section you need. The remaining sections can be used as required, we can set each section to a different length and give them different contents, and link them together in many ways to give the effects we require.

Clean up the editor using the Clear graphical editor button and get rid of DAC 1 from the outputs and then go into the editor. We will not have any initial outputs, so we leave the initial section alone for now and switch to section A. We set this to be 1000 milliseconds long, to run once and then halt, and then use the pulse train item from the palette (■) to generate our pulses.



We also want to associate a keyboard character with the section using the Key parameter. If you select the control track, you can see the key for the section. By default, the Key A to Key Z sections are assigned keys A to Z respectively and the Initial key is blank. However, you can set the key to any of A-Z, a-z or 0-9 (upper and lower case are different). The default value of A is fine for our purposes.



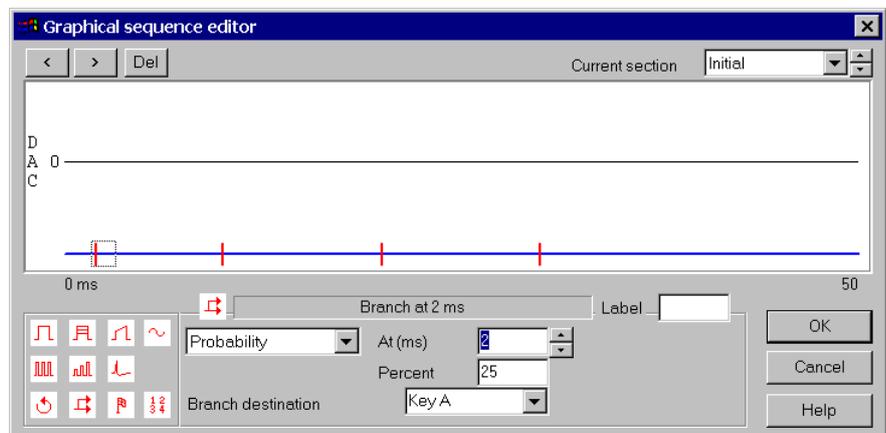
Then set up sections B, C and D with similar pulse trains. The configuration is on disk as TRAIN1.S2C so you don't have to watch me set everything up. If we run it the sequencer control panel appears with the relevant keys on buttons. We can switch to a given section by pressing the appropriate button, or by pressing a key on the keyboard (remember we used capital letters).

Switching and branching

Instead of controlling the pulse trains to be output manually, you might want something automatic. If we set the Initial section to switch to section A when it finishes, A to B, B to C, C to D and D back to A again, we will get a simple repeating sequence of outputs after an initial delay (which we can make very short by truncating the initial section).

To get more complex behaviour, we can use branch control items within the initial section to select the pulse trains. The branch control item (↗) switches execution of the sequencer to another section (or to a label in this section). It is the second item on the bottom row of the palette, once again there are many possible branch operations.

We will set up a random selection between A, B, C and D, with each frequency having equal probability of use. We do this with four branches to A, B, C and D one after another, having probabilities of 25, 33.33, 50 and 100 percent respectively. The first 3 are Probability branches, while the last one is simply unconditional. You can see that this gives an even spread of probabilities for the outputs. This is another one that I have done on disk (TRAIN2.S2C).

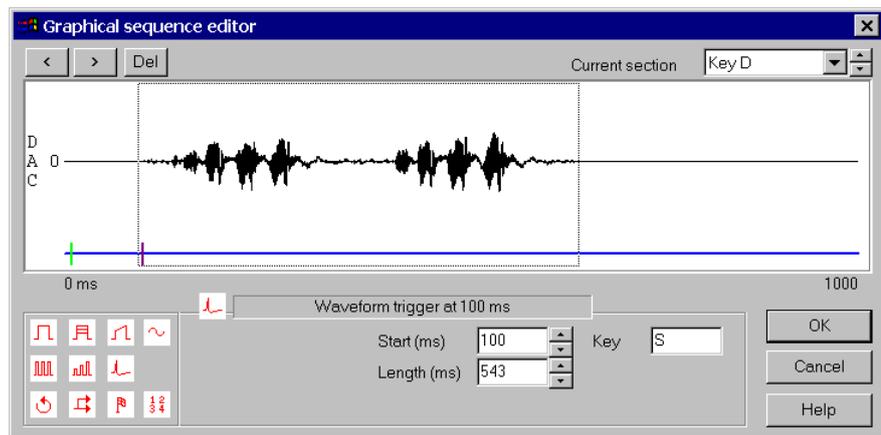


We also have to do two more things to get this working nicely. Firstly each of the other sections must be set to branch back to the initial section when it is done, which is very easy.

Secondly, you can see from the different times at which the four branches occur that the timing of this sequence will not be perfectly regular. I have shortened the initial section to keep the overall variation small, but I cannot remove it entirely. This is unavoidable – the sequencer can only do one thing at a time - but it might make analysis rather difficult. It would also be helpful if we could tell which pulse train was being generated, as we will lose the keyboard markers. We handle this by adding a control item to generate a digital marker (Ⓜ) at the start of sections A to D and adding a digital marker channel to the sampling configuration to receive the markers.

Arbitrary waveforms To finish, we shall replace the pulse train generated in section D with a bit of arbitrary waveform. Waveforms for output by Spike2 during sampling are stored in the Play waveform section of the sampling configuration; they are set up by the **Output waveform** command from the **Sample** menu and are manually triggered using a specified keyboard character.

The graphical sequence editor can also generate the trigger to start waveform output. Go to section D in our outputs, select the pulse train and delete it. Then drag a waveform trigger item (⏏) to the control track at the same position. Note that this goes on the control track, not a DAC, as this is a trigger for output to DACs that are specified in the Play waveform page. The key code for the waveform trigger item has to be set to the code for the relevant waveform. I have put a slice of cricket song into this sampling configuration, and given it S as a trigger code. You can also set the length of the waveform; if we try to set 800 milliseconds the length is truncated to the length of output available.



You can however set the length to values shorter than the entire wave to truncate the output. Now, when we run the sampling configuration, section 4 generates our waveform instead of the pulse train. In the same way we could set up different waveforms to be output by sections A to C and observe the result of (say) four different songs on the auditory system of an insect.

Overlapping items If digital pulses overlap, the result is the logical OR of the pulses.

If DAC items overlap on the same channel, the output depends on the state of the **Pulse levels rather than pulse amplitudes** checkbox in the **Graphical editor settings**. If this box is clear, the result is the sum of the outputs. If this box is checked, the last item in the overlapped area sets the output level. There is an exception; arbitrary waveform output overrides all other items.

When adding single pulses and pulse trains where the result would exceed the range of the DAC, the output is limited to the DAC range. However, pulses with an amplitude change on repeats can exceed the DAC range and wrap around. A value that goes off the top of the range will reappear at the bottom; a value that goes off the bottom of the range will reappear at the top.

If items on different channels overlap or get close to the start or end of a section in such a way that the correct timing cannot be maintained, the timing error is indicated with red marks below the control track.

Ramps 

This item can be used with any DAC output. The **From** and **To** fields set the initial and final amplitudes or levels of the ramp depending on the state of the **Pulse levels rather than pulse amplitudes** checkbox in the Graphical editor settings.

In the current implementation, no other activity is allowed while a ramp is generated except a sinusoid. You will find that any item that starts within the time range of a ramp will cause a timing error to be flagged and the item start will be delayed until after the ramp when you run the sequence. This may be changed in a future release.

Sinusoid output 

From Spike2 version 5, sinusoids can be generated on up to four DACs in the Power1401. The remaining 1401s except the 1401*plus* allow sinusoids on DACs 0 and 1. The 1401*plus* supports DACs 2 and 3. In previous versions, you can only use 2 DACs.

The sinusoid amplitude is defined by the **Size (units)** field; this is not affected by the **Pulse levels rather than pulse amplitudes** checkbox. You can offset the sinusoid with the **Centre (units)** field. If the **Pulse levels rather than pulse amplitudes** checkbox is clear, the sinusoid and offset is added to the DAC value. If the checkbox is clear, the DAC output is defined by the sinusoid and offset.

The **Period** field sets the time for one cycle of the sinusoid in seconds or in milliseconds. The **Start phase** field sets the initial phase in degrees. The output is a cosine, so a phase of 0 means start at maximum amplitude. A phase of -90 or 270 produces a sine output.

The full details of the graphical editor and the available items are given in the Spike2 manual and the online help, which should be consulted if you need definitive information.

Sampling, control and the output sequencer

Output sequencer In addition to sampling data, Spike2 is also capable of outputting waveforms and digital signals and controlling other equipment. Output options include:

Digital outputs

1. Spike matches template signal on digital o/p bits 0-7 (dealt with in other sessions)
2. DIGOUT sequencer command sets digital o/p bits 8-15
3. DIGLOW sequencer command sets digital o/p bits 0-7 (micro1401 and Power1401)

DAC outputs

1. DACn and ADDACn sequencer commands set DACs to a value
2. Sinusoidal output from the sequencer (not standard 1401)
3. Ramp output from the sequencer (version 5 only, not standard 1401 or 1401*plus*)
4. Arbitrary waveform output (not standard 1401)

Other outputs

1. Sound card output of .WAV files using the `Sound()` script command
2. Serial line output using the `SerialXXX()` family of script commands.
3. Text and binary files written for other programs to read.
4. Screen images (for example generated with XY views)

The text output sequencer The output sequencer generates accurately timed digital pulses and analogue outputs such as pulses of varying amplitude, ramps and sinusoids. It can also respond to external digital and analogue signals and randomise events.

- It controls 8 output bits (bits 8-15 on the digital output connector). Power1401 and micro1401 users can also set the lower 8 bits (0-7) with the DIGLOW command.
- The sequencer controls the DAC outputs (2 on the micro1401, 4 on other 1401s).
- The sequencer can cause data to be recorded to channel 32 (digital marker channel).
- The sequencer can control arbitrary waveform output.
- The sequencer starts running when sampling starts and is not affected by pauses in sampling or by triggered sampling mode (so the sequencer can trigger sampling).

How does it work An output sequence is a list of simple instructions that are run at a fixed rate set by the output sequencer clock. This clock can run at up to 20,000 ticks per second with the micro1401 and 1401*plus* and up to 100,000 with Power1401. The rate is set by the SET command, which sets the interval between clock ticks in milliseconds. The sequencer executes one instruction per clock tick.

The basic instructions do things like “set digital output bit 0 to 1”, “wait 10 clock ticks” and “set DAC 0 output to 2 Volts”. The following (`A.pls`) illustrates the general idea. A semicolon introduces a comment:

```
SET      100          ;100 milliseconds per tick
DIGOUT   [.....1]    ;set lowest bit high
DELAY    8           ;wait 8+1 clock ticks
DIGOUT   [.....0]    ;set the bit low again
DAC0     2.00        ;set DAC 0
DELAY    8           ;wait another 8+1 clock ticks
DAC0     0.00        ;tidy up the DAC
```

The only non-obvious command here is DIGOUT. Each digital output bit is represented by one character. A dot means make no change, a 1 or a 0 sets the output state high or low. You can also use the letter *i* meaning invert the current state. *You can find the exact syntax of all the output sequencer commands in the manual and in the on-line help.*

Before version 4, this sequence repeats after a delay of some 27 seconds. The output sequencer was 256 instructions long and was circular; if you ran off the end, you restarted at the beginning. Version 4 allows up to 1023 instructions and is not circular, version 7 allows up to 8191 instructions. Version 4 onwards automatically adds a HALT at the end.

```
HALT          ;stop the sequencer running
```

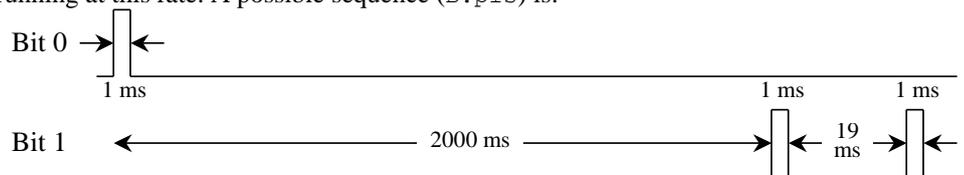
Now suppose we want to repeat this action. We can link our sequence to a keypress, or to a mouse click. Change the first instruction to:

```
'A DIGOUT [.....1] ;set lowest bit high
```

and run again. You will see that the sequencer window now holds the text "A set lowest bit high" and if you run and click on this line, the sequencer starts again from the label.

A simple task

We now know enough to attempt something more useful. Let's ask for a 1 millisecond start pulse on one output followed by a 2 second delay, then two 1 millisecond pulses on a second output with a 20 millisecond gap between the pulse starts. To implement this we can see that the shortest interval we need is 1 millisecond, so we will set out clock running at this rate. A possible sequence (B.pls) is:



```
SET 1.00 ;run at 1 millisecond
'X DIGOUT [.....00] ;Make sure both outputs low
HALT ;and stop >Waiting for user!
'G DIGOUT [.....1] ;Start sequence >Running
DIGOUT [.....0] ;end pulse >Running
DELAY 1997 ;allow for steps used >Running
DIGOUT [.....1.] ;pulse bit 1 >Running
DIGOUT [.....0.] ;back low >Running
DELAY 17 ;wait >Running
DIGOUT [.....1.] ;pulse bit 1 >Running
DIGOUT [.....0.] ;back low >Running
HALT ;done >Waiting for user
```

We have added more text to the lines after a >. When you run this sequence the extra text appears as a comment on the current step, and can be used to prompt the operator.

Loops

The previous solution would not have been so great if we had needed 20 pulses after the delay in place of the 2. Writing the code out 20 times would not have been much fun. Fortunately, there are ways to get around this (C.pls). There are four loop counters you can use for this job, numbered 1 to 4. We will use counter 1. We also need to label the place we want to branch to, so we need to supply a label (indicated by a colon).

```
SET 1.00 1 0 ;run at 1 millisecond
'X DIGOUT [.....00] ;Make sure both outputs low
HALT ;and stop >Waiting for user!
'G DIGOUT [.....1] ;Start sequence >Running
DIGOUT [.....0] ;end pulse >Running
DELAY 1996 ;allow for steps used >Running
LDCNT1 20 ;set times around the loop >Running
LOOP: DIGOUT [.....1.] ;pulse bit 1 >Looping
DIGOUT [.....0.] ;back low >Looping
DELAY 16 ;wait >Looping
DBNZ1 loop ;loop back >Looping
HALT ;done >Waiting for user
```

The LDCNT1 instruction sets the counter value (in this case it is set to 20), and the DBNZ1 instruction (Decrement and Branch if Not Zero) reduces the count by 1 and branches to the label if the counter has not reached 0. You can run round a loop up to 65535 times in version 3. From version 4 you can use variables as loop counters (see the DBNZ instruction in the manual) and loop up to 2147483647 times.

Variables

If you wanted to change the number of times you loop around, the only way to do this with the instructions we have seen so far is to write the code out again. However, from version 3 of Spike2 you can use variables for this task. Variables can also be used in version 2, but there is much less that you can do with them. This sequence (D.pls) is for versions 3 onwards. The standard 1401 cannot use variables.

```

          VAR    V1,Loops=20    ;set variable 1 to 20, call it Loops
          SET    1.00 1 0      ;run at 1 millisecond
'X DIGOUT [.....00]          ;Make sure both outputs low
          HALT                    ;and stop                >Waiting for user!
GO:  'G DIGOUT [.....1]      ;Start sequence    >Running
          DIGOUT [.....0]      ;end pulse        >Running
          DELAY  1996           ;allow for steps used >Running
          LDCNT1 Loops          ;set times around the loop >Running
LOOP: DIGOUT [.....1.]        ;pulse bit 1     >Looping
          DIGOUT [.....0.]      ;back low        >Looping
          DELAY  16             ;wait            >Looping
          DBNZ1 loop            ;loop back          >Looping
          HALT                    ;done                >Waiting for user

```

There are 64 variables from version 4, called V1 to V64 and 16 variables in version 3 (V1 to V16). From version 7, modern 1401s allow 256 variables. You can give them more meaningful names and initial values with the VAR statement. Incidentally, VAR, like SET, is an instruction to the sequence compiler and is not counted as one of the steps in the sequence. Now we can have different numbers of pulses by adding the following lines:

```

'1 MOVI  Loops,1      ;Single pulse
   JUMP  Go           ;Start the output
'2 MOVI  Loops,2      ;Dual pulse
   JUMP  Go           ;Start the output
'3 MOVI  Loops,3      ;Triple pulse
   JUMP  Go           ;Start the output

```

Now you can run the sequence with 1, 2 or 3 pulses after the delay. However, you might want to choose an arbitrary number of pulses while you were sampling data. You can achieve this with a run-time script. The simplest possible would be something like:

```

'SeqVar1.s2s - script to set variable 1 to a user defined value.
var count% := 10;
count% := Input("Number of pulses", count%, 1, 100); 'new value
SampleSeqVar(1, count%); ' Set the sequencer variable number 1
SampleKey("G"); ' Make it run now

```

You can make much fancier scripts using toolbars and control all aspects of sampling too, but this shows the general idea.

Random delays and branches

It is often necessary to produce stimuli in pseudo-random order or after pseudo-random delays. The sequencer has a few instructions to help you achieve this:

LD1RAN (MOVRND is preferred in modern systems) sets counter 1 to a random number in the range 1 to 256. BRAND label,prob branches to a particular label with a given probability. For example, BRAND Go,0.5 will branch to label Go with a 50% probability. You can also branch with a probability based on the value of a variable using the BRANDV label,variable instruction.

There are various ways to produce delays, depending upon the distribution of delays you want. For a uniform distribution:

```

LoopR: LD1RAN                    ;get a random number 1-256
        DELAY  9                 ;scale the delay
        DBNZ1 LoopR             ;loop 1 to 256 times

```

Each time round the loop takes 10 clock ticks, so this produces a delay of $1+10*(\text{random number})$, so there is a delay of between 1 and 2561 clock ticks, with a uniform distribution. Use this to replace the DELAY 1997 in the last script.

The following generates a delay with a mean of $(130*n+2)$ clocks, a minimum of $(2*n+2)$ and a maximum of $(258*n+2)$ clocks. The distribution tends to Normal as n increases.

```

          LDCNT2 10          ;n=10 in this case
DL:      LD1RAN             ;load counter 1 1-256
DM:      DBNZ1  DM         ;count out delay, mean is 128
          DBNZ2  DL         ;count out counter 2

```

The following generates a delay with a Poisson distribution:

```

DL:      DELAY 9            ;scale the delay
          BRAND DL,0.99

```

Beware that BRAND can only produce a probability to an accuracy of one part in 256, however, BRANDV is much more accurate.

Testing external conditions

You can test the state of 8 digital input bits from the sequencer. These are the digital bits that are labelled 0-7 on the digital input connector. These are the digital bits that can be programmed to be inputs or outputs in the standard 1401 and 1401*plus* (and which are programmed as inputs in Spike2 unless the spike shape code is set to use them as outputs). The Micro1401 has separate input and output bits, so these inputs are always available.

The current input state is tested with the DIGIN [pattern] instruction or WAIT [pattern]. The pattern is 8 characters being 0, 1, c or a . (dot). The input is read and each bit is compared with the value in the pattern. If the pattern is a 0 or 1, the result for that bit is 0 if the pattern matches the bit and 1 if it doesn't. If the pattern is a dot, the result for the bit is 0 (this means that dot means we don't care what state this bit is in). If the pattern is a c, the result is a copy of the input bit. The output sequencer remembers the result of the last DIGIN and you can branch based on the result being zero or non-zero using the BZERO label and BNZERO label instructions.

WAIT holds the sequence at this instruction until the input matches the pattern.

```

; BRAND.PLS randomly chooses one of three possibilities. This implements
; an experiment where the subject is shown one of two LEDs or both LEDs.
; If one LED shows, the subject must press a key, if both the subject must
; NOT press a key. Digout bits 7 and 6 control LEDs, data ports 7
; and 6 (the digital input bits 15 and 14), and dig in bits 7 and 6 (the
; bits tested by WAIT and DIGIN)
          SET 1.00 1 0      ;run at 1 ms
TEST: 'G WAIT [11.....]   ;Start          >unpress!
          DIGOUT [00.....] ;LEDs off       >Waiting..
          DELAY 1000       ;Start          >Waiting..
          LDCNT1 333       ;set time out   >Waiting..
AA:      BRAND aa,0.9922   ;rand wait    >Waiting..
          BRAND bit7,0.3320 ;button A test
          BRAND bit6,0.5000 ;button B test
;
; Come here to test BOTH buttons (NO response)
BOTH:    DIGOUT [11.....]  ;both LEDS on  >BOTH
          MARK 3           ;signal test both >BOTH
WAITB:   DIGIN [11.....]  ;must match   >BOTH
          BNZERO BFAIL    ;fail if no match >BOTH
          DBNZ1 WAITB     ;allow 1 second >BOTH
;
; Here for a Pass!
PASS:    MARK 0           ;code for pass  >PASSED!
          JUMP TEST
;

```

```
; here to fail both test
BFAIL:  MARK    131          ;128+3, fail both >FAIL
BFLOOP:  DELAY   2           ;           >FAIL
        DBNZ1   BFLOOP      ;keep same timing >FAIL
        JUMP    TEST
;
; Test bit 7 only
BIT7:    DIGOUT [10.....]    ;LED on           >A
        MARK    2           ;flag bit 7 test >A
BIT7L:   DIGIN  [11.....]    ;see if anything >A
        BNZERO  BIT7TRY      ;           >A
        DBNZ1   BIT7L        ;           >A
NORESP:  MARK    128        ;No response error >FAIL
        JUMP    TEST
BIT7TRY: DIGIN  [01.....]
        BZERO   PASS
        JUMP    BFAIL
;
; Test bit 6 only
BIT6:    DIGOUT [01.....]    ;LED on           >B
        MARK    1           ;flag bit 6 test >B
BIT6L:   DIGIN  [11.....]    ;see if any change >B
        BNZERO  BIT6TRY      ;           >B
        DBNZ1   BIT6L        ;           >B
        JUMP    NORESP
BIT6TRY: DIGIN  [10.....]    ;see if correct
        BZERO   PASS
        JUMP    BFAIL
```

The script uses the Poisson distribution method to generate a random delay. The number 0.992 comes about because we can only accurately generate probabilities which can be represented as an integer divided by 256, in this case 254/256.

The stimulus is chosen by branching with a probability of 1/3 to the button A test (0.332 is as close as we can get to 0.333, being 85/256). In case you are wondering, I didn't have to work this out myself, I just typed in 0.333 and the asked the sequencer to tidy itself up and it works out the values that are really going to be used.

The next step is a branch with a probability of a half to button B or to the both buttons. The end result is that the chance of getting to any of the three tests is 1 in 3 (or as close to this as makes no difference). The code for each of the three tests is similar, here is the code to test for both lights on:

```
; Come here to test BOTH buttons (NO response)
BOTH:    DIGOUT [11.....]    ;both LEDS on     >BOTH
        MARK    3           ;signal both      >BOTH
WAITB:   DIGIN  [11.....]    ;must match       >BOTH
        BNZERO  BFAIL        ;fail if no match >BOTH
        DBNZ1   WAITB        ;allow 1 second   >BOTH
;
; Here for a Pass!
PASS:    MARK    0           ;code for pass    >PASSED!
        JUMP    TEST
;
; here to fail both test
BFAIL:   MARK    131        ;128+3, fail both >FAIL
BFLOOP:  DELAY   2          ;           >FAIL
        DBNZ1   BFLOOP      ;keep timing the same >FAIL
        JUMP    TEST
```

We start by turning both lights on and we use the MARK instruction to record the fact. To use this instruction usefully you must have enabled the digital marker channel (channel 32) in the sampling configuration. The instruction simulates a digital marker and gives it a code in the range 0 to 255. In this test we use code 1 for a button A test, code 2 for a button B and code 3 for both.

We now start testing the inputs and we will fail the user if they press either button. Counter 1 has already been loaded with the value 333 and there are 3 instructions in the

test loop, so the user has approximately 1 second to respond in (or in this case, to not respond in). If the response time was an important part of the test, you could wire the signals from the buttons to level event channels so that you could measure these times.

If the user passes the test, we store the code 0 in the digital marker channel, if the test fails we store the code 131. If the user should respond and doesn't, we store the code 128. This means that bit 7 of the code is set for a failure.

The job of writing the script to analyse the results of the test is left as an exercise for the student...

Communicating with the sequencer

If you do not run a script while capturing data, the only way to control the sequencer actions is to make it jump to a specific location in the sequence. You can track what the sequence is doing by looking at the sequencer control panel (as long as you have written suitable messages in your sequence), by connecting outputs from the sequencer to input data channels and viewing the result, and by using the `MARK` or `REPORT` instructions with the digital marker channel.

If you run a script, there are more ways to control it. You can make it jump to a known location with `SampleKey()` which simulates a keypress, and you can also set the values of sequencer variables with `SampleSeqVar()` and (from version 3) you can read back the current value of a sequencer variable using the same command.

Voltage pulses

A very common requirement is to produce a series of pulses with a given amplitude, duration and separation. The next example does this and also lets you change the pulse parameters. The output sequence used is the following (`varex.pls`):

```

                SET      1.0 1 0      ;run at 1 millisecond, DACs in Volts
                DAC0    0.0          ;make sure pulse starts from 0
                HALT                    ;Wait for command >Waiting
PULSE:'G DAC0    0.0          ;Make sure DAC at zero
                LDCNT1 V4          ;number of pulses
L0:            DAC0    V2          ;set pulse height >High
                DELAY   V3          ;length of pulse-2 >High
                DAC0    0.0          ;remove pulse >Low
                DELAY   V1          ;length of gap-3 >Low
                DBNZ1   L0          ;loop round >Low
                HALT                    ;done >Waiting

```

This example uses the variables `V1` to `V4` as follows:

- V1 The gap in clock ticks between the end of a pulse and the start of the next. By looking at the sequence you can see that the gap will be 3 clocks longer than this, so this variable must be set to the required number of clock ticks minus 3.
- V2 The amplitude of the pulse. This is in 1401 DAC units, so it must be scaled. Most 1401s have a ± 5 Volt DAC output range. The DACs are set to the value -32768 for -5 Volts, to the value 0 for 0 Volts, and up to 32767 for up to 4.9998 Volts. However, most 1401s have 12-bit DACs, which means that the maximum useful value is 32752 for 4.9975 Volts. If we have a value in millivolts to output, we must multiply it by 32768/5000 to convert it into DAC units, which is 6.5536.
- V3 The length of each pulse in clock ticks. Looking at the code we can see that the pulse is in fact 2 ticks longer, so we must set the variable to a value 2 ticks less.
- V4 The number of pulses.

This example does not use variable names or initial values (added at version 3), so it will run with any version of Spike2. However, we must use a script to set the variable values:

```

'varex.s2s
var vh%;           'will hold the sampling view
var pulAmp:=2000.0; 'pulse amplitude (mV)
var pulWid%:=20;   'pulse width in (ms)
var pulInt%:=100;  'gap between pulses (ms)
var pulNum%:=5;    'number of pulses
ToolbarClear();    'Remove any old buttons
ToolbarSet(1, "Quit", OnQuit%);
ToolbarSet(2, "Settings", OnSettings%);
ToolbarSet(3, "Output", OnOutput%);
Labels(0);         'set labels for not sampling
Toolbar("Sequence variables demonstration", 231);
halt;

Func OnQuit%()     'User wants to exit
if SampleStatus()>=0 then SampleStop() endif;
return 0;          'Cancel the toolbar
end;

Func OnSettings%() 'Change pulse settings
DlgCreate("Pulse settings"); 'Start new dialog
DlgReal(1, "Amplitude (mV)", -5000.0, 4997.5);
DlgInteger(2, "Width (ms)", 3, 1000);
DlgInteger(3, "Separation (ms)", 3, 10000);
DlgInteger(4, "Pulse count", 1, 100);
if DlgShow(pulAmp, pulWid%, pulInt%, pulNum%) then SetVars() endif;
'update if changed
return 1;         'This leaves toolbar active
end;

Func OnOutput%()  'trigger sequence
SampleKey("G");   'Letter G starts pulse
return 1;         'Leave toolbar active
end;

Func OnStop%()
SampleStop();     'stop sampling
CloseView();      'Close view
Labels(0);        'prepare for not sampling
return 1;         'Leave toolbar active
end;

Func OnSample%() 'Start to sample...
SampleClear();    'default settings, no channels
SampleWaveform(1, 0, 1000); '1000 Hz on ADC chan 0
SampleSequencer("varex.pls"); 'select sequencer
vh% := FileNew(0, 3); 'Open with user config
if (vh%>0) then 'if ok
    if SampleStart()>=0 then
        SetVars(); 'set variables
        Labels(1); 'prepare labels for sampling
    else CloseView(); 'There was a problem...
    endif;
endif;
return 1;         'Leave toolbar active
end;

Proc Labels(Sampling%)
if Sampling% then
    ToolbarSet(4, "Stop", OnStop%)
else
    ToolbarSet(4, "Sample", OnSample%)
endif;
ToolbarEnable(3, Sampling%);
end;

```

```

Proc SetVars() 'Update variables
if SampleStatus() >= 0 then 'if sampling...
  SampleSeqVar(1,pulInt%-3); 'set interval
  SampleSeqVar(2,pulAmp*6.5536);
  SampleSeqVar(3,pulWid%-2); 'length
  SampleSeqVar(4,pulNum%); 'count
endif;
end;

Proc CloseView()
if vh%>0 then View(vh%); FileClose(); vh% := 0 endif;
end;

```

The exact details of the script are not important, but `Proc SetVars()` is where the variables are set in the sequencer. If you use variables, it is important to give them values before running the output sequence. If you do not have version 3 or 4 of Spike2 where you can use the `VAR` statement to initialise the variables, you must arrange for a script to do it, otherwise all variables will start with the value 0, which is usually not very useful.

Sinusoidal output

The output sequencer can also handle up to two channels of sinusoidal output. In this case, you do not specify the output for each clock tick. Instead, you specify the rate at which the output is to run and the amplitude and phase of the sinusoid.

You set the speed of the output by setting the angle increment in degrees per clock tick. The outputs are from DACs 3 and 2 for the standard 1401 and 1401*plus* and from DACs 1 and 0 for the micro1401 and Power1401. The simplest way to start output is:

```

CSZ      1.0          ;set amplitude, 1=maximum 0=minimum
CRATE    .6          ;set the degrees per tick

```

To stop the sinusoidal output set the rate to 0. You can also set the phase angle with `CANGLE`. The output is the cosine of the angle, so an angle of 0 produces the maximum output, 90 produces 0 and so on. If you want to make an output that starts at zero and increases, set the phase angle to 270 to start with. The following example (`freq.pls`) shows how you can produce several frequencies.

```

          SET      1.00 1 0      ;1 kHz
'h CSZ      1.0          ;Halt sequence      >Wait for input
   CRATE    0            ;Stop sinewave     >Wait for input
   CANGLE   270         ;Start 0 amplitude >Wait for input
START:    JUMP     START      ;hang about   >Wait for input
'1 CRATE    3.6         ;Stimulate 10 Hz
WAIT1:    JUMP     WAIT1      ;
'2 CRATE    7.2         ;Stimulate 20 Hz
WAIT2:    JUMP     WAIT2      ;
'3 CRATE    10.8        ;Stimulate 30 Hz
WAIT3:    JUMP     WAIT3      ;
'4 CRATE    14.4        ;Stimulate 40 Hz
WAIT4:    JUMP     WAIT4      ;
'r CRATE    0           ;Ramp the dac
   DAC1     -5
WAIT5:    ADDAC1 .1       ;
          JUMP     WAIT5      ;

```

This example also shows you how to generate a ramp from a DAC. This uses a new instruction `ADDACn` that adds an increment to `DACn`.

The second channel of sinusoidal output is enabled with a similar set of commands, but they have the initial letter `D` in place of the `C`. It is possible to run the sequencer at rates up to 10,000 ticks per second, so it is possible to make useful audio tones up to two or three kHz with this, but you must filter the output to remove the 10 kHz steps.

DAC scaling

You will notice that some of the examples have additional numbers after the time setting in the `SET` command. The full syntax of the command is:

```
SET ms, DACscale, DACoffset
```

`ms` is the milliseconds per clock tick. It is in the range 0.01 to 3000 in steps of 0.001 for the Power1401 and Micro1401 mk 2, 0.05 to 3000 in steps of 0.01 for other 1401s.

The `DACscale` and `DACoffset` fields are used to convert from Volts (assuming the 1401 is set as a ± 5 Volt system) into units of your choice. Your units are defined as:

```
Units = Volts * DACscale + DACoffset
```

The normal values are 1.0 for `DACscale` and 0.0 for `DACoffset`. To enter DAC values in mV you would set `DACscale` to 1000.

Arbitrary waveform output

The most recent addition to the arsenal of output options is arbitrary waveform output. You can define up to 10 areas of 1401 memory to hold waveforms that are played through the 1401 DACs. Each area has its own list of DAC channels, sample rate and cycle count (number of repeats). You can also change the replay rate of each area.

Compatible areas can be linked so that when one area finishes output, the area it is linked to starts to play with no gap. To be compatible, areas must have the same list of DAC channels. They need not have the same replay rate, but if they do not, when an area that is linked to plays, it runs at the output rate of the previous area.

Areas can be replayed from a toolbar or from an on-line script or from the output sequencer. Some dynamic variation of areas is possible during replay using the script language and the output sequencer:

1. A script can change the replay rate of an area while it plays. The rate change can be either immediate or delayed until the current area completes the current cycle.
2. A script can update the replay data during output, allowing you to replay waveforms too large for 1401 memory or generate waveforms that relate to sampled data.
3. The way that areas are linked together can be changed while areas are replaying.
4. It is possible to replay an area at a precisely controlled time from the output sequencer so as to synchronise the waveform output with other stimuli.
5. The output sequencer can make an area that is playing multiple times do one more cycle then stop or continue with the next linked area if there is one.

Creating waveforms for arbitrary output

The simplest way to generate an arbitrary waveform is to open a spike2 data file containing waveform data, select a region to play, then add this to the on-line play list. You can add waves as either a reference to the data file or convert the reference into the binary data that is played. The list of waveforms for output is held in memory and saved in the sampling configuration, so it saves memory and disk space if you leave the data as a reference to the data file. However, when you come to replay the data, this data file must still be in the same place and must hold the same channels.

There is a limit of 2 MB per area on the size that can be saved. The waveforms are copied to the 1401 each time you start to sample, so if you have several big waves to copy (and your 1401 has lots of memory) this can slow down the start of sampling.

You can also generate arbitrary waveforms with the script language `PlayWaveAdd()` command which allows you to generate a wave from a data file, or from an array of data, or which lets you reserve an area of memory in the 1401 that you will fill up with the `PlayWaveCopy()` command during sampling. You can also use `PlayWaveCopy()` to update areas dynamically during output.

An example Suppose that we want to replay a blood pressure signal to drive an experiment and we have a slow heart rate waveform and a faster heart rate waveform. We generate one cycle of each, either by selection from a data file or by generating them with a script. We also make sure that the waveforms join together smoothly.

Now if we set each waveform to play a large number of times, and then link them both to each other, we have a system in which we can swap between two waveforms by using the output sequencer `WAVEST C` command to cause the currently playing wave to finish the current cycle, then swap to the other signal. The configuration file `bp.s2c` and the sequencer file `bp.pls` illustrate this.

Alternatively, we could use the script `PlayWaveSpeed()` command to change the effective heart rate by slowing down or speeding up the replay rate.

Arbitrary waveforms and the sequencer Three sequencer instructions control arbitrary waveform output: `WAVEGO`, `WAVEST` and `WAVEBR`.

WAVEGO The `WAVEGO` instruction starts the output from a play wave area, or arms the output ready for an external trigger. Because starting output can take more time than we can allocate to a single step of the sequencer this command sets a flag and the next time the 1401 has time to start the playing operation (usually within a millisecond), the playing is started. It is possible to get a precisely timed start by requesting that an area start on a trigger, then using the `WAVEST` instruction to trigger the wave. The `WAVEGO` instruction has the following format:

```
WAVEGO code{, flags}
```

code This is either a single case sensitive character standing for itself, or a two digit hexadecimal code. This is the code of the area to play.

flags These are optional single character flags. The flags are not case sensitive. If a `T` is present, the waveform output is triggered. If a `W` is present, the sequencer will loop at this step until background code has set the hardware ready to play. For example:

```
WAVEGO X           ;area X, no wait, no trigger
WAVEGO 23,T        ;area coded hexadecimal 23, triggered
WAVEGO 0,WT        ;area 0, wait until trigger armed
WAVEGO 1,W         ;area 1, wait until play started
```

WAVEST The `WAVEST` instruction can start output that is waiting for a trigger and stop output that is playing, either instantly, or after the current cycle ends.

```
WAVEST flag
```

flag This is a single character flag and specifies the action to take:

- `T` Trigger a waveform that is waiting for a triggered start.
- `S` Stop output immediately, no link to the next area.
- `C` Play the current cycle, then end this area. If there is a linked area, it will play.

The following code starts output with an internal trigger and then stops it

```
WAVEGO X,TW        ;arm area X for trigger
WAVEST T           ;trigger the area
DELAY 1000         ;wait
WAVEST S           ;stop output now
```

WAVEBR The `WAVEBR` instruction tests the state of the waveform output and branches on the result. No branch occurs if there is no output running or requested.

```
WAVEBR LB, flag
```

LB	Label to branch to if the condition set by the flag is not met.
flag	This is a single character flag and specifies the condition to branch on. The conditions are:
W	branch until a WAVEGO request without the W flag is complete.
C	branch until the play wave area or cycle count changes.
A	branch until the play wave area changes.
S	branch until the current output stops.
T	branch until output started with WAVEGO begins to play.

The following sequence tracks the output when we have two play areas labelled 0 and 1. Area 0 is set to play 10 times and is linked to area 1. The sequence below will track the changes. You must be aware that the DAC output happens one DAC clock tick after the output is changed, so the sequence will know that a DAC output is about to change.

```

                WAVEGO 0,WT          ;area 0, wait for armed, trigger start
                LDCNT1 5             ;load counter 1 with 5
WT:             WAVEBR WT,T         ;wait for external trigger>Trigger?
W5:             WAVEBR W5,C         ;wait for cycle >Waiting for cycle
                DBNZ1 W5            ;do this 5 times >Waiting for cycle
WA:             WAVEBR WA,A         ;wait for area >Wait for area
WE:             WAVEBR WE,S         ;wait for end >Wait for end
    
```

The WAVEGO command requests a triggered start and waits until the trigger is armed before moving on. It then waits for an external trigger at the WT label. Next the sequence tracks the end of 5 output cycles. At label WA the sequence waits for the area to change and finally the sequence waits for output to stop. If you need to know when a requested play has started to produce output, use the WAVEBR T option. If you need to know that the request to start the playing operation has been honoured, but you do not want to hang up the sequencer with the W option, use the WAVEBR W option.

If a waveform is playing when you use the WAVEGO command, the output will be cancelled just before the requested area starts to play. If you want to use a WAVEBR instruction after the play request, unless you use the WAVEGO or WAVEBR W option to be certain that the new area is active, the result of the WAVEBR may be based on the previous area, not the new one.

Versions 4 - 7 The output sequencer was rewritten for Spike2 version 4 to make it a little faster by removing the restrictions imposed by the standard 1401 (version 4 does not support the standard 1401). All previous sequences should run in version 4. The main changes are:

- Sequences can be up to 1023 steps long, 64 variables and used in more instructions
- New, more efficient instructions replace DIGIN, BZERO and BNZERO
- Restrictions on number sizes removed, e.g. DELAY 1000000 allowed
- Expressions and utility functions converts seconds to steps and Hz to angle increments
- The phase of the sinusoidal output can be changed and the output can be offset
- Extensions to the maths functions, many instructions have optional branch

Version 5 Further features were added in version 5:

- Indexed table of values holds long sequences of numbers
- Power1401 supports 4 sinusoids
- Automatic ramping of DACs (not 1401*plus*)
- Sequencer now supports division with new DIV and RECIP commands.

Version 7 Many new features were added, include files, longer sequences, more variables, dynamic loading of sequences and much more. See the version 7 release notes for all the changes.

Spike shapes

Spike shapes

Spike2 captures spikes as short waveform sections detected by threshold crossing. The 1401 can detect and sort incoming spikes in real time giving the host computer more time to display and analyse data. Instead of identifying spikes based on amplitude alone, we capture the time of the spike plus a number of accompanying waveform data points that describe its shape. You can capture up to 32 spike shape channels with a Power1401, up to 16 with a micro1401 mk II and up to 8 with the original micro1401 or a 1401*plus*. Spike shape capture is not supported by the standard 1401.

Channel holding this type of data are called WaveMark channels. Once WaveMark data has been captured it can be re-sorted automatically or manually off-line. A whole chapter of the Spike2 manual is devoted to the details of WaveMark data capture and analysis. From version 5, you can use clustering techniques to separate spikes. The images in this section are taken from Spike2 version 4.

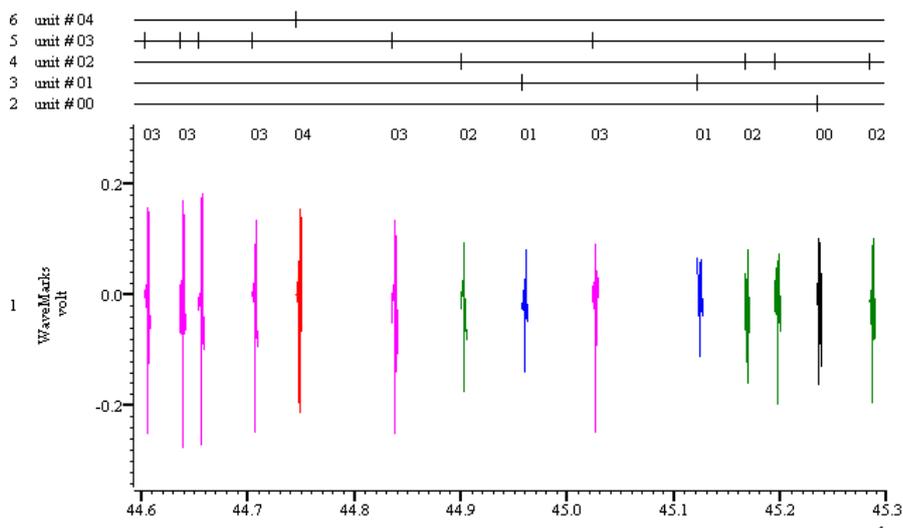
Why use WaveMark data?

Spike data usually needs sampling at rates of 20 kHz or more in order to get reasonable resolution of the spike shape. If computer systems could handle gigabyte sized files as easily as kilobyte sized files, the best approach would be to keep all the data. However, each channel of this data would use up around 2.5 MB of disk space a minute, and as files get larger, it takes longer to process them.

When you use WaveMark data, you set trigger levels, and capture short sections of waveform around the points where the data crosses the trigger. The disadvantage of this method is that you will not be able to go back and look at events that were smaller than the trigger. However, the advantage is that with normal unit firing rates, the resulting files can be 20 to 100 times smaller and you can code each event to aid analysis.

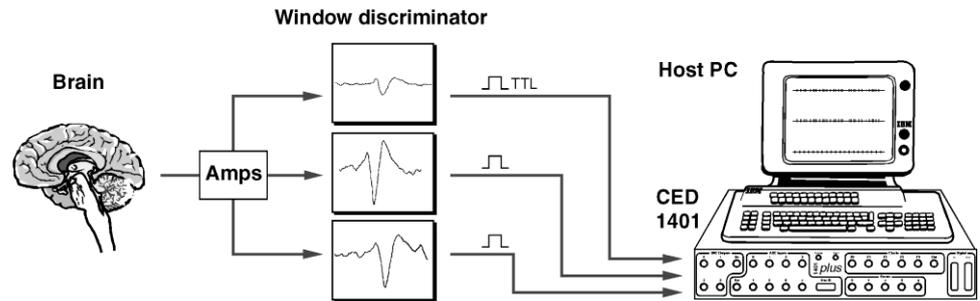
WaveMark data is not as space efficient as using a window discriminator to convert data into TTL pulses and recording this in an event channel. For example, if you stored 32 data points per spike, WaveMark data uses 18 times as much disk storage per spike as an Event+ or Event- data channel. If your input data is known to come from one spike and is easily detected by a window discriminator, this may be your best choice.

However, most window discriminators can't separate multiple spikes in the same channel. It would be necessary to have more, relatively expensive, window discriminators set to different amplitudes/duration and the output would be fed in to more digital channels at the 1401. The illustration below contains one channel of units in WaveMark form and several channels containing the equivalent window discriminated units.

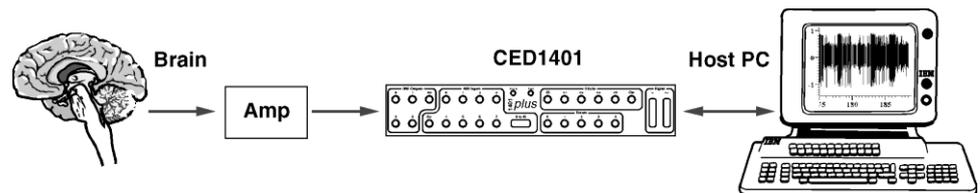


The next picture shows a window discriminator set-up and below that an equivalent Spike2 / WaveMark combination.

Window discriminator set-up



WaveMark set-up

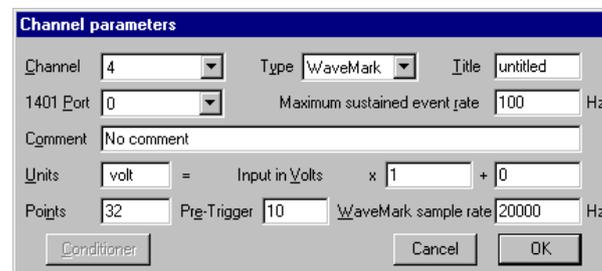


Sampling rates of WaveMark data

A high sampling rate is required if spikes (of typical duration 1 to 2 milliseconds) are to be usefully discriminated therefore a sampling rate of 20 kHz is often used. From Spike2 version 4 you can set the desired sample rate in the Channel parameters dialog. Prior to version 4, the WaveMark sample rate depended on the number of waveform and Wavemark channels. The total number of data points to capture for each spike can be set in the range 6 to 126.

Although we have achieved a sampling rate sufficient to capture the WaveMark make sure that the incoming waveform is of a reasonable amplitude. If a signal is say +/-0.5 Volts it only has 1/10 of the resolution that the 1401 inputs can accept.

The dialog below shows how the user might set-up one channel of WaveMark capture. This image is from version 3. In version 4 you can set the ideal WaveMark sampling rate.



This is available from the sampling configuration menu. The user has selected a capture rate of approximately 20 kHz for the channel and will describe the incoming spike with 32 data points, 10 of which are before the trigger. The number of points can also

be changed at the template set-up stage. The maximum sustained event rate is set to 10 in this configuration. This affects the amount of memory inside the 1401 that is allocated to the incoming spikes; too high a mean rate results in less efficient system performance.

Hardware required

Spike sorting is available on-line when using, a Power1401, a Micro1401 mk II, a micro1401 or a 1401plus; it is not possible to capture WaveMark data using a Standard 1401. It is possible, however, to discriminate spikes off-line (See off-line analysis, create new WaveMark channel.)

On-line set-up

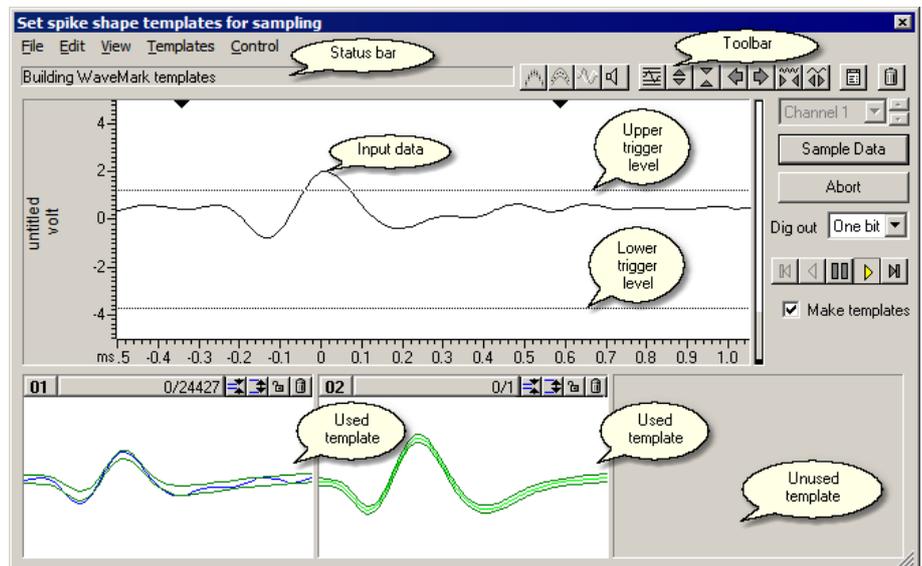
When a new data document is opened to capture WaveMark data you are presented with the template set-up window (below) prior to sampling. It is here that we construct the templates to match the incoming data. User-defined parameters influence the envelope of each template and tolerance to variations in size and shape. The templates generated can be selected by the user and merged with others if the user feels they are too similar. Once this stage has been completed the actual recording can take place and the incoming spikes are checked against existing templates, sorted and displayed accordingly.

The spike detection algorithm

1. Wait for the signal to lie within half the two trigger levels. When it does go to 2
2. If the signal crosses either one of the trigger levels (one for positive and one for negative going spikes) then go to 3.
3. Track the negative or positive going peaks.
If we have a new peak (or trough) restart the peak (or trough) search.
If we have sufficient post-peak data to complete the spike, go to step 5.
If the waveform level is below the baseline, then go to step 4.
4. Wait for sufficient points to complete the spike, then go to step 5.
5. Save the spike and go to step 1.

The template setup window

This window displays the incoming waveform data plus any spikes that cross the triggering thresholds. The first eight templates are displayed in the lower half of the window. At the top of the window is a toolbar, which controls template formation:



	Overdraw	Switches between overdrawing the spikes in the template or displaying the last spike.
	Template outlines	This switches the template border surrounding the spikes on and off.
	Show unmatched	Overdraw the raw spike data on all templates to see how similar it is to different templates.
	Sound	Switches sounds associated with templates on and off/
	Best guess	A good starting point for determining the threshold levels. The computer takes over at this point and calculates the best levels for both the trigger and base measurements. This can aid manual adjustment.
	Scale data	These two buttons increase and decrease the y axis for easier placement of the threshold and base levels.

	Scroll	These buttons change the pre-trigger time. You can also click and drag the time axis to change the pre-trigger time.
	Points	You can increase or decrease the number of data points that will be captured when writing to disk. You can also do this by clicking and dragging the number in the time axis.
	Time range	Off-line only. This opens a dialog in which you can set the time range to process.
	Clear templates	This button throws away all confirmed and provisional templates. There is a similar button to delete an individual template at the top of each confirmed template shown.
	Parameters	This button opens the parameter set-up dialog shown below.

The template parameters dialog

New Template

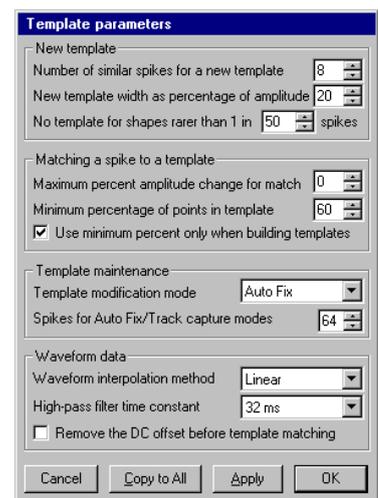
This dialog is available in all on and off-line spike shape sections. It controls how templates are created.

Number of spikes for a new template

This controls the creation of new templates.

New template width as a percentage of amplitude

This sets the number of spikes at which a provisional template is promoted to a real one; 8 is a reasonable starting value.



This is the percentage of the spike amplitude to give the initial (and minimum) template width. As spikes are added to the template, the width changes to represent the variation in amplitude of the spikes in the template. The maximum template width is set to 4 times the minimum width. A value of 30% is a reasonable starting value.

No template for shapes rarer than 1 in n spikes

If this is n, this is roughly the same as saying that you are not interested in spike classes which occur less often than once every n total spikes. If you want to keep all spikes as potential templates you should set this to a large number; 50 is a good starting value.

Matching a spike to a template

The items in this group are used when comparing a new spike with the existing templates to determine if the new spike is the same or should start a new provisional shape. As well as the conditions mentioned below there is always a limit to the error between a spike and a template (unless amplitude scaling is enabled) to prevent totally ridiculous results and to avoid wasting time interpolating spike shapes for data that could never fit a template.

Maximum percent amplitude change for match

Spike2 will scale spikes up or down by up to this percentage to make the area under the spike the same as the area under each target template. This is very useful if you have spikes that maintain shape but change amplitude. **Do not set this non-zero on-line unless you need to as it slows down the template matching process.** The maximum change permitted is 100%, which allows a spike to be doubled or halved in amplitude. Set this to 0 unless you need it. The value you set depends on the amplitude variability of your spikes; 25% is a reasonable starting value.

Minimum percentage of points in template

The percentage of a spike that must lie within a template for a match. If more than one template passes this test, spikes are matched to the template with the smallest error between the mean template and the (scaled) spike. 60% is a reasonable starting value. If you enable amplitude scaling you may want to increase this value to 70% or more.

Use minimum percentage only when building templates

The template width is most useful in the setup phase when we are looking carefully at differences between spikes. Once templates are established it can sometimes be better to match to the template with the smallest error and ignore the width. If you check this field,

this sets the percentage of points that must lie in the template to 0% unless you are building templates.

Template maintenance This group of fields determines how the shape of a template changes as more spikes are added to it. On-line, the template shape is fixed in all modes apart from **Track**. From our experience, **Add All** and **Auto Fix** are the most useful modes.

Template modification mode

Add All	All spikes that fit the template are added and modify the template. The effect of each spike becomes smaller as the number of matched spikes gets larger.
Auto Fix	The template is fixed once a set number of spikes have been added. If you have several similar spikes, using Auto Fix after a fairly small number of spikes can stop a template gradually changing shape and becoming the same as another template.
Track	The template shape tracks the spikes. The contribution of each spike to the template decays as more spikes are added. This is only really useful for slow changes in spike shape and always brings with it the danger that all your shapes will merge together. It also slows down on-line spike classification.

Spikes for Auto Fix/Track capture modes This sets the number of spikes for the previous field in **Auto Fix** and **Track** modes. In **Track** mode, the smaller the number, the more rapidly the template shape changes.

Waveform data The final group of fields control how the raw waveform data is processed into spikes:

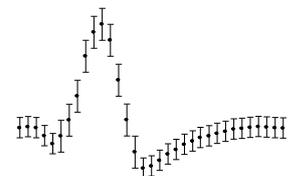
Waveform interpolation method Spike waveforms are shifted by fractions of a sampling interval to align them with templates using linear or parabolic interpolation. Parabolic should be slightly better, but is slower. The on-line 1401 code always uses linear interpolation for speed reasons.

High-pass filter time constant Use this to remove baseline drift. Set this to a few times the width of the spikes. Do not set the value too low or it will significantly change the spike shapes. If your signal has abrupt baseline changes, you may get better results with the DC offset option.

Remove the DC offset before template matching Check this field to subtract the mean level from each spike before matching. This effects template formation and matching, not the saved data. Unless your baseline has sudden DC shifts, it is usually better to use the high-pass filter to follow signals that drift. There is a small time penalty for using DC offset removal on-line.

Template formation Spikes trigger on a positive or negative threshold and are aligned on the peak (or trough) value. The first spike is considered as a provisional (hidden) template and the starting width is calculated based on the values in the parameters menu. New spikes are compared against real and then provisional templates. If a spike matches a template it is added; if it matches no template it forms a new provisional template. When the number of spikes in a provisional template reaches a pre-set threshold (*Spikes for a real template*) the template is promoted to a real template.

The diagram illustrates the allowed percentage amplitude between the spike and the template to which all the other spikes are compared. This amplitude value can be determined by the user in the template parameter menu.

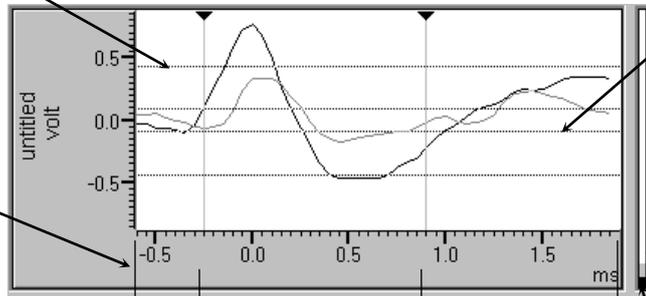


You can change the template area by moving the two small triangles at the top of the raw data view. A common mistake is to include a lot of baseline in the template. Ideally you want to compare spikes on the sections that are most different between spike classes; baseline regions, by definition, are very similar.

Threshold levels. These set the positive or negative levels that spikes must reach to be considered as a spike

To reduce the area given to the oscilloscope view we can use the  button. This will result in the spike rather than the noise dominating and this optimised area is what will be recorded to the data file. Careful selection of the area of interest can reduce the amount of data that is written to disk, improves the spike sorting and enhances the speed of the system.

Area of the waveform for capture to disk



Base levels. Spikes must start inside these levels. In version 4 these are omitted and 1/2 of the trigger levels set the base levels.

This lighter section indicates the number of provisional templates being considered

The area of the waveform to be considered for templating

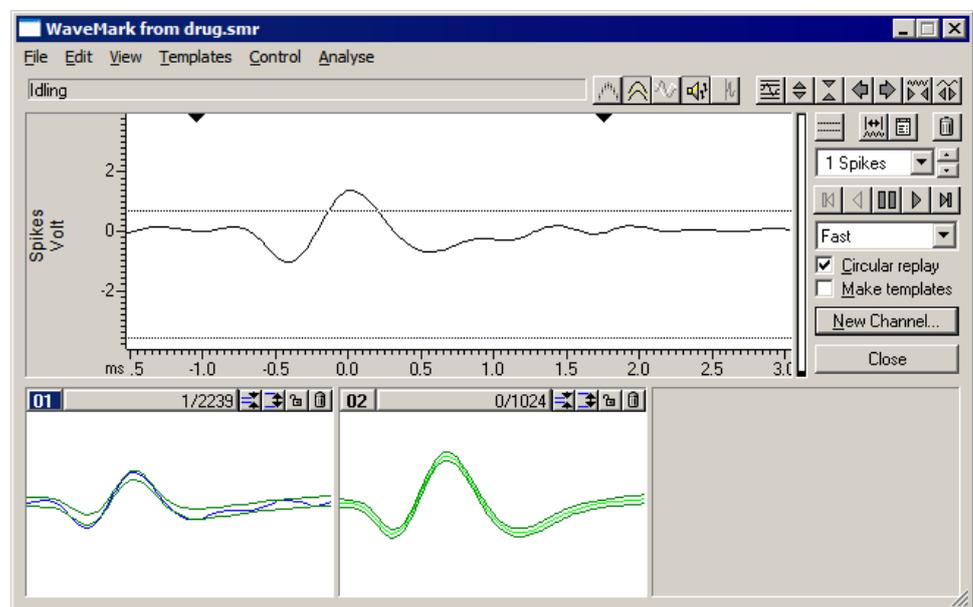
This dark section indicates the number of confirmed templates

Off-line analysis

In addition to capturing spike shapes directly, you can extract them off-line from waveform channel (or even WaveMark channels) and edit them to classify spikes that ended up as code 00 on-line or even resort the spikes completely.

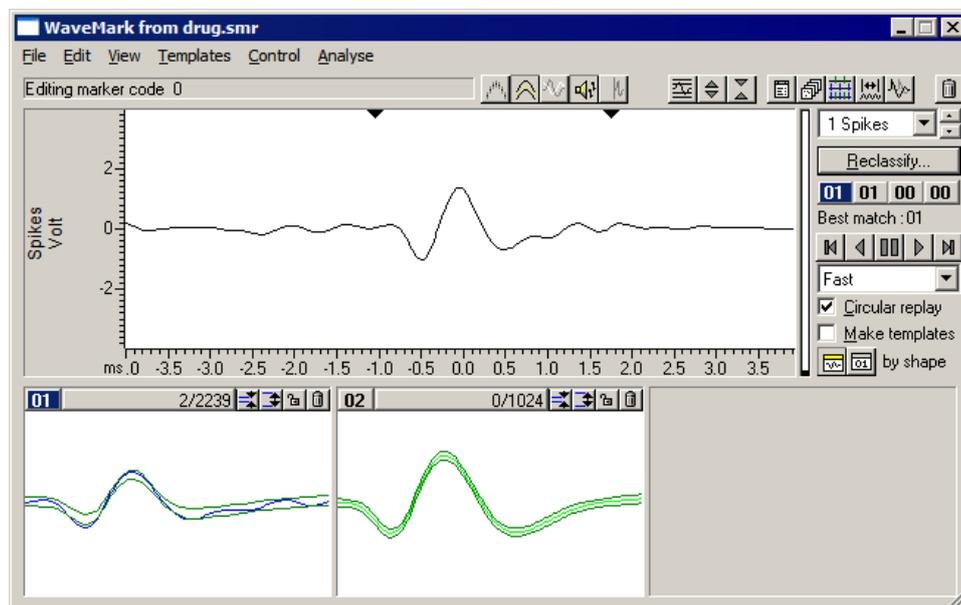
Create new WaveMark channel

It is possible to extract WaveMark data from a waveform channel or from an existing WaveMark channel. This is available from the Analysis menu New WaveMark command. This function acts in the same way as the on-line WaveMark creation by setting up templates for use on the data to follow. The set-up view is very similar to the on-line section but instead of sample data we are given an option to write the new WaveMark data to a new channel.



Edit WaveMark channel This off-line option allows the user to modify their selection of classified spikes. You can move through the data file spike-by-spike or free play in either direction by means of the tape recorder style buttons.

There is also a cursor in the data file that indicates the position in the file that the user is currently considering. You can drag this cursor with the mouse to any position in the data file. When released, the cursor jumps to the nearest spike and this is displayed in the oscilloscope view. You can drag and drop the current spike into a template window to create a new template based on that spike.



With the reclassify option you can re-template the data files spikes against newly constructed templates. In version 4, the **Duplicate** button generates a duplicate channel for each template marker code and sets the marker filter so that each duplicate channel shows only one class of spikes.

Event draw modes There are a number of different ways that we can display our WaveMark (or event) information. When using the draw modes it can be useful to duplicate the so that both the raw and the processed data are displayed simultaneously. The draw modes include:

Rate This type of display counts how many events fall within a time period (selected by the user) and displays the result in histogram form. This can be useful when comparing applying a drug to a preparation for example. This display could indicate a change in firing rate before and after the drug was applied.

Mean frequency At each event a mean rate is calculated over the preceding data. The duration of the mean is user-defined.

Instantaneous frequency This style of display takes the inverse of the time difference between the current event and the one preceding it. This is displayed in Hz. The data is drawn as dots.

Raster This mode creates a pattern of events based on a second trigger or stimulus channel. An example of its use might be to see which WaveMark classes react to the stimulus being applied. It is also possible to view negative times by changing the y axis values.

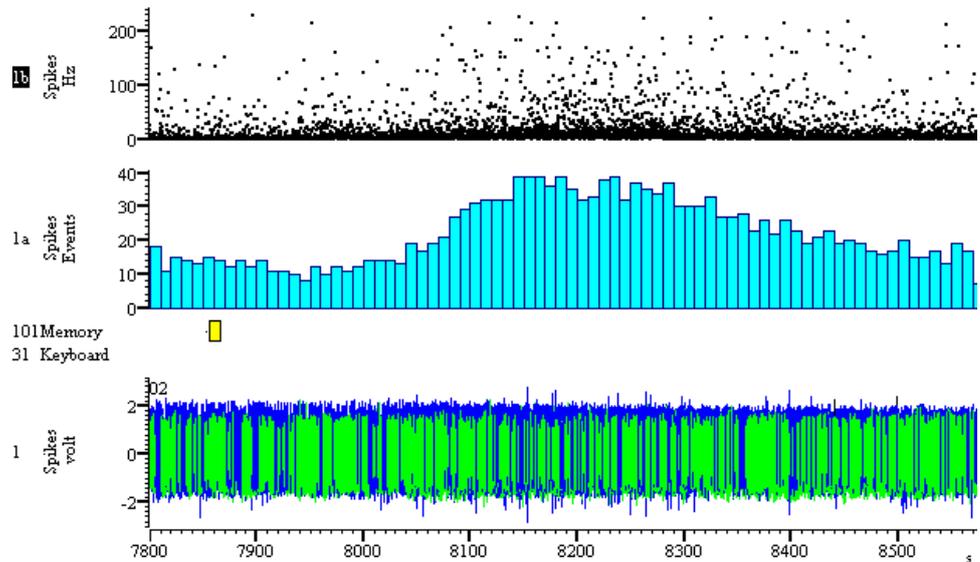
Dots and Lines

The simplest of the drawing modes. This display changes the WaveMark to a vertical line for ease of display. It is also possible to select Dots. By default, Spike2 uses small dots, but it is also possible to use large dots which is beneficial for some computer monitors.

When displaying WaveMark data as dots or lines it is important to note that the events are drawn at the beginning of the WaveMark not at the peak.

WaveMark This mode draws WaveMark data in its 'natural' state, showing the waveform.

The view below shows how event draw modes can give a better indication of what is really happening than the original data. At the text mark a drug was applied and took some time to cause a reaction. It is difficult to see from the WaveMark channel at the bottom that anything has happened so duplicates of the channel were created and filtered so that channel 1a displayed unit #1 and channel 1b unit #2. The user then changed the draw mode for each to display a rate histogram and an instantaneous frequency to determine firing rate. It is therefore possible to see that both units increased their firing rate in response to the drug. This is much clearer for unit #1; the display mode set for unit #2 does not make this so obvious.



Spike triggered averaging

This is a method of determining correlations between spikes and a waveform channel. This can be used in behavioural techniques. An example of this is to average a rectified EMG channel using a spike as a trigger. If there is a correlation between the spike occurrence and the EMG there should be evidence of increased EMG activity soon after the trigger.

Interval histogram

The interval histogram analysis shows how intervals between events are distributed. This can be used as a test that the spike stream being analysed truly comes from a single cell as there is a minimum period between firings from one source. If the interval distribution does not tail off to zero as the intervals get shorter, the events are unlikely to have a single source.

Stimulus histogram A stimulus histogram or PSTH show the likelihood of an event falling at a given period after an event on a different channel (usually marking a stimulus). The histograms simply show the number of events that fell in a particular time bin.

From Spike2 versions 3 you can choose to display the histogram results as a spike rate rather than as a count. The rate is calculated by dividing the count per bin by the number of sweeps and by the bin width.

Cross-correlation Cross-correlograms can be generated to produce a measure of the likelihood of an event on one channel occurring at a time before or after an event on another channel. In simple terms it means that given a trigger, a histogram of pre-determined width is produced and events from the source channel are 'dropped' into histogram bins corresponding to their original time from the trigger point.

All the histograms within Spike2 are of a constant bin width type; the width of the bins is user defined.

Use of marker filters and duplicate channels To use the above analysis techniques with WaveMark data it is often necessary to use the marker filter and duplicate channel options. The marker filter (from the Analysis menu) allows the user to show or hide any combination of WaveMark or coded events. This in turn allows you to analyse any particular set, or if a duplicate channel was created and a different set of codes was shown within it, you can perform unit to unit analysis.

Script introduction

Introduction This chapter has two sections. The **Introduction** 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 section, starting at the **Script language** title on page 59, introduces you to the basics of Spike2 (and Signal) scripts by examples and description. If you are experienced in programming, you may prefer to use the *Spike2 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 Spike2 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, Spike2 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 Spike2. 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$, type% {,mode% {,text$}});
```

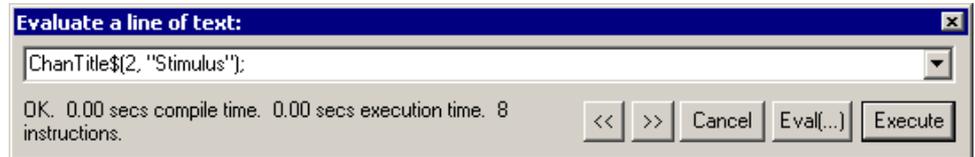
The items within the brackets are the requested arguments. In this case, `name$` requests the name of the file, `type%` determines which type of file it should be (for example a data file or a text file) and `mode%` determines the state of the window when opened (for example visible or maximised). Details 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.

Spike2 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 the 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 `demo.smr` 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:

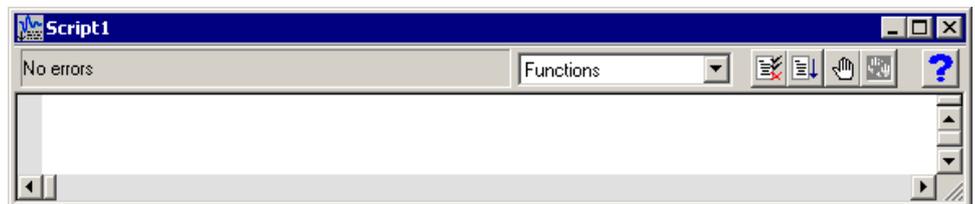
```
1/Binsize(1)
```

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 Spike2. 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

Spike2 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:\\Spike5\\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:\\Spike5\\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 Spike2 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 `Spike5\\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 6 script window improvements

The text editor was completely revised for version 6 of Spike2, 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 Spike2 and Signal. 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 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. Dividing the two forms the mean.
- 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], i%;
data%[0] := 10;
data%[1] := 20;
data%[2] := 30;
data%[3] := 40;
var total := 0; 'No need to set to 0 as happens automatically
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 contents of element `i%` of the array.
- In this example, the array `data%[]` has one index. We call this an array with one dimension or a vector. If we declared an array of reals as `var fred[8][6]`; this

creates an array with 2 dimensions and 48 elements addressed with two indices. We call an array with two dimensions a matrix.

- Before Spike2 version 6, the maximum number of array dimensions was 2. From version 6 onwards, arrays with up to 5 dimensions are allowed.
- From version 7, arrays can be resized with the `resize` keyword.
- We set variable `total` to zero to make the script action clear. However, numeric variables are set to zero if they are not initialised to a value.

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 Spike2 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 that 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 times as the 'for' loop is executed.

Views and view handles

An important idea to understand in the Spike2 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 that 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.

A number, called the *view handle*, identifies each view. 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.s2s
var vh%;                               'Variable for the view handle
vh% := FileOpen("demo.smr",0,1);      'Open file and make 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);                  'set window to 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 you move or show it, and the inner region with the user data or text. When you run a script, the inner region 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.s2s
var vh%;                               'Variable for the view handle
vh% := FileOpen("demo.smr ",0,1);      'Open file and make 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. Spike2 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.s2s
var vh%;                               'Variable for the view handle
vh% := FileOpen("demo.smr ",0,1);      'Open file and make visible
if vh% <= 0 then message("failed to open window");halt; endif;
Window(0,0,100,100);                  'whole screen
View(LogHandle());                    'make log view 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.s2s
var vh%;                               'Variable for the view handle
vh% := FileOpen("demo.smr ",0,1);      'Open file and make 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); 'log wnd 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.

New script features

At version 6, we added the ability to include separate files into a script. This allows you to have a set of common definitions, functions and procedures that are shared between scripts. At version 7 we added the `resize`, `break` and `continue` keywords; see your documentation for details. At version 8 we added default values for user-defined functions and the size of an integer became 64-bits.

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()` (pronounced “view dot”) 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.s2s
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 a percentage of the screen area. As an exercise, add two more arguments to randomise the size and run that. The full form of the function is: `Window(xLo, yLo{, xHi, yHi})`. Arguments in curly brackets are optional.

Drawing and updating Another important function in the script when dealing with data views is the `Draw(...)` function. `Spike2` 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.s2s
var vh% := 0;
var w, t;
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
w := Maxtime();                 'get maximum time in the file
XRange(0, 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 important part 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 Spike2 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.

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 4 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 `demo.smr` data file for this example:

```
'Cursor1.s2s
ToolbarText("Cursor functions");      'Stop screen jumping
if UCase$(FileName$(3)) <> "DEMO"
    then Message("Wrong file!"); halt endif;
ViewStandard();FrontView();          'Get into a tidy state
ChanHide(-1);ChanShow(1);           'Show channel 1 only (waveform)
CursorSet(2);                         'Add two vertical cursors
HCursorNew(1,1.0);HCursorNew(1,-1); 'These will be cursors 1 and 2
Interact("Position around the feature",4+32); 'user can modify
CursorRenumber();HCursorRenumber(); 'Get cursors in 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);                         'delete all vertical cursors
while HCursorDelete() do wend;        'and 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.s2s
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, save state
FrontView();ViewStandard();           'Make it visible
CursorSet(2);CursorLabel(2);           'show 2 numbered cursors
Interact("Swap cursors over and click OK",4+32);
CursorRenumber();                       'Get cursors in order
Interact("Now they are back in order",0);
View(App(3)).WindowVisible(sVis%);     'restore script 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 Spike2 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.s2s
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 that corresponds to the button that 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 Spike2 to do.

To make using the toolbar family easier, there is a script, ToolMake.s2s 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 ToolMake. 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 - MyIdle.s2s
DoToolbar(); 'Try it out
Halt;

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

Func MyIdle%() 'Idle - 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, 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.s2s
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 (but 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));
```

In case you don't come from the United Kingdom, and think that we've entirely lost our senses, you need to know that we are blessed with a National Lottery, or put another way, a tax on stupidity. The lottery is drawn by pulling 6 numbered balls, plus a "bonus" ball, from a machine holding balls numbered 1 to 49. To win the top prize you must match the 6 balls drawn out. There are lesser prizes for 5 matches, 4 matches and so on, and other prizes for combinations including the bonus ball.

The odds against winning (matching the 6 balls) are approximately 14 million to 1, assuming that you pick truly random numbers. Humans are notoriously bad at picking random numbers, which reduces the return on the bet (if you win) due to sharing the prize.

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.s2s` 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.s2s` 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:

1. Use `DlgCreate()` to start the creating process, give the dialog a name, and optionally position and size the dialog.
2. Use `DlgChan()`, `DlgCheck()`, `DlgInteger()`, `DlgLabel()`, `DlgList()`, `DlgReal()`, `DlgString()` and `DlgText()` to define fields in your dialog
3. 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

Spike2 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.s2s
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 (Pentium III 450 - this was written a long time ago), the hand written loop took 0.244 seconds. The same calculation using the array arithmetic took 0.0085 seconds, I had to repeat the array operations 100 times to get an accurate time. Some years later, on an AMD 1900+ machine, the times were 0.033 and 0.0011 seconds,

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

String functions

Spike2 scripts can use the usual string handling functions that you would expect:

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

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 un-joined.

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:

```
'XY.s2s
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 that may be used with XY view. These are listed below:

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

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

Scripts and Spike2 data

To follow this chapter it is assumed that the reader is familiar with the basics of the script language: Views, variables, arrays of data, looping constructs, Interact and the Toolbar commands. Full details of the script commands can be found in the Spike2 manuals and in the on-line Help, they are not replicated here.

These scripts do not use version 4 features, so should work on all Spike2 versions.

Opening files and channel information

The purpose of this session is to show you how to access data in a Spike2 data file from the script language. We will start with an example script that scans all the Spike2 data files in a folder and prints a summary of the channels found in each file.

```
'FileLst2.s2s
var fName$[100];           'allow for up to 100 files
var n%,i%,fh%;
View(LogHandle()).Window(0,0,100,100);
FrontView(LogHandle());   'Make sure log view visible
EditSelectAll();EditClear(); 'delete all existing text

var ChKind$[9];           'Names of all the channel types
ChKind$[0] := "Off";      ChKind$[1] := "Waveform";
ChKind$[2] := "Event-";  ChKind$[3] := "Event+";
ChKind$[4] := "Level";   ChKind$[5] := "Marker";
ChKind$[6] := "WaveMark"; ChKind$[7] := "RealMark";
ChKind$[8] := "TextMark";

FilePathSet("",0,"Set directory to list files in");
n% := FileList(fName$,0); 'get names of type 0 (son files)
for i%:=0 to n%-1 do      'if we got any files
  fh% := FileOpen(fName$[i%], 0, 0); 'Open file invisible
  if fh% < 0 then
    PrintLog("Error opening %s\n", fName$[i%]);
  else
    DumpFileInfo();       'routine to dump data
    FileClose();         'close the file
  endif;
next;
halt;

Proc DumpFileInfo()
var cList%[33],i%,c%,t%; 'channel list and work space
ChanList(cList%[]);     'get full channel list
PrintLog("%s\nTime resolution: %d us\n", FileName$,
BinSize()*1000000.0);
PrintLog("File comment : %s\n",FileComment$(1));
PrintLog("Chan Title Type Hz Scale Offset Units
Comment\n");
for i% := 1 to cList%[0] do
  c% := cList%[i%];     'get the channel number
  t% := ChanKind(c%);   'the channel type
  PrintLog("%2d %8s %8s", c%, ChanTitle$(c%), ChKind$[t%]);
  if t%=1 OR t%=6 then 'Waveform or WaveMark data
    PrintLog("%8g %8g %8g %5s", 1/BinSize(c%),
ChanScale(c%), ChanOffset(c%), ChanUnits$(c%));
  else
    PrintLog("%32s", ""); 'lots of blanks
  endif;
  PrintLog(" %s\n",ChanComment$(c%));
next;
PrintLog("\n");         'Make a gap between files
end;
```

The first part of the script uses `FilePathSet()` to select a current directory, then uses `FileList()` to collect the file names. Once we have a list of files, we attempt to open each in turn. Notice that the `FileOpen()` command allows us to open files without

displaying them. In this case we do not want to see the files, but it also means that you can arrange data files on the screen without wasting time drawing them in the wrong position. In fact, this is so useful that anywhere that the script language creates a new window it is initially invisible so that you can position it and then display it in the correct position. The full command is:

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

In this case we are supplying the file name with a `type%` of 0 for a Spike2 data file and using a `mode%` of 0, which means open invisibly ignoring any `.s2r` resource file and also not to worry if the file is already open in Spike2.

All script routines that create new windows also make the new window the current view, even when the window is invisible. The script calls the user-defined `Proc DumpFileInfo` to print information about the file. The file name plus the time resolution and the first line of the file comment are printed, then we iterate through the channels in the file and dump information about each channel.

We use the `ChanList()` function to get a list of all the channels in the file. This is a useful routine that can give you a list of the channels in a file that hold data of a particular type. In this case we ask for all channels. Instead of using this routine, the loop could have been written:

```
for c% := 1 to 32 do
  t% := ChanKind(c%);           'the channel type
  if t% > 0 then                 'if the channel is not Off
    . . .                       'the rest of the code
  endif;
next;
```

Usually you know what data type you expect to find in a channel, but in this case we do not so we use `ChanKind()` to find out. `ChanTitle$()` and `ChanComment$()` give us the channel title and comment, and can also be used to set the title and comment. `BinSize()` returns the sampling interval for a waveform or a `WaveMark` channel and the underlying time resolution of an event channel. The `ChanScale()` and `ChanOffset()` get the scale factors that convert the 16-bit data used for waveforms into user units. `ChanUnits$()` gets the units. Apart from `BinSize()`, all the other functions mentioned here can also change the value that they fetch.

`FileClose()` is used to shut the file once it has been dumped. If you forget to do this, Spike2 will accumulate a list of files. There is a limit to the number of files Spike2 will allow you to have open at a time, and there is also a limit imposed by the operating system. Further, each open file uses system resources and memory. For a fast, responsive system it is always a good idea to have as few files open at once as possible.

You could change the `DumpFileInfo()` procedure into an `AnalyseFile()` procedure to do batch processing on all the files in one folder.

ChanValue

One of the simplest commands for looking at the data in a channel is `ChanValue()`. This command generates the same values as you get from the `Cursor Values` dialog; that is it gives you the value of a channel at a particular time.

```
'ChanVal.s2s
var vh%;vh% := FileOpen("c:\\TrainDay\\demo.smr",0,0);
if vh% < 0 then halt endif;      'Stop if no file
Window(0,0,100,50);             'Set display position
ViewStandard();                 'All channels on view, no cursors
XRange(0,Maxtime());           'Display all data
```

```

FrontView(View());           'Make sure visible and at front
CursorSet(1);                'add one cursor in centre
Interact("Select position for cursor",4+32); 'allow move and View

var cList%(33),i%,c%;        'space for channel list
ChanList(cList%[],2048);     'exclude hidden channels
for i%:=1 to cList%[0] do    'for each visible channel
  c% := cList%i%;           'get the channel number
  PrintLog("%2d : %8g %s\n",
    c%, ChanValue(c%, Cursor(1)), ChanUnits$(c%));
next;
FrontView(LogHandle());

```

In this case we have used the command in its simplest possible form where it returns the value of the channel as it is displayed. However, this means that the command may return different values depending on the display mode. There are additional command arguments that can be used to force the display mode for a particular measurement. The full command is:

```
Func ChanValue(chan%, pos {, &data%{, mode%{, binsz{, trig%|edge%}}})
```

If there is no data at `pos`, the function returns 0, which is not too useful. The `data%` argument can be used to detect missing data. If used, it is returned as 1 if there is data and as 0 if there is not.

If you need direct access to the raw data, for example when you want all the waveform values or event times in a time range, `ChanValue()` is a slow way to get the values compared to `ChanData()`. However, when you only need a few spot measurements, or you want to re-sample data to present it for a spreadsheet so that you can produce a square table of results, it may be the only way to do it. This is also the only easy way to get at mean frequency values.

ChanMeasure

As `ChanValue()` is to the Cursor Values dialog, so `ChanMeasure()` is to the Cursor Regions dialog and is available from Spike2 version 3. Some of the measurements can be obtained using other commands, for example using `Count()` or `MinMax()` but if you want a quick and easy way to measure the standard deviation of a waveform, `ChanMeasure()` is the command for you.

```

'ChanMeas.s2s
var vh%;vh% := FileOpen("c:\\TrainDay\\demo.smr",0,0);
if vh% < 0 then halt endif;   'Stop if no file
Window(0,0,100,50);          'Set display position
ViewStandard();              'All channels on view, no cursors
XRange(0,Maxtime());         'Display all data
FrontView(View());           'Make sure visible and at front

CursorSet(2);                 'set two cursors
Interact("Select area to measure",4+32); 'allow move and View
CursorRenumber();            'make sure the cursors are in order
PrintLog("Standard deviation of channel 1 is %8g %s\n",
  ChanMeasure(1, 12, Cursor(1), Cursor(2)), ChanUnits$(1));
View(LogHandle());
Window(0,50,100,100);        'position the window
FrontView(LogHandle());      'make sure it is visible

```

The full command is:

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

The `type%` sets the measurement to take and `sPos` and `ePos` define the time range and `data%` is set to 1 if there is a result and to 0 if there is no data. For a waveform channel the possible values are:

1 Area	The area between the data points in the time range and the y axis zero.
2 Mean	The sum of the data points in the time range divided by the number of data points.
3 Slope	The slope of the least squares best fit line to the data in the time range.
4 Sum	The sum of the data values in the time range.
5 Area (scaled)	The same as Area. This is here to match the Cursor dialog.
6 Curve area	Each data point makes a contribution to the area of its amplitude above a line joining the endpoints multiplied by the x axis distance between the data points.
7 Modulus	Each data point makes a contribution to the area of its absolute amplitude value multiplied by the x axis difference between data points. This is equivalent to rectifying the data, then measuring the area.
8 Maximum	The maximum value found in the time range.
9 Minimum	The minimum value found in the time range.
10 Peak to Peak	The difference between maximum and minimum values in the time range.
11 RMS	The RMS level of the values in the time range.
12 SD	The standard deviation from the mean of the values in the time range.
13 Extreme	The maximum absolute value in the time range. Thus if the maximum value was +1, and the minimum value was -1.5, then this mode would display 1.5.
14 Peak	The maximum value in the time range measured relative to a baseline formed by joining the first and last points.
15 Trough	The minimum value in the time range measured relative to a baseline formed by joining the first and last points.

Getting a list of values

If you want to collect a list of consecutive data values from a channel, then you need:

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

It fills an array with waveform data or event times and optionally returns the time of the first item. The script `ChanData.s2s` is a modified version of `ChanVal.s2s` that illustrates the use of `ChanData()`.

```
var work[10],n%;
for i%:=1 to cList[0] do
  c% := cList[i%];
  n% := ChanData(c%, work[], Cursor(1), Maxtime());
  if n% > 0 then
    PrintLog("%2d : %8g %s\n", c%, work[:n%], ChanUnits$(c%));
  endif;
next;
```

The changed lines are shown above. Instead of using `ChanValue()` to get a single value we use `ChanData()` to fill an array with data. `ChanData()` needs a time range, so we say start collecting data from the cursor time, stop at the maximum time in the file. Unlike some programming languages, Spike2 knows how big the array you passed in is, so it will not overflow it.

Notice that `PrintLog` and `Print` will print an array for you (avoiding `for` loops), as long as it is acceptable for elements to be separated by a comma and for two-dimensional arrays have dimensions separated by a new line.

Although you can do almost any task by collecting the data in an array and processing it point by point, there are often better methods as the next section shows

Finding minimum and maximum values

This example finds the next maximum within 1 second of a cursor on channel 1 and moves the cursor to it. It assumes you have a suitable data file loaded (we suggest you use the `Demo.smr` file provided with Spike2). You will get script errors if you do not have a suitable file.

```
'MinMax.s2c
if UCase$(FileName$(3))<>"DEMO" then Message("File!"); halt endif;
ViewStandard(); 'get into a tidy state
Draw(0,4); CursorSet(1); 'Show 4 seconds and a cursor
ToolBarVisible(1); 'keep toolbar on show
var n%, minv, maxv, minP, maxP;
repeat
  MinMax(1,Cursor(1)+0.5, Cursor(1)+1.5, minV, maxV, minP, maxP);
  XRange(maxP-2, maxP+2); 'centre the maximum found
  CursorSet(1, maxP); 'move cursor to new position
  n% := Interact("Position the cursor", 32, 0, "Stop","Next");
until n%=1;
```

The `ToolBarVisible(1)` command keeps the toolbar line (used for the `Interact()` command) permanently visible. Without it, the interact command will scroll the screen down and up one line each time it is used, which is visually unpleasant.

The `MinMax()` command scans the waveform data on channel 1 from 0.5 seconds beyond the cursor to 1.5 seconds beyond the cursor and finds the maximum in this range.

`Interact()` is used with two buttons so we can choose to move on or stop.

When you try this script you will discover that the data in the file is not well suited to this peak picker. You really need a method to find all the peaks in the signal and a scheme to move backwards and forwards to the next or previous peak. Picking peaks is available through memory channels; we'll cover that topic in a separate session. You can also pick peaks interactively using the Analysis menu Memory buffer commands.

Finding the next and previous data item

As it happens, the `Demo` data file already has peaks marked on channel 2. The next script implements a simple script that moves you through the data peak by peak, based on events on channel 2. Again we assume that the current file is `Demo.smr`.

```
'NextLast.s2c Simple demonstration of NextTime and LastTime
if UCase$(FileName$(3))<>"DEMO" then Message("File!");halt endif;
ViewStandard();FrontView(); 'Get into a tidy state
ToolBarVisible(1); 'Stop screen jumping
CursorSet(1); 'Add a single cursor
var Action% := 5; 'Set to move to the start
var t; 'work variable, holds cursor time
repeat
  t := Cursor(1); 'Starting cursor position
  docase
    case Action%=2 then t := MaxTime(2);
    case Action%=3 then t := NextTime(2, t);
    case Action%=4 then t := LastTime(2, t);
    case Action%=5 then t := NextTime(2, -1);
    case Action%=6 then Optimise(1);
  endcase;
  if (t>=0) AND (t<>Cursor(1)) then
    CursorSet(1, t);
    Draw(t-(XHigh()-XLow())/2); 'centre the cursor
  endif;
  Action% := Interact("Move cursor and choose action", 4+32, 0,
    "Quit", "End", "Next", "Last", "Start", "Optimise");
until Action% = 1; 'Run until user selects Quit
```

This is starting to behave more like a useful script! The repeat loop gets a new button from the `Interact()` command and then does the appropriate task. By initialising `Action%` to 5, we start the system as though the user had just pressed the button for "Start". The `docase` statement is a slightly tidier way of implementing a multiple way branch.

The important new commands here are:

```
Func LastTime(chan%, time{,&val|code%[{}],data[]|data%[]|&data$});
Func NextTime(chan%, time{,&val|code%[{}],data[]|data%[]|&data$});
```

These find the previous and next data item on a channel and return the time of the item, or -1 if there is no item. Remember that waveform data is composed of individual, equally spaced data points; each point is a data item.

The commands can also return additional data about the item. For a waveform channel, they can return the value of the waveform point. For event level data you can find the level, for markers and derived types the data associated with the marker can be read. When you need to iterate through a data file, these commands are often your best choice.

Channel duplication and drawing mode

There are a few channel commands we haven't yet used, so here is a script that uses most of the rest! This uses the Demo file again, and hides channels 1 and 2, duplicates channel 3 twice and draws it in three different ways.

```
'DrawMode.s2c
if UCase$(FileName$(3)) <> "DEMO"
  then Message("Wrong file!"); halt endif;
ViewStandard();FrontView();           'Get into a tidy state
ChanHide(-1);ChanShow(3);             'Show only channel 3
var d1%,d2%,d3%;                       'Three duplicate channel numbers
d1%=ChanDuplicate(3);                 'Duplicate the channel 3
d2%=ChanDuplicate(3);                 'and again...
d3%=ChanDuplicate(d2%);               'Just to show you can...
DrawMode(3,1);                         'dots mode
DrawMode(d1%,2);                       'lines mode
DrawMode(d2%,5,1.0);                  'Rate with 1 second bins
DrawMode(d3%,6,4.0);                  'mean frequency, 4 secs smoothing
ChanShow(d1%,d2%,d3%);                'show them all
XRange(0,Maxtime());Optimise(-1);     'show all and optimise
Draw();                                'Update the display
Seconds(0);                            'display time remaining
While Seconds()<10 do WindowTitle$(Print$(10-Seconds())) WEnd;
WindowTitle$("Script finished...");
ChanDelete(d1%);                       'Kill off the duplicates...
ChanDelete(d2%);                       '...so we leave everything...
ChanDelete(d3%);                       '...nice and tidy
```

Result views

A result view is a one-dimensional array that is drawn in a window in a variety of styles. You can create result views using the data analysis commands, where a data channel or channel in a time view is processed and the results added into a result view; you can use the `SetResult()` command, which generates a user-defined result view; or you can open an existing result view on disk using `FileOpen()`. We'll return to analysis later; now we'll create a user-defined result view and fill it with data:

```
'Result1.s2s
var rh%;                               'to hold the view handle
ToolbarVisible(1);                     'Keep toolbar visible
rh%:=SetResult(100,0.02,-1,"Hi","XUnit","yUnit","x","y");
Window(0,0,90,60);                     'set some useful size
WindowVisible(1);                       'so we can see it
Grid(1);                                 'and lets have a grid
Interact("See the empty view!",0);

var i%,v;                               'working variables
for i% := 0 to 99 do                     'set bin contents the hard way
    [i%] := Cos(0.1*i%);                 'access result bins as unnamed arrays
    next;
Optimise(0);                             'use a channel number of 0 for now
Interact("See a cosine wave!",0);

for i%:=1 to 4 do                         'Show the various drawing modes
    DrawMode(0,i%);
    Interact(Print$("DrawMode %d",i%),0);
    next;

DrawMode(0,1);                           'histogram mode again
for v:=0 to 1 step .0025 do
    ArrConst([40:20],v);                 'fill with constant value
    ArrIntgl([40:20]);                   'convert into a ramp
    Cos([40:20]);                         'take cosine
    Draw();                               'update, only changed bins update
    next;
FileClose(0,-1);                         'kill it off, don't ask about saving
```

The `SetResult()` command generates a new view and returns the view handle. All script commands that create a new view make the new view the current view. A result view can be treated much the same as a time view with a channel number of 0.

Access to result view contents

You can access the contents of a result view by treating it as an array with no name. For example, to increase the contents of the first bin you could write:

```
[0] := [0] + 1;                          'increase bin contents by 1
```

You can read the contents of a result view that is not the current result view with:

```
value := View(OtherView%).[binNumber%];
```

From version 3 you can set non-current result view bins in the same way. You can also access result view data in terms of the x axis units using `ChanValue()`:

```
value := ChanValue(0,-0.1); 'data in bin with -0.1 as x value
```

From version 5 you can use `View(handle%,chan%).[bin%]` to refer to a specific channel in a result view or `View(handle%,chan%).[]` to mean an entire channel.

Save and load result view

You can save a result view using `FileSaveAs()` and load it again with `FileOpen()`:

```
'Result2.s2s
var rh%;                               'to hold the view handle
rh% := SetResult(100, 0.02,-1, "Example result",
                "XUnit","yUnit","xTitle","yTitle");
Window(0,0,90,60);WindowVisible(1);Grid(1); 'so we can see it
```

```

ArrConst([],.1); ArrIntgl([]); Cos([]);Draw(); 'make cosine, draw
FileSaveAs("MyResult.srf",4, 1);           'Save, no questions asked
FileClose(0,-1);                           'don't ask about saving
Message("The view has gone...");
rh% := FileOpen("MyResult.srf",4,1);       'Open and show it again

```

Save result view as text You can also save the entire result view very easily as a text file:

```

'Result3.s2s
SetResult(100, 0.02,-1, "", "");           'Make a result view
ArrConst([],.1); ArrIntgl([]);           'Fill with a ramp
FileSaveAs("MyResult.txt",1, 1);         'Save it as text
FileClose(0,-1);                         'don't ask about saving
FileOpen("MyResult.txt",1,1);            'Open, show text

```

This is a minimalist example; the simplest form of result view creation is used, we fill the result view with a ramp, then we save it as text. We also ignore any error codes returned. The output in the text view is:

```

100
0.1
0.2
0.3
...

```

The first line is the number of bins, the following lines are the contents of each bin.

Result view text output with user-defined format

If you want more information to be written to the output, you could write your own output file. You can create two forms of text file: A text window that you write to and save later, or an external text file with no window. The external text file writes much faster as it does not need to update a screen view. The following code creates both types and writes the same information to each:

```

'Result4.s2s
var rv%, tv1%, tv2%;
rv% := SetResult(100, 0.02,-1, "", ""); 'Make a result view
ArrConst( [],.1); ArrIntgl( []);       'Fill with a ramp

tv1% := FileNew(1);                    'Make hidden text window
OutputTo(tv1%, rv%, 40, 60);           'Write to text window
View(tv1%).WindowVisible(1);          'make text window visible

tv2% := FileOpen("text2.txt",8,1);     'Make external text file
OutputTo(tv2%, rv%, 40, 60);           'Write to external file
View(tv2%); FileClose();               'close external file

View(rv%); FileClose(0,-1);            'kill the result view
halt;

Proc OutputTo(tv%, rv%, bSt%, bEnd%);   'Write rv% data to tv%
var i%;
View(tv%); Print("bin xValue Data\n"); 'print headings
for i%:=bSt% to bEnd% do                'Loop for each bin
    Print("%3d %5.2f %g\n",i%,
          View(rv%).BinToX(i%), View(rv%).[i%]);
    next;
end;

```

This method, although slower, gives you a free choice of the data you want written to the text file. If you need to create a large output file, the external text file method is recommended as it is fast and has no limit on the file size. (Spike2 version 2 text windows have a limited size; there is no limit in versions 3 and 4).

Note the use of `BinToX(I%)` to get the x value corresponding to a particular bin.

Analysis commands

In this context, an analysis command is one of the `SetXXXX` family of commands that create a result view and the associated `Process` command that takes time view data and adds the result into the result view. In the discussion of the analysis commands we do not have time to look at the individual commands, we will restrict ourselves to looking at one command, `SetPSTH` and to keep things simple, we will use a single data file `Demo`.

Script analysis commands are basically very similar to the on-line interactive analysis system. One way to write such analysis scripts is to record your actions, and then edit the arguments to suit the application.

The first example script simply creates a `PSTH` analysis of the entire file, using channel 2 as the trigger and channel 3 as the data to be analysed. We will set the script up so that time delay 0 is in the centre of the output, and that the output runs from -1 to +1 seconds. We will make it easy to change the bin width.

```
'analyse1.s2s
var vh%, rh%;
vh% := FileOpen("Demo.smr", 0, 1);    'Open demo file and show it
if (vh% <= 0) then Message("No DEMO file!"); halt endif;

var bins%, binSz:=0.01, offset:= -1.0;
binsz := Input("Set the binsize in seconds", binsz, 0.0001, 0.1);
bins% := 2.0/binsz;                    '2 seconds wide, calc bins
rh% := SetPSTH(3, bins%, binsz, offset, 2);
Process(0, View(vh%).MaxTime(),1,1);  'process, clear, optimise
WindowVisible(1);                     'display the result
```

The important points to remember when using commands like this are:

1. `SetXXXX` creates a new view and makes it the current view.
2. The new view is created invisible. It is up to you to position it and make it visible.
3. To use `Process()`, the current view must be the result view. This is because you can have many result views attached to a single time view and `Spike2` needs to know which one to update.
4. You can choose to empty the result view before the new data is added in, and you can choose to optimise the display each time data is added.

The script does not include a `Draw()` command to update the result, yet you see the result anyway. This is because the `WindowVisible(1)` command made the window frame visible immediately, the frame included the display area which is not updated, but is marked as invalid. As soon as idle time became available (when the script ended) `Spike2` draws the invalid region. This probably seems a small point, but many people get very confused when a running script (with no idle time due to `Dialogs`, `Interact` or `Toolbar` calls) creates empty frames, but no data.

To add the analyses of several areas together, you would not clear the array results before adding in the new data. A slightly more developed version of the above script might look like:

```
'analyse2.s2s
var vh%, rh%;
vh% := FileOpen("Demo.smr", 0, 1);    'Open the demo file, show it
if (vh% <= 0) then Message("No DEMO file!"); halt endif;
ToolbarText("Process demo");          'so toolbar doesn't bounce
Window(0,0,100,50);
FrontView();                           'Make sure it is visible
View(LogHandle()).EditSelectAll();     'Select all text...
View(LogHandle()).EditClear();         '...and clear it
PrintLog("Process demonstration\n");  'Will clear selected text
View(LogHandle()).Window(50,50,100,100); 'Show log window too
CursorSet(2);                           'Show two cursors
var bins%, binSz:=0.01, offset:= -1.0;
```

```

binsz := Input("Set the binsize in seconds", binsz, 0.0001, 0.1);
bins% := 2.0/binsz;                                '2 seconds wide, calc bins
rh% := SetPSTH(3, bins%, binsz, offset, 2); 'create result view
Window(0,50,50,100);                               'position the view
WindowVisible(1);                                  'display new view
While View(-1).Interact("Select area",36,0,"Quit","Add area")=2 do
  CursorRenumber();                               'get the right way around
  Process(View(-1).Cursor(1), View(-1).Cursor(2),0,1); 'process
  PrintLog("Processed from %.6f to %.6f\n",
          View(-1).Cursor(1), View(-1).Cursor(2));
WEnd;

```

Points to notice include:

1. We leave the current view as the target result view, but we use `View(-1).xxx` as a view override. The -1 means the time view associated with this result view. This is provided as a shorthand method, and saves explicitly specifying the time view.
2. A common mistake is to write `Process(Cursor(1),Cursor(2));` which can cause a lot of head scratching, especially if there are cursors in the result view, as you may get no error, but instead the data range added makes no sense.

This script does have one more problem left to solve. Each time you run it you are left with another result view. One simple solution is to add a `FileClose()` command at the end so the user will be asked if they want to save the view, or let them cancel the close.

Memory channels

When analysing Spike2 data one often needs to create new data channels derived from the raw data. For example:

1. An event channel holding an event every time there is a peak in a waveform channel.
2. A channel with an event every time a waveform channel has a peak below zero volts.
3. A waveform channel containing a rectified version of another waveform channel (or maybe a differentiated, filtered or integrated version).

Spike2 can add new channels to a data file and fill them with data. These channels are called memory channels. They differ from file-based channels in the following ways:

1. They are not saved and restored when you close and open a data file.
2. They are limited by available memory (in Windows they are limited by the virtual memory you have enabled, on the Macintosh they are limited by the memory space you have allocated to your Spike2 application).
3. You can edit the data, and insert or delete sections.
4. You can create up to 26 memory channels that are attached to a data file.
5. They have channel numbers starting from 101.

Creating memory channels from a script

There are three steps to creating your own channel in a data file with a script:

1. Create a new channel with the desired properties (channel type, resolution etc.)
2. Place data into this channel, then you will probably want to either:
 - 3a. Save this channel to disk so it is permanently part of the data file, or
 - 3b. Extract a table of times/values from this new channel and print them to a text file and then discard the new channel.

You can, of course, extract a table of results AND save the new channel to disk. In this tutorial, we will write two scripts. One will create an *event* memory channel and dump data from it. The second will create a *waveform* memory channel with a modified version of the raw data waveform in and then save this channel to disk. These two scripts cover most of the basic concepts and standard functions that you will need to write your own data analysis scripts. You will find more information about memory channels in the *Advanced topics* chapter.

Memory script 1

Our first script will do the following:

1. Open the demo file “*demo.smr*”.
2. Create a new *event* memory channel
3. Import as markers, all the peaks from channel *I Sinewave*.
4. Create a text file and dump a table of x,y co-ordinates of the peaks in *I Sinewave* to this file. (This text file can then be imported to a spreadsheet for further analysis).

Step 1: Open file and make memory channel

To start with we need to open the demo data file and create a new memory channel.

```
'mem1a.s2s
'
'This script opens the demo file and creates a new memory marker
'channel. This new memory channel is made visible, but is only
'stored in memory and not on the hard disk, so when the file is
'closed this channel disappears
var vh%;                               'View handle
var mmChan%;                           'Memory channel number

vh%:=FileOpen("demo.smr", 0, 1+2);'Open the demo data file
if vh%<0 then Message("No demo file");Halt endif;      'open file?

Interact("Press OK to create new channel", 36); 'Wait for OK
mmChan%:=MemChan(5);                               'Create memory marker channel
ChanShow(mmChan%);                                 'Make the new channel visible
```

Creating memory marker channels

`MemChan()` is used to create a new memory channel in a data file. Type 5 (which we have used) means we want to create a *marker* channel. The `MemChan()` function returns the channel number of the new channel that has been created. This is important as we cannot know in advance what channel number this newly created channel will be and later in the script we will need to refer to this new channel by channel number. The line:

```
mmChan%:=MemChan(5);
```

creates a new memory marker channel and places the channel number of this newly created channel into the variable `mmChan%`. From now on, whenever we need to refer to this channel by channel number, we will simply refer to the variable `mmChan%`.

When a channel is first created, it is created hidden. So though we would be able to add data to the channel, we would not be able to see it. We use the `ChanShow()` function to show the channel in the file. The argument passed to `ChanShow()` is the number of the channel to display.

Run the script. The first time you run the script, the demo data file will be opened and displayed on the screen. The script waits at the `Interact()` statement until the user clicks “OK” in the toolbar. Then it creates the new memory channel.

The new memory channel is channel *101 Memory* (*101* is the channel number and *Memory* is the channel title). When the script was run this first time, `mmChan%` was set to 101 when the `MemChan()` function was called.

Now run the script again without closing *demo.smr* file first. The script makes a new memory channel but this time it is number 102 (since there is already a channel 101).

If you close *demo.smr* and reopen it, the newly created memory channels will disappear.

Step 2: Import peaks into memory channel

We will do two things here.

1. When the demo data file is first opened, there may be any number of the channels in it hidden or showing. Since we are only interested in channel 1 and our new memory channel, we will hide all the others and just show these two. This will make the file visibly clearer for the user.
2. Import the peaks using the `MemImport()` function.

Just before the `Interact` statement, add the following two lines (or open `mem1b.s2s`):

```
ChanHide(-1);           'Hide all channels
ChanShow(1);           'Show channel to import
```

We have already come across `ChanShow()`. `ChanHide()` works in exactly the same way but hides channels instead of shows them (an argument of -1 means to hide all the channels). So these two lines hide all the channels except for channel 1 (the one we are going to extract peaks from). Now, at the end of the script, add the line:

```
MemImport(mmChan%, 1, 0, 120, 0, 0, 1); 'Import peaks
```

This line imports peaks into our memory channel. `MemImport()` expects 7 parameters:

The memory channel number (`mmChan%` in our case)

The input channel number (1)

The start time (0 sec)

The end time (120 sec)

The mode of importation (0 - peak detection mode)

The time and level values (0 and 1, these are explained in detail in the *Spike2* manual)

Run the script. This will import the peaks to the memory channel as markers. Try going to the view menu and changing the draw mode for the memory channel.

Step 3: Print the output

Next, we will write a function to print the co-ordinates of each of the detected peaks to a new text window (or open `mem1c.s2s`). This is a task that is ideally suited to be a user-defined function. This function will have two arguments, an event channel number and a waveform channel number. The function will firstly create a new text window. It will then find the time of the first event in the event channel and the value of the waveform channel at that time and then print that pair of numbers to the text window. Then it will find the time of the next event and do the same thing. It will repeat this process until there are no more events to find in the event channel. The function looks like this:

```
'Prints the value in waveChan% at times specified
'by events in eventChan% to a text window
func PrintMarkVals(eventChan%, waveChan%)
var time:=0;           'Set to start of file
var txt%; txt%:=FileNew(1,1); 'Create a new text window
if txt%<0 then Message("Text file not opened");Halt endif;

repeat
  time:=View(vh%).NextTime(eventChan%, time); 'Find next event
  if time<>-1 then           'if found, print wave value at time
    Print("%f\t%f\n",time,View(vh%).ChanValue(waveChan%,time));
  endif;
until time=-1;           'until no more events left
FileClose();
end;
```

There is a new system function here, `NextTime()`. It expects two arguments, a channel number and a time and returns the time of the next event in the channel after the time passed into it. If there are no events after this time, the function returns -1.

The user-defined function `PrintMarkVals()` has a variable, `time`, which is initialised to 0 seconds. `time` is set to the time of the next event after 0 seconds. If an event is

found, it prints the time and the waveform channel value at that time to the text window. This is the x,y co-ordinate of the first peak. Then we go back to the `repeat` statement and do this again until we don't find a new event.

When `NextTime()` fails to find an event, printing is skipped and when the script reaches the `until` statement, it does not go back to the `repeat`, but instead carries on to the end of the function.

Note: We used a Tab (`\t`) to separate the time from the waveform value when printing to the text view. We could have used a comma, or a space, or indeed any character but in our experience Tab is the most commonly accepted character in spreadsheets and word processors.

Append this function to the script and then add the following lines after the `MemImport()` statement.

```
Interact("Press OK to print to a text file", 36); 'Waits for OK
PrintMarkVals(mmChan%, 1); 'Print the peak values
```

The first line we are familiar with, it just makes the script wait until the user presses OK in the toolbar at the top of the screen before continuing. The second line calls our newly defined function `PrintMarkVals()`, passing in `mmChan%` as the event channel and 1 as the waveform channel.

Try running it now. The extracted results are printed to a text window on the screen and then when the script tries to close the text window, you are prompted about whether you wish to save it or not. You should select **Yes**. This file can then be opened in a spreadsheet application.

Step 4: Tidy up The script now does the task. There are a few things that should be done to tidy it up:

1. It is not good programming practice to have numbers in a program without it being obvious what they are or how to change them. For example, we have extracted the peaks from channel 1, but what if we wanted to extract troughs from channel 6? We would have to find which "1"s in the script were referred to the channel and change them to "6"s and change the `MemImport()` function to detect troughs, not peaks.

It is better to have a list of "constants" at the top of the script specifying such things and replace the numbers with the constants in the body of the script. In the final version we have replaced many of the numbers with constants.

2. A problem with `Interact()` is that it makes a toolbar appear at the top of the screen and then disappear when it leaves the statement. This means that the screen gets jiggled about a lot during the running of the script and is not very pleasing to the eye. So near the beginning of the script, we have added the line:

```
ToolbarVisible(1); 'Make toolbar always visible
```

This makes a toolbar appear at the top of the screen throughout the running of the script whether the script is waiting at an `Interact()` statement or not.

3. Before Spike2 version 3, text windows could hold a limited amount of text (typically less than 32000 characters). This is no longer the case, but text windows get slower as the text within them increases. For our example this did not matter very much as we were not printing a huge number of results to it. However, if you do not need to see your results being printed and you will always add text at the end of the file, it is better to print them to an "external text file". These files can be of any size and are usually faster to write to, especially for very large files.

To use an external text file instead of a text window, we have used the line

```
txt%:=FileOpen("",8,1); 'Open an external text file
```

instead of the `FileNew()` command. This will prompt the user for a file name for the text file and then dump all the results directly to disk.

The final `mem1.s2s` script looks like this:

```
'mem1.s2s
'This script opens the demo file and creates a new memory marker
'channel. This new channel is made visible, but is only stored in
'memory, not on disk, so when the file closes it is lost. The
'script uses MemImport to fill the new channel with peak times.
'Finally it prints the peak positions to an external text file
const channel%=1;           'Channel to import peaks from
const sTime:=0;            'Start time
const eTime:=120;         'End time
const mode%=0;            'Importation mode
const time:=0;            'Min interval between events
const level:=1;          'level to fall/rise/cross
var vh%;                  'View handle
var mmChan%;              'Memory channel number

ToolbarVisible(1);        'Makes toolbar always visible
vh%:=FileOpen("demo.smr", 0, 1+2); 'Open the demo data file
if vh%<0 then Message("No demo file");Halt endif; 'check OK
ChanHide(-1);            'Hide all channels
ChanShow(channel%);      'Show channel to import from

Interact("Press OK to create new channel", 36); 'Wait for OK
mmChan%:=MemChan(3);     'Create a memory marker channel
ChanShow(mmChan%);      'Make the new channel visible
                          'Import peaks to new channel
MemImport(mmChan%, channel%, sTime, eTime, mode%, time, level);

Interact("Press OK to print a text file", 36); 'Wait for OK

PrintMarkVals(mmChan%, channel%); 'Print peaks to the log window

'This function prints the value in waveChan% at times specified
'by events in eventChan% to a text file
'func PrintMarkVals(eventChan%, waveChan%)
var time:=0;              'Set to start of file
var txt%;

txt%:=FileOpen("",8,1);   'Open an external text file
if txt%<0 then Message("Text file not opened");Halt endif;

repeat
  time:=View(vh%).NextTime(eventChan%, time); 'Find next event
  if time<>-1 then ' if there is one print wavechan% value
    Print("%f\t%f\n",time,View(vh%).ChanValue(waveChan%,time));
  endif;
until time=-1;           'until no more events left
FileClose();
end;
```

Memory script 2

For our second memory channel script, we will create a waveform memory channel and fill it with a waveform that is a rectified version of the waveform in channel *I Sinewave*. The script will:

1. Open the demo file “*demo.smr*”
2. Create a new waveform memory channel.
3. Copy a section of the waveform from channel *I Sinewave* into an array.
4. Rectify the array.
5. Copy the waveform from the array into our new memory channel.

As a final refinement, we will allow the user to choose the file and channel to rectify.

Step 1: Make a waveform channel

The first step is almost identical to the first step of *Memory channel script 1*.

```
'mem2a.s2s
'
'This opens the demo file and makes a waveform memory channel.
'This new channel is made visible, but is stored in memory, not
'on disk, so when the file is closed this channel disappears
var vh%;                               'View handle
var mmChan%;                             'Memory channel number

vh%:=FileOpen("demo.smr", 0, 1+2); 'Open the demo data file
if vh%<0 then Message("No demo file");Halt endif;      'Check OK

Interact("Press OK to create new channel", 36);        'Wait for OK
mmChan%:=MemChan(1, 0, 0.01);      'Make waveform, 0.01 binsize
ChanShow(mmChan%);                 'Make the new channel visible
```

The difference is that `MemChan()` has different arguments. The first argument tells it what type of channel to create. We have used type 1, waveform. The next argument is not used when creating a waveform channel and so is set to zero and the last argument is the *bin size* of the channel (1/sampling frequency). We have set this to be 0.01 to match as the bin size of channel *I Sinewave*.

Try running the script. You will see the new memory channel appear in the data file. Note that, again, it disappears if you close the file and reopen it.

Step 2: Extract waveform data

Now we create a function to extract the waveform from channel 1 and (for the moment) print all the values in it to the log view. We will call this function `DoData()`. It is quite important that this is done as a separate, user defined, function. The reason for this will be made clear later. Our new function looks like this (add it to the end of the script we have so far or open `mem2b.s2s`):

```
'Take the waveform from channel oldChan% and prints it to the
'log view. It assumes you do not have more than 1000 data points
func DoData(newChan%, oldChan%, sTime, eTime)
var arr[1000];                               'Assume 1000 points maximum
var pts%;                                    'Number of points extracted
pts%:=ChanData(oldChan%, arr[], sTime, eTime); 'get data points
PrintLog("%f",arr[:pts%]);
end;
```

The important line in this function is:

```
pts%:=ChanData(oldChan%, arr[], sTime, eTime); 'get data points
```

This function retrieves all the data points within two times (`sTime` and `eTime`) and places the values into an array (`arr[]`). We do not know in advance exactly how many data points will be extracted, so the `ChanData()` function returns the number of data points placed into the array.

From now on, when referring to the data stored in `arr[]`, we will refer to it as `arr[:pts%]`. This notation means, just consider the first `pts%` points in the array. This is important because if, for example, you want to print out the values you have extracted and you just had:

```
PrintLog("%f", arr[]);
```

This would print out 1000 values (the size of `arr[]`) to the log view when you may have only extracted 10 values from the data channel.

We need to call our new function, so after the `ChanShow(mmChan%);` line, we add:

```
DoData(mmChan%, 1, 0, 10); 'Copy waveform from chan 1 to mmChan%
```

This line calls our `DoData()` function. It tells it that the new memory channel is channel `mmChan%`, the channel to take the data from is channel 1 and we wish to look at data from 0 to 10 seconds.

If we make the log view visible and **run the script**, we will see the value at each sample in the first ten seconds of waveform printed to the log view.

Step 3: Copy data to memory channel

Now that we have seen the data extracted from the raw data waveform and placed into an array, we need a way of placing the data in the array back into the new memory channel. Either open `mem2c.s2s` or replace the line:

```
PrintLog("%f",arr[:pts%]); 'Print the array to the log view
```

with:

```
MemSetItem(newChan%, 0, sTime, arr[:pts%]);'Copy to new channel
```

`MemSetItem()` allows you to place one or more items into a memory channel. In this case, we have specified that we want to place data in channel `newChan%`, starting at time `sTime` and use the first `pts%` points in `arr[]` as our source.

Run the script. A copy of the channel 1 *Sinewave* data appears in the memory channel.

Step 4: Arrays of variable size

Try changing the line

```
DoData(mmChan%, 1, 0, 10); 'Copy waveform from chan 1 to mmChan%
to
DoData(mmChan%, 1, 0, 40); 'Copy waveform from chan 1 to mmChan%
```

i.e. instead of copying the first 10 seconds, copy the first 40 seconds. **Run the script.** It doesn't work. The script still copies the first 10 seconds. This is because our `DoData()` function assumes a maximum of 1000 data points (we defined an array with 1000 bins). For the script to copy 40 seconds of data, the array must be of size 4000. But then, suppose we want to copy even more data? Or maybe we want to choose a channel whose sampling rate is higher than 100Hz? We need to set the array size based upon the size of the data section we wish to extract.

This can be done by defining an integer variable to be the maximum number of data points that we will need. This is dependent upon the start time, end time and bin size of the data section we wish to retrieve. We can then define our array to be of this size. Our new script looks like this:

```

'mem2d.s2s
'
'This opens the demo file and makes a waveform memory channel.
'It copies the first fifty seconds of the waveform in channel 1
'to the new memory channel
var vh%;                               'View handle
var mmChan%;                           'Memory channel number

vh%:=FileOpen("demo.smr", 0, 1+2);'Open the demo data file
if vh%<0 then Message("No demo file");Halt endif;      'Check OK

Interact("Press OK to create new channel", 36);        'Wait for OK
mmChan%:=MemChan(1, 0, 0.01);      'Make waveform, 0.01 binsize
ChanShow(mmChan%);                'Make the new channel visible

DoData(mmChan%, 1, 0, 50);         'Copy from channel 1 to mmChan%
                                   'first 50 sec only

'This copies the waveform from channel oldChan% to newChan%
'It does not matter how many points need extracting.
func DoData(newChan%, oldChan%, sTime, eTime)
var maxSize%;
maxSize%:=Trunc((eTime-sTime)/Binsize(oldChan%))+1;
var arr[maxSize%];                 'Set array with points required
var pts%;                           'Number of points extracted
pts%:=ChanData(oldChan%, arr[], sTime, eTime);'Get data points
MemSetItem(newChan%, 0, sTime, arr[:pts%]); 'copy to new channel
end;

```

Our variable we have defined in our `DoData()` function called `maxSize%` is the maximum number of data points that could be extract using `ChanData()` given that we know the bin size of the channel and the length of data section we want.

We still do not know exactly how many points will be extracted because there may not be a continuous stream of data between `sTime` and `eTime`, so we still need to use the `pts%` variable to keep a record of how many points have been placed in the array.

It was mentioned earlier that we must put our data extracting routine in a user-defined function. The reason for this is that you can only define arrays with a variable specified size as local arrays existing inside a function. If you try to define a global array in this way the script will give a compiler error.

We can run our script now with any size of data section to copy and we no longer need to worry about whether all our data will fit into the array we have defined.

Step 5: Tidy up

There are now 3 things left to do:

1. We need to alter our `DoData()` function slightly, so that it does not just make a straight copy of the data section, but instead makes a rectified version of it.

We do this by changing the data in the array, after we have extracted data and before we have placed it in the memory channel. The `Abs()` function takes an array as an argument and converts every negative element into a positive element of equal amplitude. i.e. if you pass in an array holding 0, -1, 4 and -6.2 to `Abs()` it will turn it into 0, 1, 4 and 6.2.

2. It would be nice to open any file and rectify any channel. Our final version lets the user specify a file and channel. The memory channel is created with the same *bin size*, *scale* and *offset* as the selected channel. The entire channel (0 to `Maxtime()`) is rectified into the new memory channel.

3. We need to add a function to the script to save our memory channel as a proper data channel on disk so it is part of the data file and not just stored in memory. Our new `SaveChannel()` function creates a list of unused channels and then saves our new channel in the first available slot.

A listing of the final script follows:

```
'mem2.s2s
'This script lets a user open a data file and select a channel to
'rectify. It creates a memory channel with the same binsize as
'the selected channel and copies a rectified version to it. The
'user is prompted to press OK to save the newly created channel
var vh%;                               'View handle
var mmChan%;                             'Memory channel number
var channel%;                             'Channel to rectify
var ret%;                                 'Return dlg value (OK or cancel)

vh%:=FileOpen("", 0, 1+2);               'Open a data file
if vh%<0 then Message("No file opened");Halt endif;'Check OK

DlgCreate("Rectify channel");           'dialog box to select a channel
DlgChan(1,"Select channel to rectify", 1);'prompt for waveform
ret%:=DlgShow(channel%);                 'display dialog box
if ret%=0 then Halt endif; 'if user pressed Cancel
mmChan%:=MemChan(1, 0, Binsize(channel%));'Create waveform
                                         'with binsize same as channel%
ChanScale(mmChan%, ChanScale(channel%));'Set channel properties
ChanOffset(mmChan%, ChanOffset(channel%));
ChanTitle$(mmChan%,"Rectify");
ChanShow(mmChan%);                       'Make the new channel visible
DoData(mmChan%, channel%, 0, Maxtime());'Copy chan 1 to mmChan%
Interact("Press OK to save new channel", 36); 'Wait for OK
SaveChannel(mmChan%);

'This copies a waveform from channel oldChan% to newChan%
'It does not matter how many points need extracting.
func DoData(newChan%, oldChan%, sTime, eTime)
var maxSize%;
maxSize%:=Trunc((eTime-sTime)/Binsize(oldChan%))+1;
var arr[maxSize%];                       'Set array size required
var pts%;                                 'Number of points extracted
pts%:=ChanData(oldChan%, arr[], sTime, eTime); 'Get data points
Abs(arr[:pts%]);                          'rectify the data in the array
MemSetItem(newChan%, 0, sTime, arr[:pts%]);
end;

'This saves the memory channel to an unused channel on disk
func SaveChannel(chan%)
var list%[126];                           'List of unused channel numbers
ChanList(list%[],128);                     'Copy channel numbers into array
if list%[0]>0 then                          'If an unused channel
    MemSave(chan%, list%[1]);               'then save memchan into it
    ChanShow(list%[1]);
endif;
end;
```

Things to note in the final script:

1. Instead of automatically opening `demo.smr`, you are prompted to choose a data file. This is because a blank string has been placed in the `FileOpen()` command instead of `"demo.smr"`.
2. The script now puts up a dialog box prompting you to select a channel to rectify. This is because of the addition of the following four lines:

```
DlgCreate("Rectifiy channel"); 'dialog box to select a channel
DlgChan(1,"Select channel to rectify", 1);'prompt for waveform
ret%:=DlgShow(channel%);      'display dialog box
if ret%=0 then Halt endif;    'if user pressed Cancel
```

When these lines run, the dialog box appears and the script waits at the `DlgShow()` command until either *OK* or *Cancel* is pressed. If *OK* is pressed, `channel%` is made equal to the channel number selected and `ret%` is made equal to 1. If *Cancel* is pressed then `ret%` is made equal to 0 (hence the last line checks that *OK* has been pressed before continuing with the analysis).

3. The new memory channel is created with the same *bin size*, *scale* and *offset* as the selected channel and is given a title “Rectify”.
4. When `DoData()` is called, the start and end times that are passed are 0 and `Maxtime()`. `Maxtime()` is a function that returns the maximum time of the current view (in our case, the current view is the data file we opened, so `DoData()` will process the entire file).
5. In the `DoData()` function, the line

```
Abs(arr[:pts%]);          'rectify the data in the array
```

has been added before the array of data points is placed in the new memory channel. This function rectifies the values in the array. It is at this stage that any manipulation of the data in the channel should be performed. Spike2 has many powerful routines for manipulating data in arrays. Nearly all of them start with `Arr...` (e.g. `ArrConst()`, `ArrAdd()`, `ArrFilt()`) so it is worth looking these up in the Spike2 script manual. Also, many of the functions that can be used to transform one number into another can operate on arrays too. We have used `Abs()`, which usually returns the absolute value of a number, but can perform the operation on an array also (other examples of this are the `Sin()` and `Cos()` functions).

6. Finally, we have created the `SaveChannel()` function. The two main commands in this function are `ChanList()` and `MemSave()`. `ChanList()` fills an array with channel numbers of a particular type. The type depends upon the number passed to it. We have passed 128 which means get a list of *unused* channel numbers. The first element of the array is set to the number of channels placed in the list, so we check that there have been some placed in there before continuing. The `MemSave()` function saves the memory channel specified by the first argument to a channel number specified by its second argument. The second argument we have passed to it is `list%[1]` (the first channel number in the list of unused channels).

Using scripts on-line

Using scripts on-line

The turnkey tools for processing sampled data and stimulus generation that are available in Spike2 suffice for many applications. However, it is inevitable that there are situations where the specific data processing requirements of a particular application are not available as a menu option. You can customise Spike2 with the script language to perform tasks beyond those available from interactive operations.

The sample family of commands

The `Sample...()` family of script functions sets sampling configuration information, controls sampling and interacts with the sampling in various ways.

Sample setup

The following commands (and many others) operate on the sampling configuration to define sampling. They can be used so that the user does not have to manually load or set up a sampling configuration:

<code>SampleCalibrate()</code>	<code>SampleClear()</code>	<code>SampleComment\$()</code>
<code>SampleDigMark()</code>	<code>SampleEvent()</code>	<code>SampleLimitSize()</code>
<code>SampleLimitTime()</code>	<code>SampleMode()</code>	<code>SampleSequencer()</code>
<code>SampleTextMark()</code>	<code>SampleTimePerAdc()</code>	<code>SampleTitle\$()</code>
<code>SampleUsPerTime()</code>	<code>SampleWaveform()</code>	<code>SampleWaveMark()</code>

Sample control

The second group of commands control sampling in much the same way as the sampling control panel. They can usefully be used within toolbar button functions to start and stop sampling, and similar operations:

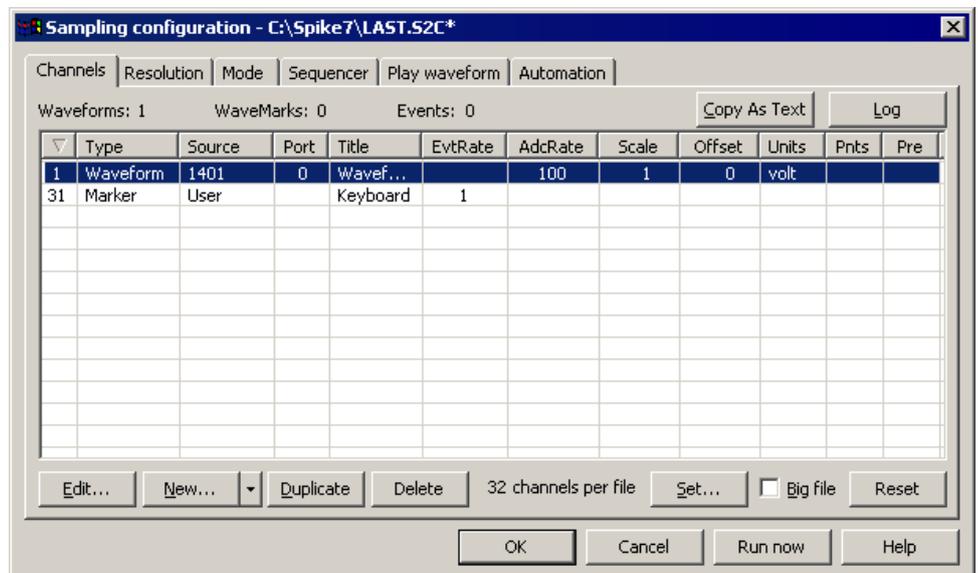
<code>SampleAbort()</code>	<code>SampleReset()</code>	<code>SampleStart()</code>
<code>SampleStop()</code>	<code>SampleWrite()</code>	

The final group of commands gets information on the sampling windows and sampling state, controls the output sequencer and writes keyboard and text markers to the data file:

<code>SampleHandle()</code>	<code>SampleKey()</code>	<code>SampleSeqVar()</code>
<code>SampleStatus()</code>	<code>SampleText()</code>	

Sampling configuration commands

The script below and on the next page generates a sampling configuration that matches the sampling configuration dialog:



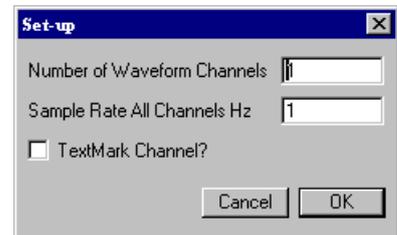
<code>SampleClear();</code>	Set a standard state
<code>SampleUsPerTime(10);</code>	Set the basic time unit to 10 microseconds
<code>SampleTimePerAdc(10);</code>	Set the number of time units per waveform sample. (100 μ s = 10 kHz overall rate)

<code>SampleMode(1);</code>	Set the sample mode to 1 (continuous)
<code>SampleLimitTime(-600);</code>	Set the Run time for sampling. In this case a negative value disables the limit.
<code>SampleLimitSize(-1024);</code>	File size to create before terminating sampling. Negative values disable the limit.
<code>SampleSequencer("");</code>	No output sequencer file.
<code>SampleWaveform(1,0,100);</code>	Set channel 1 as 100 Hz waveform on port 0
<code>SampleCalibrate(1," volt",1,0);</code>	Set channel 1 waveform calibration.
<code>SampleTitle\$(1,"Waveform");</code>	Set the waveform channel title.
<code>SampleComment\$(1,"Comment");</code>	Set the waveform channel comment.
<code>SampleTitle\$(31,"Keyboard");</code>	Set the keyboard channel title, channel 31.
<code>SampleComment\$(31,"Comment");</code>	Set a comment for Keyboard, channel 31.

An easy way to convert the settings in the sampling configuration window into a script is to go to the **Script** menu, turn recording on, use the **File** menu **New** command to create a new data file, then turn recording off to get a script matching your settings.

The script is a lot of typing to generate two channels, but it is easily extended. The code below creates a simple dialog as shown on the right. The variable `chNo%` sets the number of waveform channels we want to use and `rate` sets the rate to sample at:

```
var ok%, chNo%, rate, textMark%;
DlgCreate("Set-up");          'New dialog
DlgInteger(1,"Number of Waveform Channels", 1, 16);
DlgReal(2,"Sample Rate All Channels Hz", 1, 1000);
DlgCheck(3,"TextMark Channel?");
ok% := DlgShow(chNo%, rate, textMark%);
```



The `TextMark` box, if checked, determines whether a `TextMark` channel is added to the configuration or not. The following addition to the dialog code will result in the dialog settings being written to the sampling configuration.

```
var input%;
SampleClear();                'clear the existing configuration
SampleUsPerTime(2);          'set a time resolution
input% := chNo%-1;           'set input port for last channel
repeat                        'Repeat until all channels done
    SampleWaveform(chNo%, input%, rate); 'create new channel
    input% := input%-1;       'decrement the input counter
    chNo% := chNo%-1;        'decrement the channel number
until chNo% = 0;              'finish when no channels left
if textMark% = 1 then        'if TextMark selected then...
    SampleTextMark(200)      '...set with 200 characters
endif;
SampleUsPerTime(2);          'set a time resolution
SampleOptimise(1, 1, 7);     'Optimise ADC for a Power1401 mk II
```

`SampleUsPerTime()` sets the overall time resolution of the file. It is important to remember that if the maximum single channel rate is too small the desired sample rates for the channels will not be achieved. Either use `SampleTimePerAdc()` to set fixed total ADC rates or `SampleOptimise()` to find the best settings for a particular 1401 type.

It is not always necessary to write all the script commands to produce a complete sampling configuration; your script can be designed to work with whatever sampling configuration is loaded or it could start off with the loaded configuration and modify it in a limited manner based on user input.

Control of sampling

These script commands take the place of the various sampling control panel functions. A convenient way to make use of them is to provide toolbar buttons that provide the required functionality. This might appear to be a lot of work just to mimic the sampling control panel, but it puts the script in charge of everything and makes sure that the script code can keep track what is going on. An example of such a script might be:

```
var fh% := -1;                                'No sampling document yet
DoToolbar();                                  'DoToolbar does all the work
Halt;

Func DoToolbar()
ToolbarClear();                              'Remove any old buttons
ToolbarSet(1, "Quit");                        'Quit 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
    ... disable stop and reset buttons
return Toolbar("Your prompt", 0);
end;

Func Start%()                                'Start sampling button
fh% := FileNew(0, 1);                          'Create the new file
    ... check for errors here
SampleStart();                                'Start sampling
    ... enable stop and reset buttons
return 1;                                      'Leave toolbar active
end;

Func Stop%()                                  'Stop sampling button
SampleStop();                                  'Stop the sampling
View(fh%);
FileSaveAs("*.smr", -1);                       'Get user to save the new file
FileClose(0, -1);                              'Close the new data file
    ... disable stop and reset buttons
return 1;                                      'Leave toolbar active
end;

Func Reset%()                                 'Reset sampling button
SampleReset();                                 'Clear out all sampled data
SampleStart();                                 'Start sampling
return 1;                                      'Leave toolbar active
end;
```

This is not particularly elegant and we have omitted the code to enable and disable toolbar buttons, but the script does work, and makes sure that new data is saved to disk immediately after sampling finishes.

The script provides 4 buttons, 3 of which mimic the floating sampling control window and the 4th quits the script. This sampling is assumed to write data continuously to disk, it would be necessary to use the `SampleWrite()` command to switch on and off the sampling to pause the disk writing, presumably via an extra button (as below).

Writing data and information gathering

Now that you have implemented some control over the sampling, the script could also relay information about the sampling status and write to a file extra information not obtainable from the 1401.

For example we might want to insert a `TextMark` but this can only be achieved reliably if the `TextMark` channel is being stored to disk. To check the status and write information if disk write is on we can use code like:

```

ToolbarSet(7,"Textmark", Text%);      'add to the toolbar above
...
if SampleWrite(-1,30) = 1 then        'If disk write is active then
  SampleText("new text", MaxTime()); 'add TextMark at current time
endif;

```

The second section of script should be called from a toolbar button or at the moment when disk write is enabled so that comments can be added to the file and not lost.

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. The section of script to follow allows the user to select pause writing to disk on or off this can be accomplished by:

```

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

```

The `SampleWrite()` command is particularly interesting in that it can selectively disable writing of some or all data channels, so it is possible to choose a data channel and prevent it from being written whilst others are still being written. This could result in a file of 2 channels, neither of which is saved to disk at the same time. A copy of the completed version of this script given `sampling.s2s`.

Scripted analysis while sampling

Scripted analysis can be usefully employed in online situations where the software performs a particular task in response to a certain condition or to provide special analyses, for example measurements relative to a logged event.

Most aspects of scripted analysis operate in exactly the same manner when used online or offline, the only real difference is that the duration of the data file keeps on increasing. A key requirement is a mechanism that monitors the data being captured by Spike2 and determine if the correct time for action has been reached.

How not to do it It might be tempting to write a script such as that given below to test a condition and generate an appropriate response (assume we run the script having begun data capture with 1 event channel being sampled on channel 1):

```

var evcount%;                            'events in file so far
repeat                                    'start of monitoring loop
  evcount%:=Count(1, 0, Maxtime(1));    'Total number of events
until evcount% > 1000;                   'wait for > 1000 events
doanalysis();                             'then do the analysis
halt;

proc doanalysis()                         'analysis procedure
Message ("More than 1000 events!");      'trivial message
return
end;

```

This code does indeed detect the required number of events. However, the loop makes no provision for Spike2 to handle sundry tasks such as updating the display or reacting to mouse operations. Other Windows applications will still run (possibly rather slowly) and the loop will block other Spike2 activity such as online analysis.

Somewhat improved example using Yield()

A trivial solution to this problem is to return some time to Windows using the Yield() function, so that the loop looks like:

```
repeat                                     'start of monitoring loop
  Yield(0.05);                             'Pause script for c. 50 ms
  evcount%:=Count(1, 0, Maxtime(1));      'Total number of events
until evcount% > 1000;                   'wait for > 1000 events
```

This will give time back to Windows and the rest of Spike2 while the loop is executing. Yield() can be a satisfactory solution in some circumstances (mostly when you want a quick fix) but is best avoided as scripts designed in this manner tend to be inflexible and either block user interaction with the sampling or have trouble dealing with user actions. For example the script above would be singularly annoying if no events were being logged for some reason – there is no way to break out of the repeat loop.

Using a Toolbar idle function

A better solution is to use a toolbar idle function. This is a function just like the standard toolbar button functions, but it is executed whenever the system is not busy. We call this quiescent state *idling* hence the term *idle function*.

A toolbar idle function is defined by defining button number 0. The linked function is called repeatedly while the toolbar is active and the system is not busy. Button 0 is not displayed on the toolbar as the function is called automatically. Like the other button functions, the idle function returns 1 to keep the toolbar active and 0 to cause the toolbar to terminate. Here is an example script showing this method:

```
OnScr1.s2s ToolbarSet(0, "", idle%);          'idle routine entry
ToolbarSet(1, "Stop", stopsampling%);      'button to stop sampling
ToolbarEnable(1, 0);                       'disable stop button
ToolbarSet(2, "Start", startsampling%);    'button to start sampling
Toolbar("Sampling control", 1023);        'show toolbar
Message("Sampling is over");              'tell user we are done
halt;                                       'all done!

func startsampling%()                      'button 1 function
FileNew(0, 3);                             'create new file
SampleStart();                             'start off sampling
ToolbarEnable(1, 1);                       'enable stop button
ToolbarEnable(2, 0);                       'disable start button
ToolbarText("Sampling in progress");       'update toolbar message
return 1;                                  'keep toolbar active
end;

func stopsampling%()                      'button 2 function
SampleStop();                              'finish sampling
ToolbarText("Stopped");                    'update toolbar message
return 0;                                  'stop toolbar
end;

func idle%()                              'idle routine
if samplestatus() = 2 then                'if we are sampling
  if maxtime() > 10 then                  'if sampled > 10 seconds
    SampleStop();                        'stop sampling
    return 0;                            'stop toolbar too
  endif;
endif;
return 1;                                  'leave toolbar running
end;
```

The `MaxTime()` function used above returns the highest time in the data file and is the simplest way of keeping track of the progress of data acquisition.

Idle function design

The idle function is called many times a second and should be designed to do the minimum necessary each time it is called and then return, relying upon its being called again soon. If the idle function often takes a significant time to execute Spike2 will become somewhat unresponsive so it is important to throttle activity and/or to test to see if anything actually needs doing. It is, please note, not a problem if the idle function uses a bit of time to do some analysis on occasion, your aim should be to spread any analysis out so that any individual burst of analysis is fairly short:

```
var tDone;                                'Last time analysed

' This should be done as part of starting or restarting sampling
tDone := 0;                                'No analysis done yet

' The toolbar idle function analyses every two seconds
Func Idle%()                               'idle function
if SampleStatus()=2 then                   'if we are sampling
    if Maxtime() >= tDone+2 then          'and have 2 seconds unanalysed
        DoAnalysis(tDone);               'Analyse starting at last time
        tDone += 2;                       'Move the time done on
    endif;
endif;
return 1;                                  'leave toolbar running
end;
```

This function uses the global variable `tDone` to keep track of the time that analysis has reached and, each time there is at least two seconds of un-analysed data available, calls the analysis function and moves on `tDone` by two seconds. While the analysis function could take an appreciable time to execute, it is only called every two seconds so Spike2 should appear to run smoothly and responsively. The trick to creating a good Idle function is to make sure you test to see what needs doing and then do just that. If nothing needs doing, do nothing.

The `tDone` variable and any other variables that are needed to keep track of what is going on in the idle function must be global (that is, declared outside of any functions) to ensure that they are accessible from the idle function and that variable values persist between one idle function call and the next. If `tDone` was local to `Idle%()` (that is, declared within `Idle%()`) it would be re-created anew holding the value zero each time the idle function was called and the time that analysis had reached would be forgotten.

Idle functions are also available with script-generated dialogs so you could produce a script which displays a dialog throughout sampling with the dialog idle function managing analysis and dialog controlled values continuously available for editing.

You should bear in mind that though the idle function may be called many times a second, there may also be occasions when it is not called for a long period of time, mostly if aspects of Windows become busy. So if you are relying upon the idle function detecting a situation very quickly, you may encounter occasional delays.

Managing complex idle behaviour

It is often the case that one needs rather complex behaviour in the idle function, for example one could imagine an analysis regime that waited until the data on a waveform channel was above a given threshold and then, for each stimulation event seen while the waveform remains above the threshold, analyse the following two seconds of data. Doing this within an idle function can be difficult & complex and if you change to using a still more complicated analysis the code and control variables required to keep track of what the idle function is doing can become unmanageable, or at least impossible to change in a straightforward manner.

A powerful programming technique for dealing with this situation is called a state machine. To use this we start by defining the required behaviour as a set of states that the system can be in, with specific activities happening in each state and defined circumstances that control movement from one state to another. So using the example above we might have:

State Activity

- | | |
|---|---|
| 0 | Examine latest waveform data, move to state 1 and save current time as stimulus time if above threshold. |
| 1 | Have we had a stimulus event since the stimulus time? If so move to state 2 and save the event time as the stimulus time. |
| 2 | Has 2 seconds elapsed since the stimulus time? If so then analyse starting at the stimulus time and return to state 1. |

The way you would implement this would be to use a global variable to hold the state along with other variables holding timing information, the state variable is then used in a case statement to control activity (the complete script is given in the example code):

OnScr2.s2s

```

var state%;
var tStim;

' the state control
' last stimulus time

' This should be done as part of starting or restarting sampling
state% := 0;
tStim := -1;
' looking for threshold
' no stimuli yet

' The toolbar idle function executes the state machine
func Idle%()
' idle function
if SampleStatus()= 2 then
' if we are sampling
var now;
now := MaxTime();
' save the current time
docase
case state% = 0 then
' state 0 look for threshold
if chanvalue(1, now) >= thresh then
tStim := now;
' save current time for search
state% := 1;
' move onto stimulus search
endif;

case state% = 1 then
' state 1 look for stimulus
var stim;
stim := NextTime(3, tStim);
' look for a stimulus
if stim > tStim then
' got one?
tStim := stim;
' save current time for search
state% := 2;
' move onto post-stim wait
endif;

case state% = 2 then
' state 2 wait 2 seconds & process
if now >= tStim+2 then
' reached time?
DoAnalysis(tStim);
' analyse starting at last stim
state% := 1;
' search starting at last stim
endif;
endcase
endif;
return 1;
' leave toolbar running
end;

```

You should find it fairly easy to see what the code does, which is a good demonstration of the coding clarity that a state machine provides. There is also an error in this design – the code does not respond to the waveform level dropping below the threshold. For correct behaviour we should define state 1 as:

- 1 Check waveform and move to state 0 if below threshold, otherwise if we had a stimulus event since the stimulus time move to state 2 and save the event time as the stimulus time.

and adjust the state 1 code to:

```

case state% = 1 then      'state 1 look for stimulus
  if chanvalue(1, now) < thresh then
    state% := 0;        'back to 0 if below thresh
  else
    var stim;           'otherwise search for stim
    stim := NextTime(3, tStim); 'look for a stimulus
    if stim > tStim then 'got one?
      tStim := stim;   'save current time for search
      state% := 2;    'move onto post-stim wait
    endif;
  endif;
endif;

```

If the analysis is only to be performed if the waveform stays above `thresh` for the entire 2 seconds it would also be necessary to make similar changes to state 2. This shows the other advantage of basing your design on a state machine – organising things in this way makes it much easier to adjust or extend the behaviour. In particular you can extend the behaviour by adding new states – one could imagine a second analysis regime that could be handled by states 3, 4, 5 and 6; the rest of the script could then switch between analysis methods by setting the state to 0 or 3 and adjusting the relevant control variables.

A skeleton on-line analysis script

Many on-line scripts are very similar. The big difference between them is what happens in the idle function. Here we shall look at a skeleton on-line script. This script has buttons to start and stop sampling (as before), commands to create a new data file for sampling, and a standard `Idle()` function that can be customised to provide the analysis you require. The toolbar commands are now in a separate function so the main body of the script is very small.

```

OnSkel.s2s 'This is a skeleton on-line script. It has a toolbar from which
           'you can open a new data file and start and stop sampling.
           'The log view at the bottom of the screen can store extracted
           'values. On-line analysis should be placed in the idle function
var data%;           'Handle of new data file
var sTime;          'last time we used the idle function
ToolbarVisible(1);  'Make toolbar visible always
New%;               'Set up new sampling window
DoToolbar();        'Do the toolbar

func New%()         'New sampling window
View(LogHandle());  'Make log view the current view
EditSelectAll();    'Select all text in log view
EditClear();        'Delete it
Window(0,80,100,100); 'Display it at the bottom of the screen
WindowVisible(1);   'Make it visible

if data%>0 then     'If already a data view open then
  View(data%);      'Close it
  FileClose();
endif;

data%:=FileNew(0,1); 'Open a new data file for sampling

```

```

if data%<0 then Message("Cannot open new data file");Halt endif;
DrawMode(-1,2);           'Set event draw mode to lines
Window(0,0,100,80);      'Make data window in top bit of screen
XRange(0,10);

FrontView(LogHandle());   'Bring the Log view to the front
FrontView(data%);        'Bring the data view to the front
ToolbarEnable(3,0);       'Disable "Sample stop" button
ToolbarEnable(2,1);       'Disable "Sample stop" button
ToolbarText("SAMPLE START to sample");
return 1;
end;

proc DoToolbar()
ToolbarSet(0,"",Idle%);   'Call Idle%() when there is free time
ToolbarSet(1,"Quit",Quit%); 'Set up toolbar buttons
ToolbarSet(2,"Sample start", Start%);
ToolbarSet(3,"Sample stop", Stop%);
ToolbarSet(4,"New file", New%);
ToolbarEnable(3,0);       'Disable "Sample stop" button
Toolbar("SAMPLE START to sample", 1023); 'Wait for Quit
end;

func Quit%()              'If "Quit" is pressed
SampleStop();             'Stop sampling
return 0;                  'leave toolbar
end;

func Start%()             'If "Start" is pressed
SampleStart();            'Start sampling
sTime := 0;                'Reset the analysis time reached
ToolbarEnable(4,0);       'Disable "New file" button
ToolbarEnable(3,1);       'Enable "Sample stop" button
ToolbarEnable(2,0);       'Disable "Sample start" button
ToolbarEnable(1,0);       'Disable "Quit" button
ToolbarText("Press SAMPLE STOP to stop sampling");
return 1;                  'Stay with toolbar
end;

func Stop%()              'If "Stop" is pressed
SampleStop();             'Stop sampling
ToolbarEnable(4,1);       'Enable "New file" button
ToolbarEnable(3,0);       'Disable "Sample stop" button
ToolbarEnable(1,1);       'Enable "Quit" button
ToolbarText("Press FILE NEW to capture more data");
return 1;                  'Stay in toolbar
end;

func Idle%()              'Idle function called when PC has time
if (SampleStatus() = 2) then
  var eTime;
  View(data%);            'Switch to new data view
  eTime := Maxtime();     'Get the current maximum time
  if (eTime >= (sTime + 1)) then 'If we have enough new data

    'CODE TO ANALYSE DATA BETWEEN sTime and eTime

    sTime:=eTime;        'Update time reached
  endif;
endif;
return 1;                  'Stay in toolbar
end;

```

You can now change the sampling configuration after running the script and then use the "New file" button to implement the changes. The "New file" button calls the user defined

function `New%`(). This function first checks to see if `data%>0` (i.e. if a sampling window has already been created. This will actually always be the case except for the first time this function is called), if it is, it closes the data file associated with the handle `data%`. Spike2 automatically checks, when `FileClose()` is called, whether there is any data in the file and if there is offers to save the file before closing it.

When the `Toolbar()` function is called, it places the text “Press SAMPLE START to commence sampling” into the message section of the toolbar. The message displayed here can be changed from within the script using the `ToolbarText()` function. We have added `ToolbarText()` calls to the script so the message changes to reflect the current options available to the user.

Finally lets look at the `Idle()` function. There is a global variable in the script called `sTime`. This is used to keep track of where we were in the data file the last time the `Idle` function carried out some analysis. At the beginning of `Idle()` activity while sampling a local variable `eTime` is set to the current maximum time in the file. We now have two variables available to us; `eTime` is now and `sTime` is the last time we looked at the file. `Idle` functions often need only operate when there is a significant amount of new data available, in the example code it checks for there being at least one seconds-worth of new data but this check could be changed as required. Having carried out whatever processing is required `sTime` is made equal to `eTime` ready for testing next time the `Idle` function is called, again this might need to be different according to the particular analysis requirements – for example you might simply add 1 to `sTime` if you wanted to process data in precisely one second chunks.

Control of stimulation

It is often the case that you wish to monitor incoming data and respond to a condition with a signal output from the 1401. Spike2 features an output sequencer which can generate outputs from the 1401. The complete details of the sequencer are covered in the Spike2 documentation, here we will only look at interaction between a script and the sequence.

A script provides control over a sequence at two levels. It can activate specific sequences that are described in the sequence file and it can vary the characteristics of the signal output (waveform amplitudes, inter-pulse intervals, etc.) by passing data across to the sequence in a number of ways: sequencer variables, sequencer table data and arbitrary waveform data.

Control of sequencer execution

We can use script commands to activate specific output sequence operations by causing sequencer execution to jump to a specific location. Consider the following output sequence file:

```
AandB.pls
SET 1, 1, 0           ;1 ms steps, default DAC scaling
SP:  HALT             ;suspend sequence until key press
AA:  'A  DAC  0,5      ;Output 5 volts on DAC port 0
      DELAY 1000      ;Hold for 1000 milliseconds
      DAC  0,0        ;Reset DAC port 0 to 0 volts
      JUMP  SP        ;Suspend sequence
BB:  'B  DIGOUT [00001111] ;Output TTL high on 4 bits
      DELAY 2000      ;Hold for 2000 milliseconds
      DIGOUT [00000000] ;Set all digital outputs to zero
      JUMP  SP        ;Suspend sequence
```

Sequence execution runs from the first line starting at the moment that data acquisition begins. However, in this sequence file, the `HALT` command at the beginning suspends the execution of the instruction list and no output occurs. However, pressing `A` on the

keyboard activates the sequence starting at the line where the 'A' field has been given and pressing B activates the sequence starting at the line where the 'B' field has been given.

The `SampleKey()` script function simulates a user key press during data acquisition. We can use `SampleKey()` in a on-line script to activate a particular part of the sequence:

```
OnScr3.s2s var rawdata%;           'the raw data file window handle
SampleSequencer("AandB.pls"); 'set the sequence file
rawdata% := FileNew (0, 3);    'create a new data file
SampleStart(0);               'start data capture
ToolbarSet(1, "Run A", seqa%); 'assign button 1 to seq output A
ToolbarSet(2, "Run B", seqb%); 'and button 2 to B
ToolbarSet(3, "Quit");        'don't forget quit
Toolbar("Select output", 1028); 'display toolbar

func seqa%;                   'do sequence A
View(rawdata%);              'make raw data view active
SampleKey("A");               'simulate keypress
return 1
end;

func seqb%;                   'do sequence B
View(rawdata%);              'make raw data view current
SampleKey("B");               'simulate keypress
return 1
end;
```

It is important to ensure that the current view is the raw data file when using the `SampleKey` command. This is because keyboard markers are only registered when this view is active. There is also another script function, `SampleSeqStep()`, which returns the current step in the sequence being executed. If you look at our sequence code you will see that at the end of the stimulus it jumps to the `HALT` instruction at the start so that the current sequencer step it always zero if the sequencer is halted. This would allow us to prevent a stimulus from being triggered if the previous stimulus was not yet finished:

```
func seqa%;                   'do sequence A
if SampleSeqStep() = 0 then   'if the sequence is halted
    View(rawdata%);          'make raw data view active
    SampleKey("A");           'simulate keypress
endif;
return 1
end;
```

Note that, though I am demonstrating this using a text sequence, this control mechanism is also easily used with sequences created using the graphical sequence editor – you just have to enter key values for those sections of the output sequence which you want to be able to jump-to.

Control of sequencer variable values

Our sequence file limits us to the presentation of two fixed stimulus types. Sometimes, it is necessary to generate a much wider range of stimuli, with varying delays and repetitions. To construct a sequence file covering all possible sequence arrangements would be tedious were it not prohibited by the finite number of keyboard characters available to trigger each sequence and the maximum number of instructions that can be used in a sequence file (8192).

We can work round this problem by using variables in the sequence file. Variables are 32-bit integer numbers (a very large range of whole numbers) which can control aspects of sequencer operation. 256 variables are available for use in a sequence file, they are called v1 to v256. Variables can set the voltage amplitude of a DAC, a delay, a loop

count and so forth. Variables are initialised with zero and can be changed internally by the sequence or by the script by using the `SampleSeqVar()` function.

The bulk of this discussion will cover the use of sequencer variable values to pass information into a sequence but do be aware that they can also be extremely useful in allowing the script to read back information. For example in the example above we used `SampleSeqStep()` to find out if the sequence was in the middle of generating a stimulus. It is usually much neater to have the sequence signal 'busy' by setting a variable to 1 (using the `MOVI` sequencer instruction) and signal 'not busy' by resetting the variable back to zero. The test for OK to start a stimulus used above then becomes:

```
func seqa%;
if SampleSeqVar(40) = 0 then
    View(rawdata%);
    SampleKey("A");
endif;
return 1
end;
```

'do sequence A
'v40 will be 1 while busy
'make raw data view active
'simulate keypress

and the resulting arrangement is both more flexible (the sequence could indicate what stimulus is in progress by using different variable values) and would not break if you rearranged the sequence structure and thereby changed the step numbers.

Output control using sequencer variables

Consider the 'A' section of our `AandB` sequence with variables substituted for the amplitude and delay values. This would look like:

```
AA:   'A   DAC    0,v1      ;Set DAC 0 using v1
      DELAY v2          ;Hold for v2 ticks
      DAC    0,0        ;Reset DAC 0 to 0
      JUMP   SP         ;Suspend sequence
```

A suitable value for `v1` must be set by the script using the `SampleSeqVar` command before the output pulse can be generated. The variable value must be a 32-bit integer number (a whole number falling in the range -2147483648 to 2147483647). If the DAC output range is +/- 5 volts, then the minimum value of -2147483648 gives a DAC output of -5 volts, a value of 0 gives a DAC output of zero volts and the maximum value of 2147483647 gives a value of 4.999 volts. In general, we calculate the value to assign to `v1` from the desired voltage (assuming a 5-volt 1401) using the equation:

```
integer value := Round((voltage * 6553.6) * 65536.0);
```

but the calculation can be hard to make bullet-proof, for example when faced with out-of-range voltage values. To assist you with this task CED have produced a library of useful script functions called `seqlib.s2s` which contains various functions used to calculate sequencer variable values, this library is installed with every copy of Spike2. You can include `seqlib.s2s` in your script and then make use of the `VDAC32%` function to calculate your variable value thus:

```
integer value := VDAC32%(voltage, 1, 0);
```

where the 1 and the 0 correspond to the DAC scaling values set in the sequence, you can adjust these parameters as needed to deal with different DAC scaling such as using a 10 volt system, or outputs handled as millivolts.

The `DELAY` variable, `v2`, is a value in the range 1 to 2^{32} which sets the delay in sequencer ticks, the actual delay is one tick more than the variable value. The sequencer tick length is defined by the `SET` command in the output sequencer file and defaults to 1 millisecond, allowing delays from 2 ms to more than four million seconds with the default tick length.

Loops are created in a sequencer file by loading a variable and then repeatedly performing an instruction or group of instructions and decrementing the variable until the

it reaches zero. The DBNZ (Decrement and Branch if Not Zero) sequencer instruction provides an easy way to do this.

Creating variable pulse trains

The following output sequencer file is used:

```
OnScr4.pls      SET 1, 1, 0      ;1 ms steps, default DAC scaling
SP:            HALT      ;suspend sequence until key press
AA:   'A      MOV   v4,v1  ;copy v1 value for use
              DAC   0,v2  ;set DAC 0 using v2
              DELAY 100   ;hold pulse for 100 milliseconds
              DAC   0,0   ;reset pulse to 0
              DELAY v3    ;inter-pulse interval
              DBNZ  v4,AA  ;repeat loop at AA until v1 is 0
              JUMP  SP     ;go to suspend sequence
```

The script gives the user control over the variables and the onset of the pulse train using a script-created dialog:

```
OnScr4.s2s      #include "seqlib.s2s"
var rawdata%;      'raw data window handle
rawdata% := FileNew(0, 3);      'create new data file
SampleStart();      'start data capture
ToolbarSet(1, "Quit");      'button to quit
ToolbarSet(2, "Start train", starttrain%); 'Start sequence button
ToolbarSet(3, "Configure train", config%); 'button to configure
Toolbar("Select option", 1028); 'display toolbar

func starttrain%();      'generate sequence
if SampleSeqStep() = 0 then      'if the sequence is halted
    View(rawdata%);      'make raw data view active
    SampleKey("A");      'simulate keypress
endif;
Return 1
end;

func config%();
var rep%, amp, int%;      'reps, amp and pulse int
var amp2%;      'amp as variable value
DlgCreate("Pulse parameters");      'create parameters dialog
DlgInteger(1, "Repetitions", 1, 10);      'allow 1 - 10 repetitions
DlgReal(2, "Amplitude", -5, 5);      'allow amplitude +/- 5 v
DlgInteger(3, "Interval (ms)", 10, 10000); 'allow 10ms - 10s
if (DlgShow(rep%, amp, int%) = 1) then      'get variables, 1 if OK
    amp2% := vdac32(amp, 1, 0);      'amplitude as variable value
    SampleSeqVar(1, rep%);      'assign values to variables
    SampleSeqVar(2, amp2%);
    SampleSeqVar(3, int%-1);
endif;
return 1      'return
end;
```

Variables can be used instead of fixed values in most sequencer instructions, see the sequencer instruction reference in the documentation for details of variable use. There are also a number of sequencer instructions for directly manipulating and testing variable values.

Control using sequencer table data

The sequencer variables are very useful but a bit limited, for example there are only 256 of them which rather limits the information that can be stored. With a lot of care the variable values can be updated 'on the fly' while a stimulation sequence is underway but it can be very hard to ensure this is done at the right time and makes for an extremely complex script. To help with this sort of issue the sequencer also incorporates a table, a single variable-length array that can be used to store large amounts of information. Many

sequencer instructions can use information from the table in the same manner as values held in a variable, or table data can be loaded, stored, added or subtracted to/from variables. Table data is accessed using a table index held in a variable plus an optional offset (the first location in the table is at index 0); there is a useful `TABINC` instruction which is used to move a variable used for indexing through the table and detect that the end of the table has been reached.

Consider the previous sequence that generated a sequence of pulses based on variable values. We can alter it to generate a long sequence of pulses using values taken from the table with sets of four table values defining a set of pulses thus:

Index	Function
0	Count of pulses, or zero to stop output
1	Pulse amplitude value scaled as for a variable
2	Pulse duration in sequencer ticks
3	Inter-pulse interval in sequencer ticks

A sequence to use table data to do this might look like:

```
OnScr5.pls      SET 1, 1, 0      ;1 ms steps, default DAC scaling
                TABSZ 1000    ;Enough space in the table

SP:            HALT          ;suspend sequence until key press

                'A          MOVI v2,0      ;start at the beginning of table
L1:            TABLD v1,[v2+0] ;Load pulse count into v1
                BEQ v1,0,SP   ;Stop if zero pulses
L2:            DAC 0,[v2+1]   ;set DAC 0 using table data
                DELAY [v2+2]  ;hold pulse for correct period
                DAC 0,0       ;reset pulse to 0
                DELAY [v2+3]  ;inter-pulse interval
                DBNZ v1,L2    ;repeat loop at L2 until v1 is 0
                TABINC v2,4,L1 ;move on 4 and loop to L1
                JUMP SP       ;stop if reached table end
```

note the use of `TABSZ` to define the table size, `TABLD` to load a variable value from the table and the use of `[v2+x]` to use a table value directly within an instruction. The `TABINC` instruction adds 4 to `v2` and branches to the `L1` label as long as the resulting value is a valid table index (0 to 999 in this case) which provides protection against running `v2` past the end of the table.

Obviously a very wide range of pulses could be generated by this output sequence given suitable table data, here is a script that defines a variable set of pulses in the middle of two fixed sets :

```
OnScr5.s2s #include "seqlib.s2s"
var rawdata%;
rawdata% := FileNew(0, 3);
SampleStart();
ToolbarSet(1, "Quit");
ToolbarSet(2, "Start pulses", starttrain%);
ToolbarSet(3, "Configure pulses", config%);
Toolbar("Select option", 1028);

func starttrain%;
if SampleSeqStep() = 0 then
    View(rawdata%);
    SampleKey("A");
endif;
Return 1
end;
```

```

func config%;
var table%[1000];           'Our table data
var rep%, amp, dur%, int%;  'reps, amp, duration and interval
var amp2%;                  'amp as variable value
DlgCreate ("Pulse parameters"); 'create parameters dialog
DlgInteger (1, "Repetitions", 1, 10); 'allow 1 - 10 repetitions
DlgReal (2, "Amplitude",-5, 5); 'allow amplitude +/- 5 v
DlgInteger (3, "Duration (ms)",2,10000); 'allow 2ms - 10s
DlgInteger (4, "Interval (ms)",10,10000); 'allow 10ms - 10s
if (DlgShow (rep%, amp, dur%, int%) = 1) then 'edit, 1 if OK
    table%[0] := 2;           'start with two pulses
    table%[1] := vdac32(2, 1, 0); 'two volts
    table%[2] := 9;           '10 milliseconds long
    table%[3] := 49;          '50 milliseconds apart
    table%[4] := rep%;        'second part as set by user
    table%[5] := vdac32(amp,1,0); 'amplitude as variable value
    table%[6] := dur%-1;      'Timings as entered
    table%[7] := int%-1;      '-1 adjusts for DELAY
    table%[8] := 4;           'finish with four pulses
    table%[9] := vdac32(4,1,0); 'four volts
    table%[10] := 9;          'ten milliseconds long
    table%[11] := 49;         '50 milliseconds apart
    table%[12] := 0;          '0 to signal end
    SampleSeqTable(table%[]) 'transfer data into 1401
endif;
Return 1                     'return
end;

```

Control of arbitrary waveforms

One of the facilities within a Spike2 sampling configuration is a list of stored waveforms that can be replayed at any point, either in response to a key press or triggered by sequencer actions. A script can set up waveforms before sampling with the `PlayWaveAdd()` script function or can use of waveforms that are already defined. In both cases waveform output can be triggered directly or indirectly by `SampleKey()` to trigger a waveform or by causing the sequencer to jump to a given location.

You can read waveforms from a data file or use the script to synthesise them as arrays of data; generating waveform data directly is a particularly powerful technique and much used for generating audio stimuli.

You can also define waveform play areas that initially contain no data and load these (or any area) with new data while sampling is in progress with the `PlayWaveCopy()` function. `PlayWaveCopy()` can even update parts of a waveform play area while data from other parts of the area is being output to the DACs. By playing the area circularly (so output wraps round to the area start once the end is reached) and monitoring replay progress by using `PlayWaveStatus()`, you can generate continuous waveform output of any duration. This avoids all limits to waveform play area sizes but does require that the waveform output rate be slow enough, and script execution fast enough, for the script to keep up.

The **Spike2 scripts** section of the **Downloads** section of the CED web site (www.ced.co.uk) contains a number of scripts that generate and play out arbitrary waveforms. Example scripts that provide continuous waveform output by repeatedly updating a waveform play area are also available – contact CED with details of your requirements if you need one of these.

Advanced topics

Advanced topics

This chapter is based on a talk given at the 1996 User Day in which we attempted to cover the topics of memory channels, external text files, binary data files and serial line use in around one hour! History does not record how far the presenter got in the allotted time, but it certainly wasn't to the end.

What is a memory channel?

A memory channel is a standard Spike2 data channel held in computer memory. Within the channel, data is stored as a list of items that can be accessed either by their index or their time, by the memory channel-specific routines. The data items can also be accessed by the same script routines that access standard Spike2 data channels.

Memory channels are created by the `MemChan()` function. The following script shows how you can create a memory channel and add and subtract data items.

```
'MemCh1.s2s
var vh%, mc%;                               'view handle, memory channel
vh% := FileOpen("Demo.smr", 0, 0);         'Open the file invisibly
if (vh% <= 0) then Message("No DEMO file!"); halt endif;
ChanHide(-1);Window(0,0,100,100);         'Hide all channels
WindowVisible(1);Draw(0, MaxTime());     'show the lot

mc% := MemChan(2);                          'Create an event channel
if mc% < 0 then Message("No memory channel!"); halt endif;
ChanShow(mc%);                              'and show it
DrawMode(mc%,2); DoDraw();                 'Lines mode
DrawMode(mc%, 5, 2);DoDraw();              'Rate mode
DrawMode(mc%, 6, 2);DoDraw();              'Mean frequency
DrawMode(mc%, 7, 3);DoDraw();              'Inst Frequency
FileClose();
halt;

Proc DoDraw()
Seconds(0);                                 'zero the timer
While Seconds() < 10 do                    'run for 10 seconds
    MemSetItem(mc%, 0, Rand()*Maxtime()); 'add an item
    Draw();                                'let us see it
WEnd;
var i%;                                    'to count events
i% := Count(mc%, 0, Maxtime());           'count items added to buffer
while i% do
    MemDeleteItem(mc%, i%*Rand()+1);      'kill random item
    Draw();                                'update the display
    i% := i%-1;                            'reduce count of items
Wend;
End;
```

In this script we create the simplest possible type of channel, an event channel. This channel type needs no other information, so the create call is very simple and returns the channel number of the new channel (or an error, for example if we ran out of memory). Memory channels are created invisible, so you have the opportunity to change the display mode and set a suitable y axis range (if appropriate) before making the channel visible.

The example spends 10 seconds adding events at fairly random times into the file using:

```
MemSetItem(mc%, 0, Rand()*Maxtime()); 'add an item
```

You add event-based data items to a memory channel by specifying a time. If you also specify an index, the item at the index is deleted and the new item is added at the position specified by the time. The script `MemCh2.s2s` shows what happens when you also supply an index when adding data items. The modified script adds events for 1 second, then for the next 9 seconds it replaces randomly selected events with new ones. Finally it removes the events in order, from the start of the buffer.

```
'MemCh2.s2s
...
...   This code is the same as the previous example
...
Proc DoDraw()
Seconds(0);                               'zero the timer
While Seconds() < 1 do                     'run for 1 second adding
  MemSetItem(mc%, 0, Rand()*Maxtime()); 'add an item
  Draw();                                  'let us see it
WEnd;

var i%;
i% := Count(mc%, 0, Maxtime());           'Count events
while Seconds()<10 do                     'replace random events
  MemSetItem(mc%, Rand()*i%+1, Rand()*Maxtime());
  Draw();
WEnd;

while MemDeleteItem(mc%, 1)=1 do          'kill all items
  Draw();                                  'update the display
Wend;
End;
```

We delete items using:

```
MemDeleteItem(mc%, i%*Rand()+1);         'kill random item
```

Uses of memory channels

The main uses we have found for memory channels (so far) are:

1. As a target for the Analysis menu Memory buffer commands (most often used to pick peaks or level crossings in a signal).
2. To convert channels from one type to another.
3. To edit events out of or into an event-based channel.
4. To combine waveform channels, or to produce a waveform channel that is a function of other channels. For example to produce a Painter filter of EMG data.
5. To store tables of data sampled at discrete times, for example blood gas data, using RealMark data.
6. To display an on-line heart rate extracted from an ECG signal.
7. To import data from a foreign data file to create a new Spike2 file without sampling.

MemChan()

The `MemChan()` function creates a new channel in memory and attaches it to the file in the current time view. You can have up to 26 memory channels belonging to a file. This command gives the new channel default (i.e. useless) channel title, comment and units (if relevant). If the channel is a waveform, the scale is set to 1 and the offset to 0. It is up to you to set the values you want for the channel.

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

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

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

For codes 2, 3 and 4, you need no more arguments. The other channel types need additional information to create the channel.

- `size%` Used for TextMark, RealMark and WaveMark channels to set the maximum number of characters, reals or waveform points to attach to each item. This is ignored for all other channel types and should be set to 0.
- `binsz` Used for waveform, RealWave and WaveMark data to specify the time interval between the waveform points. This is rounded to the nearest multiple of the underlying time resolution. If you set this 0 or negative, the smallest bin size possible is set. Remember that the time resolution of a file is the number of microseconds per clock tick and that the waveform sampling resolution is set to an integral multiple of this. Both these values are set (and are fixed) when the file is created, either by sampling, when these values are set from the sampling configuration, or when the file is created with `FileNew(7, mode%, upt%, tpa%, maxT)`.
- `pre%` The number of pre-trigger points for WaveMark data. This value is used when importing data from a waveform channel into a WaveMark channel. You can get this value back with the `MarkInfo()` command. The time stamp of a WaveMark data item is the time of the first waveform point, not the time of the trigger point.

The function returns the channel number of the newly created channel, or 0 if there are no free channels, or a negative error code. The channel numbers start at 101, and the system always uses the lowest free number, but you should not rely on this as it might change in the future.

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

```
'MemCh3.s2s
func CopyWave%(wChan%) 'Copy waveform to a memory channel
var mc%;
if ChanKind(wChan%)<>1 then return 0 endif; 'Not a waveform!
mc% := MemChan(1,0,BinSize(wChan%));      'Create waveform channel
if mc%>0 then                               'Created OK?
  ChanScale(mc%, ChanScale(wChan%));      'Copy scale...
  ChanOffset(mc%, ChanOffset(wChan%));    '...and offset...
  ChanUnits$(mc%, ChanUnits$(wChan%));    '...and units
  ChanTitle$(mc%, "Copy");                 'Set our own title
  ChanComment$(mc%, "Copied from channel "+Str$(wChan%));
  MemImport(mc%, wChan%, 0, MaxTime());   'Copy data
  ChanShow(mc%);                           'display new channel
endif;
return mc%; 'Return the new channel number
end;
```

MemImport() This function imports data into a channel created by `MemChan()`. The function not only imports data from a channel of the same type, but it can convert data from channels of a different type. There are some restrictions on the type of data channel that you can import from, depending on the type of the destination channel. The table on the next page describes all the combinations.

Although this seems complicated, in fact all it means is that all the event and marker channels can be copied to each other, but that the information transferred is the lowest common denominator of the two channel types. Missing data is padded with zeros. Waveform data is compatible with itself, but only if the two channels have the same

sampling rate. You can also extract events or markers from waveform data using peak search and level crossing techniques.

Destination	Source	Restrictions
Waveform	Waveform All others	Data copied, but must match sample interval Not available, see <code>EventToWaveform()</code>
Event	Waveform All others	Can extract event times Times are extracted from the channel
Level	Waveform All others	Can extract event times Times are extracted from the channel. First time in destination is assumed to be a low to high transition
Marker	Waveform Event, Level All others	Can extract event times, coded for peak/trough etc. Marker codes all set to 0 Marker codes are copied
TextMark	Waveform Event, Level TextMark All others	Can extract event times, coded for peak/trough etc., empty strings Copies times, marker codes set 0, empty strings Copies all data, strings may be truncated if too long Copies marker information, empty strings
RealMark	Waveform Event, Level RealMark WaveMark All others	Can extract event times, coded for peak/trough etc. Times copied, marker codes and reals set to 0 Copied, real data truncated or zero padded as needed Copied, waveform to reals, padded/truncated Marker portion copied, reals filled with 0
WaveMark	waveform Event, Level WaveMark All others	Special option, event channel marks waveforms Copies times, marker codes set to 0, waveform set 0 Copies all data, waveform truncated/zero padded Copies marker, waveform filled with zeros

You can also convert waveform data to WaveMark data with a special option that chops out sections of waveform data based on event times on a third channel.

```
Func MemImport(memC%, inCh%, start, end {,mode% ,time, level});
Func MemImport(memC%, inCh%, start, end, eCh%); Waveform->WaveMark
```

`memC%` The channel number of a memory channel created by `MemChan()`.

`inCh%` The channel number to import data from.

`start` The start time to collect data from.

`end` The end time to collect data up to (and including).

The following arguments are used when extracting events from waveform data. The level crossing modes use linear interpolation between points to find the exact time of the waveform crossing. The peak and trough modes fit a parabola to the three points around the peak or trough to estimate the time more accurately. Note that when extracting events to a WaveMark channel, the time saved is the time of the start of the waveform section, not the peak/trough or level crossing time.

`mode%` The mode of data extraction. The modes are:

- 0 Extract events based on the time of a peak in the waveform. If the destination is a marker, these events are coded as 2.
- 1 Extract events based on the time of a trough in the waveform. If the destination is a marker, these events are coded as 3.
- 2 Extract events based on the time of a level crossing with the waveform rising. If the destination is a marker, these events are coded as 4.
- 3 Extract events based on the time of a level crossing with the waveform falling. If the destination is a marker, these events are coded as 5.

- `time` The minimum time period between detected events. This can be used to filter noisy signals.
- `level` The level that the waveform must fall after a peak, rise after a trough, or cross, to detect an event.

The special mode to convert waveform to WaveMark data uses an extra channel to mark the waveform sections to be extracted. The waveform must have the same sampling rate as set for the WaveMark channel.

- `eCh%` A channel holding event times to mark the waveform sections to extract. The waveform starts the number of points set in the `pre%` parameter to `MemChan()` before each event. The time saved is the time of the first point in each waveform section.

Whichever version of the function you call, it returns the number of items added to the channel, or a negative error code.

The next example converts the DEMO data file waveform channel into WaveMark data by picking the peaks in the WaveMark data, then using these as the trigger points for the WaveMark data:

```
'MemCh4.s2s
var vh%, me%, mwm%, n%;           'view handle, memory chans
vh% := FileOpen("Demo.smr", 0, 0); 'Open demo file invisibly
if (vh% <= 0) then Message("No DEMO file!"); halt endif;
ChanHide(-1);Window(0,0,100,100); 'Hide all channels
WindowVisible(1);Draw(0, MaxTime()); 'show the lot

me% := MemChan(2);                 'create an event channel
if me% < 0 then Message("No event chan"); halt endif;
n% := MemImport(me%, 1, 0, Maxtime(), 0, .1, 1.0);
Message("Found %d peaks", n%);
ChanShow(me%); Draw();           'display the event markers

mwm% := MemChan(6,32,BinSize(1),16); 'copy rate from waveform
if mwm% < 0 then Message("No WaveMark"); halt endif;
ChanScale(mwm%, ChanScale(1));    'Copy scale...
ChanOffset(mwm%, ChanOffset(1));  '...and offset...
ChanUnits$(mwm%, ChanUnits$(1));  '...and units
ChanTitle$(mwm%, "Peaks");        'Set our own title
ChanComment$(mwm%, "Imported from channel 1");
n% := MemImport(mwm%, 1, 0, Maxtime(), me%); 'extract wavemarks
Message("Imported %d WaveMarks", n%);
ChanShow(mwm%);Draw();           'show what we got
```

MemSetItem() and MemGetItem()

`MemSetItem` edits or adds an item in a channel created by `MemChan()`. An item to edit is identified by its ordinal position in the channel (any mask set for markers is ignored). A new item is identified by a time alone. The ordinal position of the new or edited item is determined by the item time. You may not have two items at the same time, so if a new or edited item has the identical time to an existing item, the existing item is replaced.

`MemGetItem()` is the reverse of `MemSetItem()` and collects data for an item identified by the ordinal position of the item in the channel. If you want to identify data items by time, not ordinal position, you should use `ChanData()`. Also, `ChanData()` uses any marker filter that is set whereas `MemGetItem()` does not.

You can also use `MemGetItem()` with an index of 0, or no index, to get the number of data items in the channel.

```
Func MemSetItem(memC%, index%, time{, code%[] {, data$|data[]});
Func MemSetItem(memC%, index%, time, wave|wave%|wave[]|wave%[]);
```

- memC%** The channel number of a channel created by MemChan().
- index%** The index into the channel of the item to change. The first item is number 1. If you specify index 0, the function adds a new item to the buffer at a position determined by the time (which must be supplied).
- time** The time of the item, or -1 if the original time is to be preserved. If **index%** is 0, you must supply a time.
- For a waveform channel, it sets the time of the first waveform point. If there are already data points in the channel, the time is adjusted by up to half the sampling interval to be compatible with the sampling interval of the channel and the time of the existing data.
- code%** This is an integer array of at least 4 elements that hold the marker codes for the channel. If the channel does not require marker codes, this argument is ignored. If this parameter is omitted for a channel with markers, the codes are set to 0.
- data** This must be a string variable for TextMark data or a real array for RealMark or WaveMark data. It holds the data associated with the item. If the type of data is incorrect for the channel it is ignored. The number of points or characters set is the smaller of the size of the array or string and the size expected by the item.
- With RealMark and WaveMark data, if the array is too short, the extra values are unchanged when editing and have the value 0 when adding new data values.
- For WaveMark and waveform real data, the data values are limited to the range $-5*\text{scale}+\text{offs}$ to $4.99985*\text{scale}+\text{offs}$. Values outside this range are set to the nearer limit.
- wave** For a waveform memory channel you can set an individual value, or a list of values in an array. With a real variable or array, the values are limited as described in the previous paragraph. With integer values, the lower 16-bits of the 32-bit integer are copied to the memory channel (values greater than 32767 or less than -32768 will overflow).

The function returns the index at which the data was stored. If an index is given that is outside the range of items present, the function returns -1.

```
Func MemGetItem(memC% {, index% {, code%[] {, &data$|data[]}}});
Func MemGetItem(memC%, index%, &wave|&wave%|wave[]|wave%[] {, &n%});
```

- index%** The ordinal index into the channel to the item required. The first item is numbered 1. If you omit the index, or specify index 0, the function returns the number of items in the channel.

The remaining fields are only allowed if the index is non-zero.

- code%** This is an integer array of at least 4 elements that is returned holding the channel marker codes. If there are no markers in the channel, the codes are all set to 0.
- data** This must be a string variable for TextMark data or a real array for RealMark or WaveMark data. It is returned holding the data from the item. If the data type is incorrect for the channel, it is not changed. For an array, the number of points returned is the smaller of the size of the array and the number of values in the item.
- wave** This argument collects waveforms from waveform channels. If a real variable or array is passed, the waveform data is in user units. If an integer variable or

array is passed, the data is a copy of the 16-bit integer data used by Spike2 to store waveforms. The maximum number of elements copied is the array size.

When an array is used, only contiguous data is returned. A gap in the data (when the interval between two points is greater than the sample interval for the channel) terminates the data transfer.

`n%` If the previous argument is an array this optional argument returns the number of data items copied into the array.

For `index%` of 0 or omitted, the function returns the number of items in the channel. If `index%` is outside the range of items present, the function returns -1. Otherwise it returns the item time.

We have already seen how to use the functions to manipulate event data, so the next example shows how you might massage some waveform data. The `CopyWave%` function is the same as in `MemCh3.s2s`, so we have not shown it again:

```
'MemCh5.s2s
const np% := 400;
var vh%, mw%, n%, work[np%], mean, i%; 'view handle, memory chans
vh% := FileOpen("Demo.smr", 0, 0);    'Open demo file invisibly
if (vh% <= 0) then Message("No DEMO file!"); halt endif;
ChanHide(-1); Window(0,0,100,100);    'Hide all channels
WindowVisible(1); Draw(0, MaxTime()); 'show the lot

mw% := CopyWave%(1);                  'Copy it (see memCh3.s2s)
ChanShow(mw%); Draw();                'and show it

n% := MemGetItem(mw%) - np%;          'get the number of points
Seconds(0);                            'zero a timer
While Seconds() < 30 do
  i% := n% * Rand() + 1;              'index to mangle from
  MemGetItem(mw%, i%, work[]);        'get 10 points
  ArrSum(work[], mean);               'get mean of the data
  ArrSub(work[], mean);               'remove DC offset
  MemSetItem(mw%, i%, -1, work[]);    'replace mangled data
  Draw();                             'display the result
WEnd;
```

MemDeleteItem() and MemDeleteTime()

These functions delete one or more items from a channel created by `MemChan()`. To delete the entire channel use `ChanDelete()`. `MemDeleteItem` deletes based on an index range whereas `MemDeleteTime` deletes items based on a time range.

```
Func MemDeleteItem(memC% {, item% {, num%}});
Func MemDeleteTime(memC%, mode%, t1 {, t2});
```

`memC%` The channel number of a memory channel created by `MemChan()`.

`item%` The ordinal index into the channel to the item to delete. The first item is numbered 1. If you specify an index of -1 or omit this argument, all items are deleted.

`num%` The number of items to delete from `index%`. Default value is 1.

`mode%` This sets how many items to delete and the meaning of the time range. The modes are:

- 0 Delete a single item. The nearest item to `t1` in the time range `t1-t2` to `t1+t2` is deleted. If `t2` is omitted, it is taken as 0.
- 1 Delete all items in the time range `t1-t2` to `t1+t2`. If `t2` is omitted, it is taken as 0.

- 2 Delete the first item in the time range t_1 to t_2 . If t_2 is omitted it is taken as t_1 .
- 3 Delete all the items in the time range t_1 to t_2 . If t_2 is omitted, it is taken as t_1 .
- +4 If you add 4 to the mode and the channel has a marker filter set, only items that meet the filter specification are deleted.

t_1 , t_2 Two times, in seconds, that set the time range for items to delete.

Both functions return the number of items deleted, or a negative error code.

Here is an example that deletes items from a waveform memory channel. As in the previous example, `CopyWave%` is omitted from this listing:

```
'MemCh6.s2s
const np% := 20;
var vh%, mw%, i%;
vh% := FileOpen("Demo.smr", 0, 0);      'view handle and mem chans
if (vh% <= 0) then Message("No DEMO file!"); halt endif;
ChanHide(-1); Window(0,0,100,100);     'Open demo file invisibly
WindowVisible(1); Draw(0, Maxtime());  'Hide all channels
                                       'show the lot

mw% := CopyWave%(1);                    'Copy the waveform
ChanShow(mw%); Draw();                  'and show it

While MemGetItem(mw%)>0 do
  i% := MemGetItem(mw%) * Rand()+1;     'index to delete from
  MemDeleteItem(mw%, i%, np%);         'delete
  Draw();                               'display the result
WEnd;

For i%:=100 to 30 step -1 do WindowSize(i%, i%) next;
FileClose(0,-1);                        'Close the lot!
```

MemSave()

This function writes the contents of a channel created by `MemChan()` to the data file associated with the current window, making the data permanent. The memory channel is not changed; use `ChanDelete()` to remove it. Memory channels are not saved when the associated data file closes. All the properties of the memory channel are saved, including the title, channel comment, and if the channel has waveform data, the scale, offset and units.

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

`memC%` A channel created by the `MemChan()` function.

`dest%` The destination channel in the file. This must be in the range 1 to 32 as "real" channels on disk have this channel range. It is possible that in the future we will extend the SON file format to allow a larger (or user-defined) number of channels.

`type%` The type of data to save the data as. The type selected must be compatible with the data in the memory channel. Codes are:

0 Same type (default)	3 Event (Event+)	6 WaveMark
1 Waveform	4 Level (Event+-)	7 RealMark
2 Event (Event-)	5 Marker	8 TextMark

The special code -1 means append to an existing channel. The new data must occur after the last item in the `dest%` channel and the `dest%` channel must be a compatible type.

query% If this is not present or zero, and the dest% channel is already in use, the user is queried about overwriting it. If this is non-zero, no query is made.

The function returns the number of items written, or a negative error code.

The next example applies a crude differentiator to the waveform data in the DEMO file, and saves it in a new channel, channel 6:

```
'MemCh7.s2s
var vh%, mw%, i%;           'View handle, memory chans
vh% := FileOpen("Demo.smr", 0, 0); 'Open demo file invisibly
if (vh% <= 0) then Message("No DEMO file!"); halt endif;
mw% := Filter%(1, 0, MaxTime());
if mw% <= 0 then Message("Sorry... failed"); halt endif;
MemSave(mw%, 6);           'Write the channel
FileClose();               'kill the file
FileOpen("Demo.smr",0,0);  'open it again, and show
Window(0,0,100,100);      'Big screen production!
ChanHide(-1);ChanShow(1,6); 'show it, and new version
WindowVisible(1);Draw(0,maxTime()); 'display the lot
halt;

' This func filters wavform data on channel wChan% between sTime
' and eTime and writes the result to a memory channel. It returns
' the memory channel number or a negative error code.
func filter%(wChan%, sTime, eTime) 'returns memory channel
var flt[5];                       'filter coefficients
flt[0]:=-0.2; flt[1]:=-0.1; flt[2]:=0; flt[3]:=0.1; flt[4]:=0.2;
if ChanKind(wChan%)<>1 then return -1 endif;
if (sTime >= eTime) OR (sTime<0) then return -2 endif;
var mc%;
mc% := MemChan(1,0,BinSize(wChan%)); 'Create channel
if mc%<=0 then return mc% endif;
ChanScale(mc%, ChanScale(wChan%)); 'Copy scale...
ChanOffset(mc%, ChanOffset(wChan%)); '...and offset...
ChanUnits$(mc%, ChanUnits$(wChan%)); '...and units
ChanTitle$(mc%, "Filtered");
ChanComment$(mc%,"Filtered from channel "+Str$(wChan%));
var work%[8000]; ' buffer to process data with
var read%;      ' the number of bins read
var op% := 0, toff;' where to start output and time offset
repeat
  toff := Binsize(wChan%)*op%; 'time offset for overlap
  read% := ChanData(wChan%, work%[], sTime-toff, eTime, sTime);
  if (read%>4) then 'if we got some data
    ArrFilt(work%[:read%],flt[]); 'filter it, then copy it
    MemSetItem(mc%, 0, sTime+toff, work%[op%:read%-op%-2]);
    sTime := sTime + BinSize(wChan%)*(read%-2); 'next start
  endif;
  op%:=2; ' after first buffer, we overlap by 2.
until read%<=4;
ChanShow(mc%);
return mc%; 'Return the new channel number
end;
```

This command creates a memory command, and then adds data to it in blocks of up to 8000 points. Because the filter used (of length 5) produces imperfect data for the first and last 2 points, each block processed is overlapped by 2 points to allow for this. See the ArrFilt() command documentation for more details about the filtering operation.

In this case, we assume that there is enough memory and virtual memory to keep the entire result as a memory channel. If this were not the case, it would be possible to write the data incrementally using MemSave with a type code of -1. This is usually only needed where there are megabytes of data to process.

Text and binary file manipulation

Sooner or later you will need to move data between Spike2 and some other format for which there is no automated converter. We have covered writing data to a text file earlier in this book, so we will not go over that again. In this section we will look at importing data from a text file and from a binary file, and exporting data to a binary file.

Important To import from and export to “foreign” formats you must have detailed knowledge of the format of the file, particularly if the file contains binary data.

The steps needed to import data (from either binary or text files) are:

1. Get the length, the number of channels and waveform sample rates of the input file.
2. Create a new file using `FileNew()` in mode 7 with a suitable time and ADC resolution such that you can set any waveform sampling rate required, and so that the time base can run for long enough to include all the data (usually not a problem).
3. Read the data file (either from text or a binary file) and create memory channels in which to store the data using `MemSetItem()`.
4. Write the memory channels to permanent channels in the file using `MemSave()`. To minimise memory requirements it is a good idea to read and write one channel at a time, then delete the memory channel, but this is not always a practical proposition.
5. Close the data file, and the job is done.

Text file import

The first example imports from a text file. To keep the script short enough to serve as an example, we will use a data file with 4 columns of data, corresponding to 4 channels of waveform data. The example is written so you can easily change the number of channels.

We have been told that all the channels are sampled simultaneously and at the same rate (1 kHz), and there are 2000 lines of data. By inspection of the data file, we see that the data on each channel is listed in microvolts and the data is in the range -100 to +100 microvolts. We set the time of the first point on each channel to 0.0 seconds. If the data was sampled sequentially we would have to offset the start time of each channel.

We can see that the file will be 2 seconds long. Each channel is sampled every 1000 microseconds. In this case, a very wide range of `FileNew()` command arguments would produce a satisfactory file. Here are the two extreme cases:

```
fh:=FileNew(7,0,1,1,2); '1us per tick, 1MHz ADC, 2 seconds long
fh:=FileNew(7,0,1000,1,2); '1ms per tick, 1kHz ADC rate, 2 seconds
```

To decide which values to set, we need to look at the documentation for `FileNew()`:

```
Func FileNew(type%, mode%, upt%, tpa%, maxT);
```

- `type%` We are only interested in type 7 which creates an empty Spike2 data file.
- `mode%` In this case, 1 = create a visible file, 0=create an invisible file.
- `upt%` Microseconds per unit time for the new file. This allows values in the range 1 to 32767 (for compatibility with Spike2 sampling we recommend a range of 2 to 1000). This sets the time resolution and the maximum duration of the file. There are a maximum of $2^{31}-1$ time units in a file, so a 1 microsecond resolution limits a file to 30 minutes in length.
- `tpa%` The time units per ADC conversion for the new file. You can set values in the range 1 to 32767 (for compatibility with Spike2 sampling use 2 to 32767). The available sampling intervals for waveform data in the new file are $n\% * upt\% * tpa\%$ microseconds.
- `maxT` This sets the maximum initial time base that you can display for the file. You should set this to the expected file size.

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

The only reason to choose values that are compatible with Spike2 sampling is so that you can read a configuration from the file and use it for sampling data, so you can ignore this restriction if you wish.

The `upt%` parameter sets the time resolution of your file and sets a limit on the file length. In this case, there is no problem with length, but we must choose a value that will allow us to produce a 1000 microsecond sampling interval for the ADC. This means we must choose a value that is a factor of 1000. To fit in with Spike2 sampling we choose 2.

The `tpa%` parameter sets the fastest rate for any waveform channel. If we set `tpa%` to 2, with `upt%` set to 2, the fastest waveform channel would be 250 kHz (4 microseconds per point). We need to achieve 1000 microseconds, in this case 500 clock ticks, so the value we choose must be a factor of 500. We'll choose 50 (any factor would do).

Next, we must create memory channels to display the data. We need to set up suitable scales and offsets so that it can be stored efficiently. Waveform data is stored on disk as 16-bit integers (i.e. numbers in the range -32768 to 32767) and is then scaled to user units by the equation: $\text{User units} = \text{integer} * \text{scale} / 6553.6 + \text{offset}$. If we assume that the `offset` is 0.0, then a `scale` of 1.0 give a maximum of 5 (actually, slightly less). To get a range of -100 to +100 we need a scale of 20.

Finally, we must read the data. The first few lines are:

```
-15.564,2.28882,-10.9863,-14.801
-15.8691,2.59399,-10.6812,-12.97
-15.4114,5.0354,-5.64575,-10.5286
```

This can most conveniently be done by reading the data into an array of length 4, one line at a time. A possible script to do the job is:

```
'import1.s2s
const nCh% := 4, adcInt := 0.001; 'number of chans, adc interval
const upt% := 2, tpa% := 50; 'time and waveform resolution
const maxN% := 2000; 'maximum points to read
var th%,fh%; 'text and file handles
th% := FileOpen("data.txt", 8, 0); 'Open external file for reading
if th% < 0 then Message("data.txt not found"); halt endif;

var data[nCh%][maxN%],i%:=0, err%; 'for data and line counter
repeat
  err% := Read(data[][i%]); 'get next line
  if (err% > 0) then i% := i% + 1 endif;
until (err% <= 0) or (i%>=maxN%-1); 'allow for a shorter file
FileClose(); 'finished with the text file

fh% := FileNew(7, 1, upt%, tpa%, i%*adcInt); 'create the new view
if fh%<=0 then Message("No new file..."); halt endif;

var ch%, mc%; 'channel loop, memory channel
mc% := MemChan(1, 0, adcInt); 'create a memory channel
if mc% <= 0 then Message("Create failed"); halt endif;
ChanScale(mc%, 20.0); 'set channel characteristics
ChanOffset(mc%,0.0);
ChanUnits$(mc%, "uV"); ChanComment$(mc%, "Imported data");

for ch% := 0 to nCh%-1 do 'cycle round channels
  ChanTitle$(mc%, "Import"+Str$(ch%));
  MemSetItem(mc%, 0, 0, data[ch%][:i%+1]); 'copy data
  MemSave(mc%, ch%+1); 'copy to disk
  MemDeleteItem(mc%); 'delete all items
next;

ChanDelete(mc%);ChanShow(-1); 'kill memory channel, show rest
Window(0,0,100,100);Draw(); 'so we can see them
```

Notice that the varying parameters have been declared as `const` at the top of the script. This makes them easy to see and change. There are three improvements that could be made to this script easily:

1. Allow the user to select the source file. To do this change `"data.txt"` to `"`.
2. Allow the script to skip lines at the start as many such data files will have header lines that contain unwanted data or might contain scale factors and channel titles.
3. Make the channels self-scaling. To do this we should run the `Min` and `Max` functions on the arrays to find the extent of the data, then adjust the scaling accordingly. You would need to choose between scaling the channels individually or giving them all the same scaling.

Sometimes you will get data from other programs in ADC units. That is, the data is a list of integers, most often from 12-bit ADCs, but sometimes from 14 or 16-bit systems. This data is usually in the range -32768 to 32767.

To process this data, the approach is just the same as in the `import1.s2s` script, but the trick is to replace `data[nCh%][maxN%]` with `data%[nCh%][maxN%]`. The effect of this is to copy the data unscaled from the text file to the Spike2 file. The y axis labels you get then depend on the scale and offsets set for the channels. You could use a real array as in the previous example, but this method guarantees that exactly the same data ends up in the file as is in the text file; there are no rounding effects due to converting between real units and the 16-bit integers used by Spike2 to store waveform data.

Binary file import

In principle, binary file import is exactly equivalent to text file import, but usually a lot faster! However most people find it more confusing as they can read the contents of a text file but a binary file is much more difficult to fathom. To get around this problem, we have written a simple binary file dumping script called `BinShow1.s2s`. This displays a binary file as 32-bit, 16-bit or 8-bit integers, or 64-bit or 32-bit reals.

This can be a useful tool when you need to import a binary file and you need to check the format of the data. The program has a toolbar with the following buttons:

`Clear File String Real Double Byte Short Long Offset Bytes Quit`

`Clear` This cleans out the display in the log window (used for the dump).

`File` Use this to select a new data file to poke around in. The offset is reset to 0.

`String` This attempts to dump the file from the current offset for the number of bytes set. Non-printing characters are replaced by a "." in the output.

`Real` Dump the data from the current offset as 32-bit IEEE floating point numbers.

`Double` The same as for `Real`, but dump the data as 64-bit IEEE floating point numbers.

`Byte` The data is dumped as 8-bit bytes in hexadecimal.

`Short` The data is dumped as 16-bit words in hexadecimal.

`Long` The data is dumped as 32-bit words in hexadecimal.

`Offset` You can set the offset in the file to dump from. You can set the offset as a decimal number, or as a hexadecimal number. To enter a number in hexadecimal start the number with `0x`.

`Bytes` This sets the number of bytes to dump. You are forced to accept at least 16 and are limited to 1024 bytes at a time.

`Quit` This does what it says.

Now an example of importing a file.

```
'BinImp.s2s
var fh%;                               'file handle for the binary file
var fSz%;                               'size of the binary file
var np%;                               'number of points in the file

fh% := FileOpen("bindata.dat",9,0);
if fh%<0 then Message("No such file");halt endif;
fSz% := BSeek(0,2);                     'Seek to end of file to get size
if (fSz% < 4100) then Message("File too small"); halt endif;
np% := (fSz%-4000)/2;                   'the header is known to be 4000 bytes
ReadFile(np%, 8);                       'get the data, say there are 8 channels
halt;

func ReadFile(np%, nCh%)
var dat%[nCh%][np%/nCh%]; 'generate huge array
var got%;                          'will be points we read
BSeek(4000,0);                      'seek to start of the data
got% := BReadSize(2,dat%[0][0]);'read the file into memory
if (got% <> np%) then Message("Error reading"); return -1; endif;
MakeNew(dat%[0][0], 9542); 'convert into a new file
end;

Func MakeNew(dat%[0][0], us)
var fNew%, ppc%, nCh%;               'new handle, points per chan, chans
ppc% := Len(dat%[0][0]);             'points per channel
nCh% := Len(dat%[0][0]);             'number of channels
fNew% := FileNew(7, 1, 2, 1, ppc%*us/1000000.0);
var iCh%,thisCh%;
for iCh%:=0 to nCh%-1 do 'loop round channels
  thisCh% := MemChan(1,0,us/1000000);
  if thisCh%<0 then Message("Create channel failed"); halt endif;
  MemSetItem(thisCh%, 0, 0, dat%[iCh%][0]); 'save as memory channel
  MemSave(thisCh%, iCh%+1,1); 'copy to real channel
  ChanDelete(thisCh%); 'delete memory channel
next;
Optimise(-1); ChanShow(-1); 'so we can see the result
return fNew%;
end;
```

The data file that this routine imports came from a prospective Spike2 customer who wanted to see some old data that was sampled with a different system. All he could tell us about the data was that there was a 4000 byte header, followed by 8 channels of data and that each channel was sampled at around 105 Hz.

A little investigation revealed that the header was filled with 0's except for the very start, which had a short text string. By dumping the data from offset 4000 in the file, you can see that the data is 16-bit integers.

The script starts by opening the file (in external binary mode) and using the `BSeek()` command to find the size of the file. This function moves and reports the current position in a file opened by `FileOpen()` with a `type%` code of 9. The next binary read or write operation to the file starts from the position returned by this function.

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

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

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

The function returns the new file position relative to the start of the file or a negative error code. By moving to position 0 relative to the end of the file, it returns the file size. As we know that the file holds 16-bit integer data, we can calculate how many such data points are in the file by subtracting 4000 (the header size) from the file size and dividing the result by 2. We know that there are 8 channels; so dividing the number of points by 8 gives the number of points per channel. Armed with this information, we call the routine to read the file, passing in the number of points per channel and the number of channels.

The `ReadFile` routine uses the information passed to create an integer array that corresponds to 8 channels of interleaved data. We could have declared an array of one-dimension that was 8 times larger, but then we would need to extract the data before we could write it to a data channel. Note the use of `Len()` to get the size of the array dimensions. We could just as easily have passed them into the function, but this saves using extra arguments.

We now use `BSeek` again, this time relative to the start of the file, to move to the start of the data. Now we can use `BReadSize(2, dat%[][])` to fill the array with data. There are two routines for reading binary data into variables:

```
BRead(variable list)
BReadSize(size%, variable list)
```

The `Bread()` routine reads a block of memory equal in size to the memory used by each variable in the list and copies this to the variable. `BreadSize()` moves the number of bytes set by `size%` (which must be 1, 2 or 4 for an integer variable and 4 or 8 for a real variable, and can be any non-zero size for a string) into memory, then converts the bytes read appropriately into a Spike2 integer (4 bytes long), a Spike2 real number (8 bytes long) or into a string. In our case, the data on disk is 2-byte integers, so we use a size of 2. For every array element, Spike2 reads two bytes and sign extends them to 4 bytes.

Having read our data, we then pass the array to a routine that creates a suitable data file, and for each channel it creates a memory channel, fills it with data, copies it to a disk channel, then deletes the memory channel.

The result is a new data file made from the binary data. This file holds some 3,250 points per channel and there are 8 channels. You will notice that this imports data much faster than the previous example, which used text and imported 2000 points per channel on 4 channels.

Binary file export

Binary file output is more or less the reverse of binary file input, but it can be rather messy, as you cannot simply skip over headers; you must write them too. As an example, the script `picout.s2s` contains all the routines you need to export data as a Lotus 1-2-3 PIC format. In this case, you need a header of 17 bytes that identify the file, followed by packets of data that are the drawing instructions. The example shows how to output a square with some text in it.

A more usual requirement is to output data from a Spike2 data file into another format. For example, on the Macintosh, users quite often want to export a Spike2 waveform channel to the Igor program. Igor can read a wide variety of binary files, in much the same way that Spike2 can. The most efficient way (in the sense of losing no information and producing the smallest file) is to write the output as 16-bit integers, one 16-bit value for each data point in the channel. The most efficient way to dump times is as 32-bit integer data. This next example lets you open a data file, select a channel, and dump it as a binary file as 16-bit integers for waveforms and as 32-bit integers for times or as 32-bit reals or as 64-bit reals. If a channel is not a waveform, we treat it as an event channel.

```

'BinDump.s2s
'Declare variables:
var ok%,chan%,fh%,bh%,close%:=0, fmt%:=0, fmt$[3];
fmt$[0] := "Integer"; fmt$[1]:="32-bit IEEE real";
fmt$[2] := "64-bit IEEE real";
if ViewKind()<>0 then
  fh% := FileOpen("", 0, 0, "File to dump as binary");
  if fh%<=0 then Message("No file to dump"); halt endif;
  close% := 1; 'say we should close file
endif;
fh% := View(); 'get the starting view handle
DlgCreate("Binary file dump"); 'Start new dialog
DlgChan(1,"Choose the channel to dump",16511);
DlgList(2,"Format for the output",fmt$[]);
ok% := DlgShow(chan%, fmt%); 'ok is 0 if user cancels
if ok% <> 0 then 'dump if user selects a channel
  bh% := FileOpen("", 9, 1, "File to dump channel to");
  if bh% > 0 then BinDump(bh%, fh%, chan%, fmt%) endif;
endif;
View(bh%);FileClose(); 'close the binary file
if close% then 'if we opened the file..
  View(fh%); '...then move to it...
  FileClose(); '...and close it again
endif;

Proc BinDump(bh%, fh%, ch%, fmt%) 'binary file, data file, channel
docase
  case fmt% = 0 then IntDump(bh%, fh%, ch%); 'Integer format
  case fmt% = 1 then RealDump(bh%, fh%, ch%, 4); '4 byte real
  case fmt% = 2 then RealDump(bh%, fh%, ch%, 8); '8 byte real
endcase;
end;

Proc RealDump(bh%, fh%, ch%, bytes%)
const BSZ% := 8000; 'buffer size
var work[BSZ%]; 'work space
var t := 0, n%; 'start time, items read
View(fh%); 'in data file view
if ChanKind(ch%)=1 then 'waveform channel
  repeat
    n% := ChanData(ch%, work[], t, MaxTime(), t);
    if n% > 0 then 'if we got data, then
      View(bh%).BWriteSize(bytes%, work[:n%]); 'Output it
      t := t + n% * BinSize(ch%); 'time of next point
    endif;
  until n% <= 0; 'until no points left
else
  repeat
    n% := ChanData(ch%, work[], t, Maxtime());
    if n% > 0 then 'if we got data then...
      View(bh%).BWriteSize(bytes%, work[:n%]); 'Write it
      t := work[n%-1]+BinSize(ch%); 'next search start
    endif;
  until n% <= BSZ%; 'until buffer not full
endif;
end;

proc IntDump(bh%, fh%, ch%)
const BSZ% := 8000;
var work%[BSZ%]; 'work space
var t := 0, n%; 'start time, items read
View(fh%); 'in data file view
if ChanKind(ch%)=1 then 'waveform channel
  repeat
    n% := ChanData(ch%, work%[], t, Maxtime(), t);

```

```

        if n% > 0 then                'if we got data
            View(bh%).BWriteSize(2, work%[:n%]); 'Output it
            t := t + n% * BinSize(ch%); 'time of next point
        endif;
    until n% <= 0;
else
    repeat
        n% := ChanData(ch%, work%[], t, Maxtime());
        if n% > 0 then                'if we got any data
            View(bh%).BWrite(work%[:n%]); 'Write it
            t := BinToX(work%[n%-1]+1); 'next search start
        endif;
    until n% <= BSZ%;
endif;
end;

```

Most of this script should look familiar, however there are a few points that merit further explanation.

1. Both the real and integer dump routines use `ChanData` to collect data from the channel and then write it. You may notice that there is a difference between the loops for waveform and event data. Waveform data uses `repeat ... until n% <= 0;` but the event channels use `repeat ... until n% <= BSZ%;` and you may wonder why.

The reason is that for a waveform channel, `ChanData` returns contiguous data. If there is a gap in the data `ChanData` returns data up to the gap and you must call it again to get the next data block. With event data, there is no such thing as a gap, or rather, there is a gap between all events anyway. `ChanData` just fills up the array with event times. If it fails to fill the array this means that there are no more events.

2. The method for getting the time for the next block of data for waveform data is to take the time of the first data point on the channel, and add the number of points times the interval between each point to give the predicted time of the next event on the channel after the block. If there is a gap at this point, `ChanData` will find you the next waveform data either at or after the start time.
3. The method for getting the next start time of event data differs depending upon the format of the data. When you collect real numbers, the array holds event times and we set the start time for the next block to be the time of the last event plus the timing resolution (which is given by the `BinSize()` function when used on an event channel).

When you collect an integer array, it is filled with the times of event in the basic Spike2 timing unit for the file. So to find the read time, we add 1 tick, then use `BinToX()` which converts from ticks to seconds when used in a time view.

4. Beware... if you use 32-bit real numbers to write event times, you have limited precision. The events are coded internally in Spike2 as 32-bit integers. A 32-bit real has some 23 bits of magnitude information plus a power of two scale and a sign bit. This means that once you get past around 8 million Spike2 clock ticks, binary output of times using 32-bit real data will start to lose resolution compared to the original. It is much better to use 64-bit reals or 32-bit integers that do not suffer from this problem.

Serial line use from Spike2

You can use the serial lines attached to your computer to read and write data from Spike2. This can be used to control external equipment, or to collect additional data during data capture. The serial line ports are controlled through the `SerialOpen()`, `SerialRead()`, `SerialWrite()`, `SerialCount()` and `SerialClose()` commands.

The information given here duplicates and expands on the information in the Spike2 script language manual and in the on-line help.

Before we start looking at these commands in detail, be warned that more human time has been spent trying to connect one piece of serial equipment to another than was consumed in the invention of the wheel. The problem is the vast diversity of serial line plugs and sockets, the number of variants of control signal, and the fact that the parameters that govern the transfer speed and style must match at both ends.

This description deals only with the signals on a serial line connector that are relevant to Spike2. Do not make connections to other pins than described below. We are concerned with three types of connector on the back of your computer. PCs have either 25-way or 9-way Cannon sockets for serial line use. The Macintosh uses an 8-pin mini-DIN connector. The pins are numbered, but you need good eyesight to see the numbers. Output and Input are with respect to the computer, not the external device.

A modern PC has typically more than 1 serial port so you must first identify the port numbers. You must also be prepared for the fact that some ports are not available as they have been booked by other software, or are used for other things. For example, on our PowerPC Macintosh, port 1 is not available, port 2 (the printer port) can be used.

Name	9-pin PC	25-pin PC	mini-DIN Macintosh	Function
GND	5	7	8,4	The system ground
TX	3	2	3	Output data to the external device
RX	2	3	5	Input data from the external device
DTR	4	20	1	Output to external device to say OK to transmit
CTS	8	5	2	From external device saying Clear (OK) To Send
RTS	7	4		Request to Send (not used by Spike2)
DSR	6	6		Data Set Ready, can be used to stop output
CD	1	8		Carrier Detect (not used by Spike2) for modems

Handshake lines

You only need to consider the state of the handshake lines if you use the hardware handshake mode of operation (see `SerialOpen()`). If you use no handshake, or you use XON/XOFF protocols then you can ignore all lines except GND, TX and RX.

DTR output goes high when you open the port, and goes low when you close it. It also goes low in hardware handshake mode when the input buffer is about to overflow (this works for the Macintosh but is not yet implemented under Windows; it will be in the future).

CTS is used in hardware handshake mode to indicate that output may take place. Under Windows, DSR must also be held high to allow output in handshaking mode. You can either connect CTS and DSR together, or connect DSR to RTS to hold it high permanently.

If you use the hardware handshake, both DSR and CTS must be high for data to be transmitted. If you don't use the handshake, you can ignore these signals.

Opening the serial line

We considered making the serial ports look like external text files, but this turned out to be more trouble than it was worth, so they are view independent. The `SerialOpen()` command chooses which serial line to use and sets the characteristics of the data transfer and the handshake method (if any).

```
Func SerialOpen(port%, baud%, bits%, par%, stop%, hsk%)});
```

port% The serial port to use in the range 1 to 9. The actual number of ports depends on the computer. Two ports (1 and 2) are common on both PC and Macintosh systems, but users may have installed more.

baud% This sets the Baud rate (number of bits per second) of your serial line. The maximum character transfer rate is of order one tenth this figure. Standard Baud rates are 300, 600, 1200, 2400, 4800, 9600, 19200, 38400 and 57600. If you set any other rates they are rounded down to a rate in the list. The rates available to you on your system may be a subset of this list. Rates up to 9600 Baud should work on any system. If you do not supply a Baud rate, 9600 is used.

bits% The number of data bits used to encode a character. You may set either 7 or 8. If you do not specify the number of data bits, 8 is set.

par% Set this to 0 for no parity check, 1 for odd parity or 2 for even parity. If you do not specify this argument, no parity is set.

stop% This sets the number of stop bits as 1 or 2. If omitted, 1 stop bit is set.

hsk% This sets the handshake mode, sometimes called "flow control". 0 or omitted sets no handshake, 1 sets a hardware handshake, 2 sets XON/XOFF protocol.

If a port could be opened the function returns, otherwise it returns a negative error code which usually means that the port does not exist, or is already in use.

The **baud%**, **bits%**, **par%** and **stop%** arguments should be set to match those of the external device you want to talk to. In general, setting the highest baud rate that both your computer and device can manage is good, but the faster the rate, the more likely buffer overflow and data loss becomes. The maximum number of characters that can be transferred per second is $\text{baud\%} / (1 + \text{bits\%} + \text{stop\%})$.

The handshake control is used to stop transmissions when there is a danger of data being lost due to input buffer overflows. Both the Windows and the Macintosh systems have input buffers that can store around 1000 characters. There is similar space in the output buffers. You need these buffers because serial transmission is so slow; without them programs would spend all their time waiting for the last character to be sent, or waiting for a new character to be received.

Writing data You output data to a serial port previously opened with `SerialOpen()`.

```
Func SerialWrite(port%, out$|out$[][, term$]);
```

out\$ A single string to write to the output or...

out\$[] ... an array of strings to write to the output.

term\$ If present, the contents of the string are written to the output port after the contents of **out\$** or after each string in **out\$[]**. We suggest that you don't use "\n" as a terminator as it has different values under Windows and the Macintosh. Use `Chr$(13)` for Carriage Return and `Chr$(10)` for Line Feed.

The function returns the number of strings written or a negative error code. If the output system becomes full, the function waits for one second before timing out. If a time-out occurs, the function returns the number of strings sent before the time-out.

If you are sending a lot of data so that there is a chance that the output buffer might overflow, you **MUST** check the value returned by the function. If you do not, you will find that output is missing. Spike2 will only send complete strings; if there is no room for all the string none of the string is transmitted. There is also a limit of 255 characters in a string (this is a Macintosh limit, but we also impose it on the Windows program). Longer strings are truncated.

Reading data There are two functions to use: `SerialRead()` and `SerialCount()`. If your script is time critical (you do not want it hanging up for even a second), use `SerialCount` to check that a call to `SerialRead` will return immediately:

```
Func SerialCount(port% {,term$});
```

`term$` An optional string holding the character(s) that terminate an input item.

If `term$` is absent or empty, this returns the number of characters that could be read. If `term$` is set, this returns the number of complete items that end with `term$` that could be read.

You collect data with:

```
Func SerialRead(port%, &in$|in${}[,term${, max%}]);
```

`in$` A single string or an array of strings to fill with characters. There is no point providing an array of strings unless you have set a terminator because without a terminator all input goes to the first string in the array.

`term$` If this is an empty string or omitted, all characters read are input to the string or to the first string in the array of strings and the number of characters read can be limited by `max%`. The function returns the number of characters read.

If this is not an empty string, the contents are used to separate data items in the input stream. Only complete items are returned and the terminator is not included in a returned string. If `in$` is a string, one item at most is returned. If `in${}` is an array, one item is returned per array element. The function returns the number of items read.

`max%` If present, it sets the maximum number of characters to read into each string. If a terminator is set, but not found after this many characters, the function breaks the input at this point as if a terminator had been found. There is a maximum limit set by the size of the buffers used by Spike2 to process data and by the size of the system buffers used outside Spike2. This is typically around 1024 characters.

The function returns the number of characters or items read or a negative error code. If there is nothing to read, the function waits for one second to allow characters to arrive before timing out and returning 0.

Closing the serial line If you don't close the serial line, it will not be available to other applications. Closing a port also releases system resources and buffer space that Spike2 has allocated for the serial line use. Spike2 automatically closes any serial ports that have been opened each time a script ends. The call is:

```
Func SerialClose(port%);
```

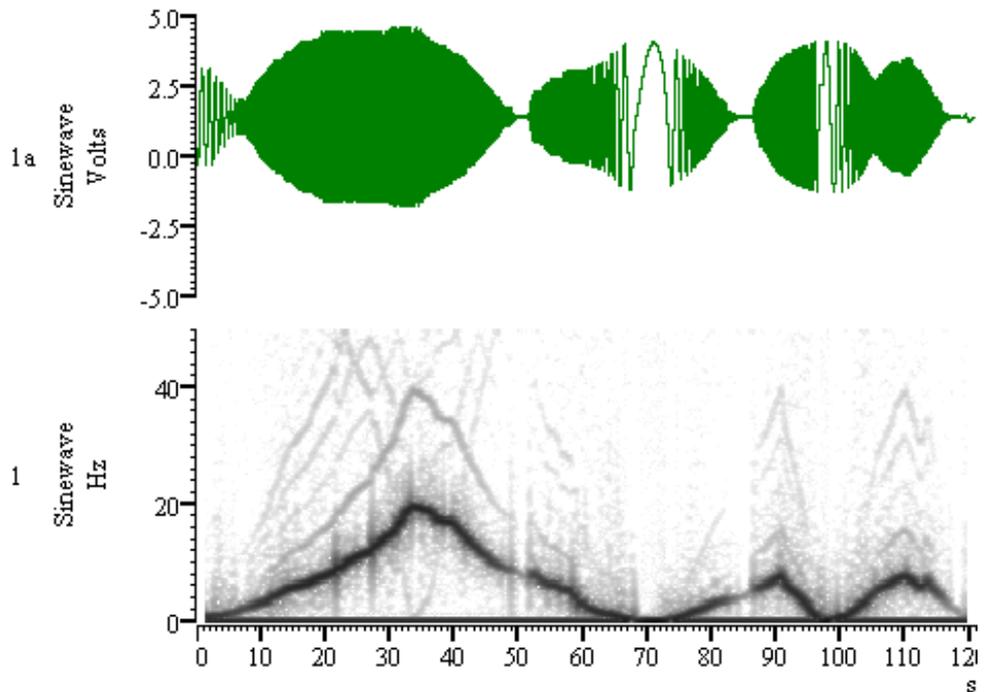
This returns a negative error code if the port had not been opened.

Sonogram display mode

Sonograms are available from Spike2 version 3 as a waveform display mode. If you are using version 2 you should skip this section.

What is a sonogram?

A sonogram is a method for drawing a waveform to display the frequency content. The x axis remains as time, but the y axis becomes frequency (Hz), and can display data from 0 Hz to half the sampling rate of the waveform channel. The display of the waveform is replaced by a density map representing the power of each frequency present at a particular time. The power is calculated using a Fast Fourier Transform (FFT).



The image above is a bitmap exported from Spike2 (sonograms are not exportable as metafiles). It shows the waveform data in the well-known `demo.smr` file drawn as a waveform, and as a sonogram. The sonogram settings are as displayed below in the Channel Draw mode dialog. The waveform data is a recording from a rather poor signal generator of a sine wave with an offset of around 1 Volt. Both the amplitude and frequency of the sine wave vary with time.

The heavy line in the sonogram shows the main component of the sine wave. The lighter lines show the harmonics of the original sine wave. The waveform was sampled at 100 Hz, so the maximum frequency in the result is 50 Hz. You can see that when the frequency components of the harmonics exceeded 50 Hz, they were “reflected” from the 50 Hz limit (and also from the 0 Hz limit). The original data was not band-limited, and this illustrates very well how frequency components that exceed half the sampling rate are “aliased” to lower frequencies.

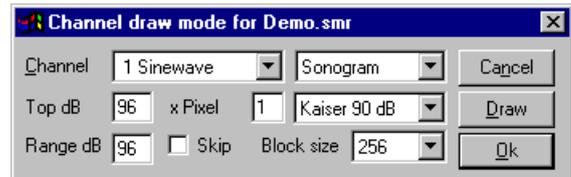
The heavy line at 0 Hz represents the DC offset in the waveform. The gaps at the start and end of the display (from 0 to 1.28 seconds and a similar gap at the end) are a function of the FFT size, see below.

Colour sonogram

From Spike2 version 6 onwards you can use a colour scale for the sonogram. This is set in the Edit menu Preferences option Display Tab.

Channel Draw mode dialog

The Channel Draw mode dialog is considerably extended for the Sonogram display mode. When you select the Sonogram mode, several new fields appear:



Top dB, Range dB

These two fields set the range of intensity that the grey scale will span. Signals more intense than the Top value will be given the highest intensity, Signals less intense than Top - Range, will be displayed with the lowest intensity.

Intensity is measured in dB (which stands for decibel). A dB is a measure of the ratio of two things. In this case it measures the relative power of the signal at a particular frequency relative to a sine wave at that frequency of amplitude 1 bit at the 1401 input. If the ratio of these two powers is R_p , then the value in dB is given by $10\text{Log}_{10}(R_p)$. Another way to look at this is as the ratio of the amplitudes of the two signals, in which case the formula is $20\text{Log}_{10}(R_a)$ where R_a is the ratio of the amplitudes. Given that waveform data is stored as 16-bit integers, the maximum possible top value is 96 dB.

Window type

This field (shown as Kaiser 90 dB in the picture) is very important, but rather difficult to explain (but we'll try anyway).

The Fast Fourier Transform is a mathematical trick to quickly calculate a set of sinusoids, equally spaced in frequency which, when added together, are equivalent to a set of data points equally spaced in time. The result is an approximation to the output of a set of notch filters, each centred on one of the equally spaced frequencies, provided that certain conditions are met. The main conditions are that the data contains no frequency components above half the sampling rate, and that the data repeats cyclically.

Now band limiting the data is usually not a problem, but real data does not repeat. If you just take FFT size data points from a data channel and then repeat them you will usually get a waveform with a big discontinuity at the repeat. If you take the FFT, you will find that the discontinuity causes considerable distortion in the result. The distortion does not move data peaks, but it spreads the power from a peak into what are called "side-lobes", which are a series of peaks of lower amplitude either side of the real peak.

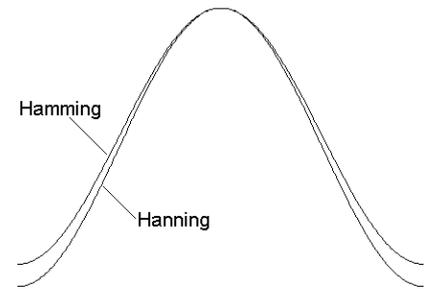
If you are looking for a small signal in the presence of a larger one, then side-lobes are a big problem. However, if you know there is only one sine wave present, then side-lobes may not matter.

The traditional solution to the problem is to multiply the data by a smooth function that is large in the middle, and small near the ends. The argument is that then the data is small at the ends, and so will "join up" smoothly, thus avoiding the problem. However, the act of multiplying the data by a window introduces its own distortion to the result.

You can find all sorts of windows discussed in the literature, each with its own advantages and disadvantages. The basic compromise in a window function is that windows shaped to have the smallest side-lobes spread the peak out the most, so by increasing the dB range of the result, you decrease the certainty of where any frequency peak actually is (or the ability to separate two peaks that are close together). Spike2 implements the following windows:

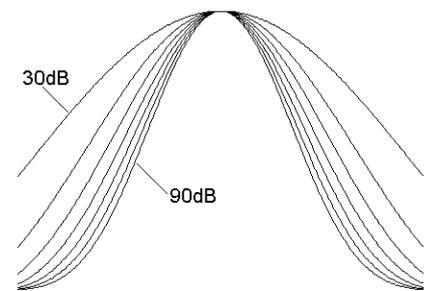
No Window Only use this if you know there is one sine wave present, or if there are more than this, they must all have similar amplitude. This gives the sharpest peaks in the power spectrum, but the worst side-lobes.

Hanning This is a good general purpose, reasonable compromise window. However, it does throw away a lot of the signal. It is sometimes called a “raised cosine” and is zero at the ends. If you are unsure about which window would be best for your application, try this one first.



Hamming This improves on the loss of signal of the Hanning window using a raised cosine. It preserves more of the original signal, but at the price of unpleasant side-lobes.

Kaiser These are a whole family of windows calculated to have a given maximum side-lobe amplitude relative to the peak. Of course, the smaller the side-lobe, the more signal is lost and the wider the peak. We provide a range of windows with side-lobes that are from 30 to 90 dB less than the peak.



FFT size This is a most important field. This determines how many data points are used for each transform. The more points you set, the better will be the frequency resolution in the result. However, the more points you set, the worse will be the time resolution in the x direction.

If each pixel in the display spans more data than the FFT size, then each screen pixel will represent the waveform data that would also be displayed in that pixel. However, if each pixel spans less data than the FFT size, then the data for that pixel is calculated by taking the number of points set by this field centred on the time of the pixel. In this case, the result will not be derived from data at that screen pixel, but also from data before and after the pixel.

Although you can zoom in as much as you like in the x direction, there is a limit to the detail you can see. Waveform features that are much smaller in time duration than the FFT size are “smeared” in time; the bigger the value of FFT size, the more the time smear.

x pixel inc To get the best screen resolution you should set this field to 1. This will cause the sonogram to be recalculated for each x axis pixel. However, you can speed up the calculation and drawing by setting values greater than 1. This effectively makes the screen pixels bigger in the x direction.

Skip This is another method of speeding up the calculation (but not the drawing). If you have the condition that each screen pixel spans much more than FFT size data points, Spike2 uses the FFT on all the data points that correspond to the pixel and accumulates the data. If you check this box, Spike2 only transforms the first FFT size data points. This can be useful if you want to display a multi-megabyte file. However, the resulting display will show the power spectrum at “sampled” points through the file, it will not represent the total power for each pixel.

Index

—A—

ADDACn, 43
Analysis commands, 82
Analysis of data, 9
Arbitrary waveform output, 44, 107
Arrays, 62
ArrConst(), 81, 92
 in result view, 80
ArrIntgl(), 71, 81
 in result view, 80
ArrSum(), 114
Asc(), 70

—B—

Backing up your data, 10
Basic screen layout, 3
Binary file export, 121
Binary file functions, 72
Binary file import, 119
Binary files, 117
BinShow1.s2s script, 119
BinSize(), 75, 90, 110, 116, 122, 123
BinToX(), 81, 123
Bitmap images, 12
BRAND, 38
BRead(), 73, 121
BReadSize(), 73, 120, 121
BSeek(), 120
 Command description, 120
BWrite(), 73
BWriteSize(), 73, 122, 123
BZERO and BNZERO, 39

—C—

CANGLE, 43
ChanComment\$(), 75, 110, 112, 116, 118
ChanData(), 77, 88, 90, 112, 116, 122
ChanDelete(), 79, 114, 115, 118
ChanDuplicate(), 79
ChanHide(), 79, 85, 114, 116
ChanKind(), 75, 110, 116, 122
ChanList(), 75, 91, 92
ChanMeasure(), 76
Channel duplicate, 79
Channel number, 3
Channel types, 6
ChanOffset(), 75, 110, 112, 116, 118
ChanScale(), 75, 110, 112, 116, 118
ChanShow(), 79, 84, 89, 110, 114, 116, 118
ChanTitle\$(), 75, 110, 112, 116, 118
ChanUnits\$(), 75, 110, 112, 116, 118
ChanValue(), 75, 77, 80, 85
Control of sampling, 95

Control of stimulation, 102
Cos(), 92
 Result array, 80
Count(), 76, 96, 97, 108
CRATE, 43
Cross-correlation, 55
CSZ, 43
Cursor functions, 67
Cursor use, 8
Cursor(), 78, 82
CursorRenumber(), 82
CursorSet(), 78, 82

—D—

DAC0, 36
Data analysis, 9
Data window, 3
DBNZ1 and LDCNT1, 37
DELAY, 36
DIGIN, 39
Digital marker channel, 6
Digital markers, 25
DIGOUT, 36
Display tools, 26
Dlg... family of functions, 71
DlgChan(), 71
DlgCheck(), 71, 94
DlgCreate(), 71, 91, 92, 94, 122
DlgInteger(), 71, 94
DlgLabel(), 71
DlgList(), 71, 122
DlgReal(), 71
DlgShow(), 71, 105
DlgString(), 71
DlgText(), 71
Document types, 5
Draw(), 78, 82, 108, 118
DrawMode(), 79, 108
 in result view, 80
Duplicate channel, 79
Duplicate window, 3

—E—

EditClear(), 82, 100
EditSelectAll(), 82, 100
Evaluate window, 57
Event channel, 6
Event correlation, 24
Event data display modes, 22
Example files, 1
Export graphics and text, 12

—F—

File types, 5
FileClose(), 75, 83, 85, 100, 102, 118, 122

 No query, 80, 115
FileName\$(), 78
FileNew(), 81, 85, 87
 Command description, 117
 Create data file, 120
 New offline data file, 110, 117
FileOpen(), 56, 74, 80, 82, 84, 86, 91
 Binary, 120, 122
 External text, 118
FilePathSet(), 74
FileSaveAs(), 80
Find next and last data item, 78
Find peaks, 85
for ... next, 61
FrontView(), 79, 82

—G—

Gradient of line, 77
Graphical sequence editor, 28
Grid(), 80

—H—

HALT, 36
Help system, 5

—I—

Idle routine, 97, 102
if statement, 60
Import data
 From binary file, 119
 From Text file, 117
Input(), 70, 82
Installation, 2
Interact(), 68, 78, 82, 84, 86
Interval histogram, 23, 54

—J—

JUMP, 38

—K—

Keyboard marker channel, 6
Keyboard markers, 25

—L—

Last data item, 78
LastTime(), 78, 79
LD1RAN, 38
LDCNT1 and DBNZ1, 37
Len()
 Array, 120, 121
 String, 70
Level channel, 6
LogHandle(), 82, 100

—M—

MARK, 41
 Marker filter, 7
 Wavemark data, 55
 Maths and Array arithmetic functions, 71
 Maximum and minimum, 77
 Maximum value, 78
 Maxtime(), 67, 78, 82, 90, 92, 98, 99, 110, 122
 Mean value between cursors, 77
 Measurement tools, 27
 MemChan()
 Command description, 109
 Event, 87, 108
 Marker, 84
 Waveform, 88, 90, 116, 118, 120
 WaveMark, 112
 MemDeleteItem(), 108, 115, 118
 Command description, 114
 MemDeleteTime()
 Command description, 114
 MemGetItem(), 114
 Command description, 113
 MemImport(), 85, 86, 110, 112
 Command description, 111
 Memory channel, 6, 108
 Memory channels, 83
 MemSave(), 91, 116, 117, 118
 Command description, 115
 MemSetItem(), 89, 91, 108, 114, 116, 118, 120
 Command description, 112
 Menu bar, 4
 Message(), 70, 78
 Mid\$(), 70
 Minimum value, 78
 MinMax(), 76, 78
 MOVI, 38
 Multi-Tasking, 2
 Multi-unit recording and marker filter, 7

—N—

Next data item, 78
 NextTime(), 78, 79, 85

—O—

On-line script skeleton, 100
 Optimise(), 78
 in result view, 80
 Output pulses, 36
 Output sequencer, 28, 36
 Arbitrary waveform output control, 34
 Clock rate, 28
 DAC scaling, 29

Graphical editor setup, 28
 milliseconds per step, 28
 On-line use, 102

—P—

Peak and trough, 77
 Peak extraction, 85
 Phase histogram, 24
 Picture export, 12
 PLS Output sequencer file, 5
 Power spectrum, 19
 Print\$(), 79
 PrintLog(), 60, 61, 82, 88
 Procedures and functions, 63
 Process(), 82
 Processing tools, 27

—Q—

Query(), 70

—R—

Rand(), 70, 108, 114
 Random delays and branches, 38
 Recording actions to a script, 58
 Rectify waveform, 88
 Renumber cursors, 68
 repeat ... until, 61
 REPORT, 41
 Result view, 3
 Result views, 80
 RMS level, 77

—S—

S2R Spike2 resource file, 5
 S2S Script file, 5
 Sample family of commands, 93
 SampleAbort(), 95
 SampleCalibrate(), 94
 SampleClear(), 93
 SampleComment\$(), 94
 SampleKey(), 38, 41, 103, 104, 105, 106
 SampleLimitSize(), 94
 SampleLimitTime(), 93
 SampleMode(), 93
 SampleReset(), 95
 SampleSequencer(), 94, 103
 SampleSeqVar(), 38, 105
 SampleStart(), 95, 101, 103
 SampleStatus(), 98, 99
 SampleStop(), 95, 101
 SampleText(), 95
 SampleTextMark(), 94
 SampleTimePerAdc(), 93
 SampleTitle\$(), 94
 SampleUsPerTime(), 93
 SampleWaveform(), 94
 SampleWrite(), 95
 Sampling, 15
 Configuration dialog, 11
 Digital outputs, 28
 Event data, 21
 Introduction, 9
 Marker data, 25
 Open a new data file, 15
 Output during, 28
 Waveform data, 15
 Sampling configuration commands, 93
 Sampling mode, 27
 Script language, 59
 Script window, 57
 Scripted analysis while sampling, 96
 Scroll bar, 3
 Seconds(), 79, 95, 108, 114
 Sequencer execution control, 102
 Sequencer table data, 105
 Sequencer variable manipulation, 103
 Serial line use from Spike2, 124
 SerialClose(), 124, 126
 SerialCount(), 124
 Command description, 126
 SerialOpen(), 124
 Command description, 125
 SerialRead(), 124
 Command description, 126
 SerialWrite(), 124
 Command description, 125
 SET, 36, 44
 SetPSTH(), 82
 SetResult(), 80
 Sinusoidal output, 43
 Slope of line, 77
 SMR Spike2 data file, 5
 Sonograms, 127
 Sound(), 36
 Spike shapes, 47
 Spike triggered averaging, 54
 Spreadsheet output, 14
 SRF Spike2 result view file, 5
 Standard deviation from the mean, 77
 State machine design, 99
 Status bar, 3
 Stimulus histogram, 23, 55
 String functions, 72
 Sum of result view bins, 77
 SXY XY view file, 5

—T—

Template formation, 51
 Template parameters dialog, 50
 Template setup window, 49
 Text and binary file functions, 72
 Text export, 13

Text files, 117
Text markers, 25
TextMark channel, 6
Time view, 3
Toolbar, 3, 26
Toolbar family of functions, 69
Toolbar use during data capture, 97
Toolbar(), 95, 101, 102, 103
ToolbarClear(), 95
ToolbarEnable(), 43, 101
ToolbarSet(), 95, 101, 103
ToolbarText(), 82, 101, 102
ToolbarVisible(), 78, 86, 100
ToolBarVisible(), 78
Trunc(), 90
TXT Text file, 5
Type of files, 5

—U—

UCASE\$(), 79
Using scripts on-line, 93

—V—

VAR in output sequence, 38
Variables for output sequencer, 38
Variables in a script, 60
Vector images, 12
View manipulation, 66
ViewKind(), 66, 121
Views and view handles, 64
ViewStandard(), 67, 78

—W—

WAIT, 39

WAVEBR, 45
Waveform average, 20
Waveform channel, 6
Waveform correlation, 20
WAVEGO, 45
WaveMark channel, 6
WaveMark data, 47
WAVEST, 45
while ... wend, 61
Window(), 82, 100, 118
WindowTitle\$(), 79
WindowVisible(), 82, 100, 114

—X—

XRange(), 67, 78, 101
XY views, 73