University of London

A signal analysis environment for MATLAB

Programming Guide

M. Lidierth (2009). sigTOOL: a MATLAB-based environment for sharing laboratory-developed software to analyze biological signals. *Journal of Neuroscience Methods* **178**, 188-196.
http://dx.doi.org/10.1016/j.jneumeth.2008.11.004

Version 0.92 November 2009
Author: Malcolm Lidierth
King's College London

http://sigtool.sourceforge.net

Note

This guide gives a summary of the features available in sigTOOL. For most functions described here, more detailed information is available from the on-line help for that function: type help or helpwin *functionname* at the MATLAB command prompt (after first running sigTOOL to set up the path) or open

…sigTOOL\documentation\sigTOOL.html

in a web browser.

Contents

Introduction ... 5
File import functions ... 5
Data Analysis ... 8
A GUI front-end ... 8
Data export ... 8
Accessing the sigTOOL functions ... 10
Loading sigTOOL data files ... 11
Data storage in the sigTOOL GUI ... 12
Adding Functions to sigTOOL ... 13
Adding menu items to a sigTOOL data view ... 13
Writing analysis routines ... 15
Calling analysis routines via scExecute ... 16
Representing channel data in sigTOOL ... 18
Overview ... 18
Data organization in sigTOOL ... 18
Representing timestamps in sigTOOL ... 22
Event data ... 23
Pulses ... 24
Markers ... 24
Analog data ... 25
Continuously sampled data ... 25
Periodically sampled data ... 25
Multiplexed data ... 27
The ChannelChangeFlag field ... 27
Event Filtering ... 28
Creating your own import functions ... 29
Methods associated with scchannel objects ... 37
Alphabetical list of methods ... 39
Building graphical user interfaces ... 53
Low level GUI tools: the jcontrol object ... 53
Creating jcontrol objects ... 54
Middle-level GUI functions ... 57
High-level GUI functions ... 61
jvDefaultPanel ... 61
jvAddPanel ... 63
Linking channel selections ... 65
Adding help to the menus ... 65
sigTOOL result objects ... 66
Appendix A: Detailed specification of sigTOOL data representation ... 75
Appendix B: MAT-file Utility Functions ... 81
Appendix C: sigTOOL data file specification ... 85
Appendix D: Memory mapping of data ... 87
Appendix E: Managing virtual memory ... 92

Introduction

sigTOOL is a signal analysis package for use within the MATLAB programming environment. It has been designed particularly to deal with neurophysiological data but may be of more general interest.

Ways of sharing neurophysiological data are being developed by many groups. sigTOOL has been designed to facilitate the exchange of data analysis functions as well as data between users.

sigTOOL provides four sets of tools as summarized below.

File import functions
A set of data import functions supporting common neurophysiological file formats. These import data into a standard MATLAB data file (a mat-file). Presently, import functions are provided for the following neurophysiological data formats[1]

ABF
: Molecular Devices Inc (Axon Instruments) format used by e.g. pClamp, AxoScope, ClampFit software.

CFS
: Cambridge Electronic Design Ltd– Signal software

MAP
: Alpha Omega

MCD
: Multi Channel Systems

SMR
: Cambridge Electronic Design Ltd– Spike2 software

NEV
: Cyberkinetics Inc

NEX
: Nex Technologies – NeuroExplorer software

PLX
: Plexon Instruments

---

[1] Support for Spike2 files is platform independent. Support for the remaining formats occurs through manufacturer supplied Windows application extensions (DLLs) and are therefore specific to the Windows OS. MAP, NEX & PLX files are supported through the manufacturers Neuroshare compliant DLLs and the Neuroshare MATLAB functions (www.neuroshare.org).

STA

Weill Medical School Spike Train Analysis Toolkit STAD/STAM format (if installed).

In addition, multimedia formats are supported via Micah Richert's[2] mmread import function for both sound and video formats (e.g. WAV, MPG, AVI). This manual describes how to write additional import functions.

[2] http://www.mathworks.com/matlabcentral/fileexchange/8028

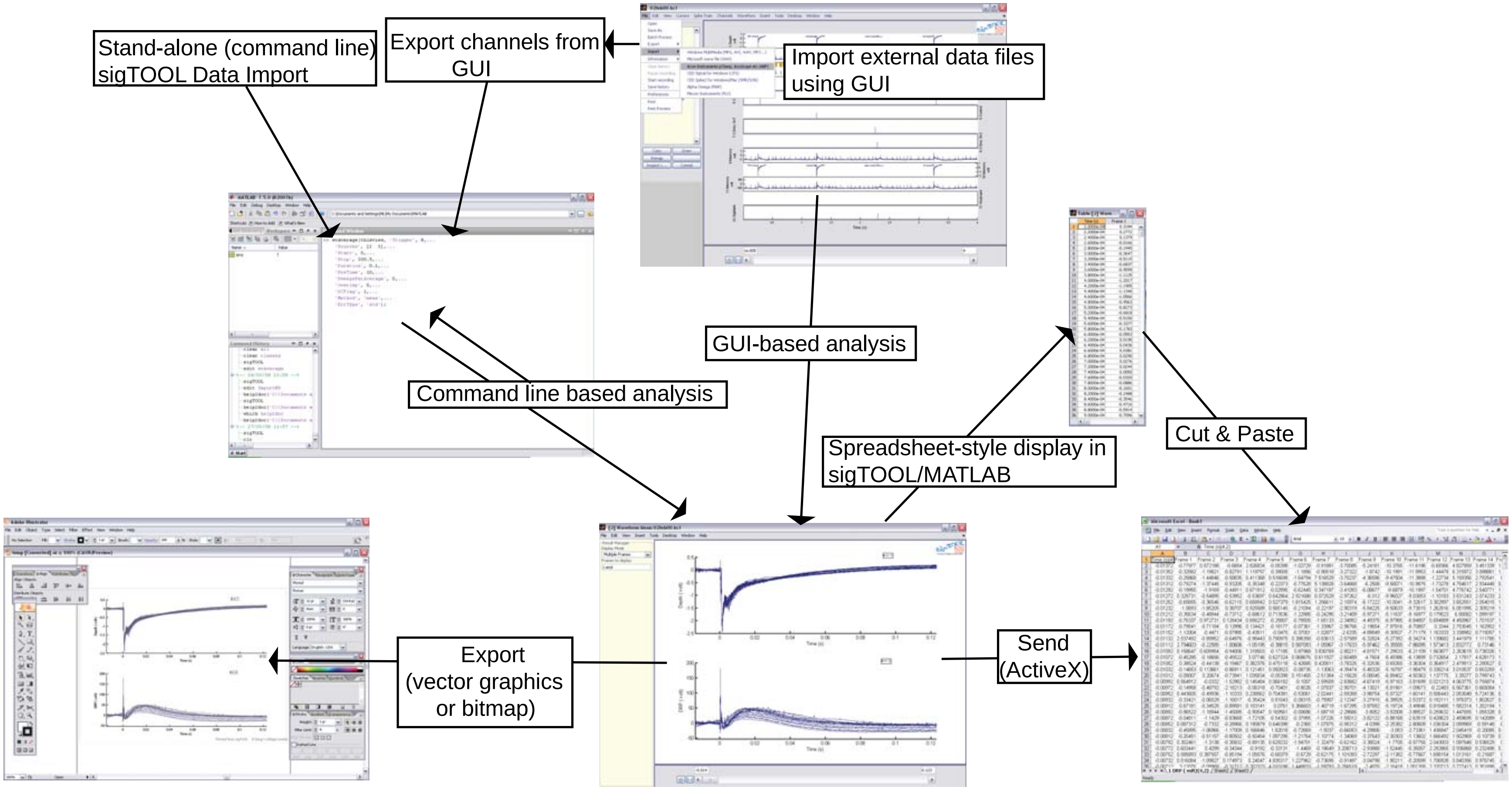


Figure 1.1

Summary of data flow in sigTOOL. Data can be imported from external files at the command line or via the GUI. In-built or user-written analyses are used to generate results. If these are formatted is a sigTOOL-compatible way, they can be plotted within sigTOOL and the results can be exported to external analysis or graphical programs. From the GUI, this can be achieved by selecting the appropriate menu options.

Data Analysis

sigTOOL provides a large set of functions for analysing the imported data. Object-oriented programming techniques have been employed to assist further programming by the end-user. Data channels are represented as MATLAB objects and basic methods have been overloaded to manipulate these e.g. to load and plot an imported data file:

```
channels=scOpen(filename); % Open the file
plot(channels{:}); % Plot the data in a standard sigTOOL data view
```

Other methods provide simple-to-use mechanisms for swapping between time, indices and subscripts to access and manipulate the data stored on these channels. These methods are used in the supplied routines for processing data e.g. waveform averaging, event correlation, digital filtering and may be used by the end-user to create custom analysis routines.

In sigTOOL, the results of analyses are also stored as objects. Methods for result objects provide easy access to the data they contain.

A GUI front-end

An easy-to-use graphical user interface has been developed to support the import and data analysis functions. The GUI provides a data analysis application running within MTALAB and can be used without any knowledge of the underlying code. It includes a powerful 'history' function that automatically records the user's actions to a MATLAB m-file which may then be run to batch-process multiple files.

As detailed below, this GUI is self-modifying. Users who develop their own routines may incorporate them easily into the GUI without the need to edit the sigTOOL source code. End-user developed suites of programs can be distributed as archives to other users. When the archive is unpacked into the appropriate sigTOOL subfolder, these functions will be made available on-the-fly from the sigTOOL GUI. End-user developed code does not disappear anonymously into sigTOOL: the GUIs allow end-users to credit their own work by displaying text/logos etc in the sigTOOL menus, progress bars and print-outs (see below). The recording of history files, as described above, also includes support for end-user written code.

Data export

Results displayed in sigTOOL can easily be transferred to other software. Double-clicking on a result displays numerical data in a spreadsheet format and these may be cut and pasted into other applications. Graphical results can be exported in vector or bitmapped formats to packages such as Adobe PhotoShop and Illustrator. In addition, on Windows platforms, context sensitive menus allow data transfer directly to other applications such as Excel and Sigmaplot using ActiveX.

Accessing the sigTOOL functions

You must run sigTOOL at least once each time you run MATLAB for it to set up the MATLAB path to access the sigTOOL functions. Make sure you have set up the path according to the instructions in the "Installing sigTOOL" guide

If you want to use the sigTOOL functions from the command line and not use the GUI, type

sigTOOL('nojvm')

at the command prompt. This will set up the search path to include:

…..sigTOOL/program + all subfolders

…..sigTOOL/CORE + all subfolders

It will not include any toolkits located in the sigTOOL main folder on the MATLAB path. This will need to be done manually.

For a full list of all sigTOOL functions, open the sigTOOL html file in the sigTOOL/documents folder. This lists the help text from all files[3] and contains a Contents and Alphabetical Index section.

[3] To make sure this HTML file is up-to-date and contains help for your own and any third party m-files run help2doc('…/sigTOOL') where …/ is the path to the sigTOOL folder.

Loading sigTOOL data files

sigTOOL stores data in a standard MATLAB data file (a MAT-file)[4]. These files are written in MATLAB Level 5 Version 6 format. This ensures backwards compatibility with Version 6 of MATLAB (~2000). Note however, than sigTOOL requires MATLAB R2006a or later for full functionality.

To distinguish sigTOOL files from other MAT-files, they are given the .kcl extension instead of .mat. However, the standard MATLAB *save* and *load* commands can be used with these MAT-files (use the –mat option to force MATLAB to recognize the files as MAT-files despite the .kcl extension).

The simplest way to load a sigTOOL data file from the MATLAB command prompt is with scOpen:

```
channels=scOpen(filename);
```

scOpen() maps the data into the elements of a standard MATLAB cell array. Each element of the cell array contains the information about one channel. Each element is a sigTOOL channel object full details of which are provided below (Representing data in sigTOOL).

As the data channels are represented as MATLAB objects, standard MATLAB commands can be overloaded to deal with them e.g.

```
channels=scOpen(filename);
plot(channels{1:10});
```

loads the data file and plots channels 1 through 10 in a sigTOOL data view. In most cases, programmers will be able to access the channel data exclusively through these methods and need have little knowledge of the internal organization of the channel data.

The channel cell array may be padded with empty entries as functions written to import data into sigTOOL from other file formats generally maintain the original channel numbering: if you have data on channels 1 and 3, the cell array will have three elements but the 2nd will be empty (as long as the author of the relevant import function has followed this convention).

Note that the virtual memory needed for a channel is not assigned when the data are loaded with scOpen. Instead, memory space will be allocated on-the-fly.

---

[4] sigTOOL channel objects encapsulate a number of other objects that are custom-defined in sigTOOL. These isolate sigTOOL from the file data source and format. As a consequence, sigTOOL code is not dependent on the use of the MAT-file format. Other formats may be supported in future versions (e.g. HDF5).

Data storage in the sigTOOL GUI

When sigTOOL is run, it sets up a figure window. This figure is a MATLAB object and its properties may be accessed using the MATLAB *get* command and altered using *set*.

There are several ways to associate data with a figure (or any other object) in MATLAB. The method used in sigTOOL is through the object's application data area. This is accessed using *getappdata* and *setappdata*.

When you open a file in the GUI, sigTOOL opens the file using scOpen and places the channel cell array in the application data area of a sigTOOL figure. This can be done from the command line as follows:

```
>> channels=scOpen(filename)
>> fhandle=plot(channels{:});
```

where fhandle is the figure handle returned by the plot function. Note the use of {:}. This causes a channel list to be passed to the plot method, rather than the cell array.

You can access the channel data in a sigTOOL data view from the command line by typing:

```
>>channels=getappdata(fhandle, 'channels');
```

Adding Functions to sigTOOL

Adding menu items to a sigTOOL data view

When sigTOOL is run, it scans the …/sigTOOL/program folder and all its subfolders looking for folders/functions prefixed by “menu_”. These functions are then added to the sigTOOL menu. The sigTOOL menu can therefore be managed dynamically simply by dropping new functions into the folder tree. Generation of menus is carried out by the *dir2menu* function. After running sigTOOL, type “helpwin dir2menu” at the command prompt to see full help details for this function.

As an example, let’s add a trivial function to the menu. Use your system file management software to navigate to the …sigTOOL/program folder and create a new subfolder called “menu_MyLab”. Now run sigTOOL from the MATLAB command prompt. Note that a new menu labelled MyLab has appeared on the sigTOOL figure menu.

Next, create a function inside the menu_MyLab folder. Open the MATLAB m-file editor and create a new m-file. We want this function to have variable numbers of input and output arguments so its first line needs to be something like:

```
function varargout=menu_MyNewFunction(varargin)
```

The dir2menu function will call this function with a single input of zero and will accept 3 output arguments from it. The first section of code needs to handle this call:

```
if nargin==1 && varargin{1}==0
    varargout{1}=true;
    varargout{2}='My New Function Label';
    varargout{3}=[];
    return
end
```

Create this file and save it as
…./sigTOOL/program/menu_MyLab/menu_MyNewFunction.m

Now run sigTOOL again. Click on the “MyLab” menu item. Note that a drop down list appears containing “My New Function Label”. If you click on this nothing will happen: we have not yet included any code to run from the menu.

To include some code, add the following to the end of menu_MyNewFunction.

```
[button fhandle]=gcbo;
fprintf('Button handle %d\n', button);
fprintf('Figure handle %d\n', fhandle);
return
end
```

In this case, button is the handle of the menu item we have just created. fhandle is the handle of the sigTOOL figure window. These are returned using the MATLAB get current button object (gcbo) command.

Click the “My New Function Label” again. The fprintf commands above will print the values to the MATLAB command window (If this does not happen try opening a new sigTOOL window. Menu callbacks such as the one created here can be buffered in memory so that changes you make to them may not take effect instantly). Here’s what the function should look like.

```
function varargout=menu_MyNewFunction(varargin)
if nargin==1 && varargin{1}==0
    varargout{1}=true;
    varargout{2}='My New Function Label';
    varargout{3}=[];
    return
end
[button fhandle]=gcbo;
fprintf('Button handle %d\n', button);
fprintf('Figure handle %d\n', fhandle);
return
end
```

Now open one of the sigTOOL demonstration data files using the File->Open menu. Replace the fprintf commands above with:

```
channels=getappdata(fhandle,'channels');
assignin('base','channels', channels);
clear('channels');
```

and save the file. Now run the routine again (re-running sigTOOL first if you have to).

```
 channels=getappdata(fhandle,'channels');
```

retrieves the channel data associated with this figure when you opened the file returning a local copy of the data in channels. Next we pass this data to the MATLAB base workspace using

```
assignin('base','channels', channels);
```

.

Go to the MATLAB command prompt, type channels{1} and MATLAB will print a summary of the contents of the channel 1 data. You can access this data using standard MATLAB commands.

Finally,

```
clear('channels');
```

deletes the copy of channels that was local to the menu_MyNewFunction workspace. This is unnecessary here, but it is good practice not to leave a chain of local copies of the file data when calling multiple functions. Clearing the data ensures that the scRemap() function will be able to free virtual memory that is no longer being used.

Ordinarily, the "menu_" function will call further functions to analyze the data and present results. The following section gives some simple rules to follow when writing these functions. If you follow these rules, features in the GUI such as recording a user history to use in batch processing of further files will work automatically.

Writing analysis routines

When writing routines to call from the sigTOOL menu, use the following conventions:

1. Provide the handle of the sigTOOL data view as the first input.
   ```
   myFunction(fhandle,...........)
   ```
   The routine should set up a local copy of the channel data by calling getappdata:
   ```
   channels=getappdata(fhandle, 'channels');
   ```

If the channel data are altered, and you want to save the changes, the application data area should be updated with a call to setappdata before the analysis routine returns[5]:

```
setappdata(fhandle, 'channels', channels);
return
```

2. Some users will want to work from the MATLAB command line so you should allow the figure handle to be replaced by a sigTOOL channel cell array:

```
myFunction(channels,...........)
```

3. If a routine allows only one channel on input, it should allow the channel to be passed as an object or as a cell element.

```
myFunction(channels{3},...........)% Passes the object
or
myFunction(channels(3),...........)% Passes the cell
```

The scParam function provides a convenient way to process these inputs. If fhandle was not supplied, it will be returned empty by scParam.

```
function MyFunction(varargin)
[fhandle channels]=scParam(varargin{1})
.
.
return
end
```

Calling analysis routines via scExecute

The scExecute function acts as a gateway between the "menu_" function and an analysis function. If you call your analysis function through this gateway and recording is switched on, scExecute will automatically write the required code into the history record. scExecute also honours the "Apply to all open files" selection that is available from the standard sigTOOL GUIs.

Lets suppose that myFunction(fhandle, input1, input2) is the function you want to call. Instead of calling it directly, use the following code:

```
arglist={fhandle, input1, input2};
scExecute(@myFunction, arglist, flag);
```

---

[5] However, if you alter an object that is passed by reference, e.g. a memmapfile object, all copies of that object will be updated immediately, including those in the application data area.

Here, arglist is a cell array containing the arguments needed by myFunction. Call scExecute, passing the handle to myFunction as the first input and arglist as the second. scExecute will now invoke myFunction passing arglist to it.

The flag above is true or false. If true, scExecute will cycle through all open files applying myFunction to each one in turn.

scExecute will also add the code to your history recording that will cause myFunction to be executed when you play the recording (for further details see …………..).

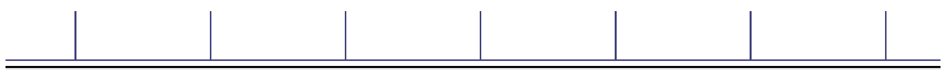
Representing channel data in sigTOOL

Overview

Data organization in sigTOOL

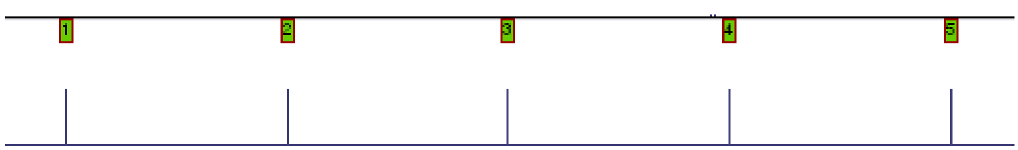
To understand the Channels menu, you need to know a little about how channel data are organized in sigTOOL.

The simplest data type is an event channel where the timestamps of discrete events are stored. These might be stimulus markers for example.



The channel type might be ‘Rising Edge’, ‘Falling Edge’ or just ‘Edge’.

Next we can associate each event with some data. The events above might be synch pulses from a video camera for example. The demo.kcl file includes an example of this.



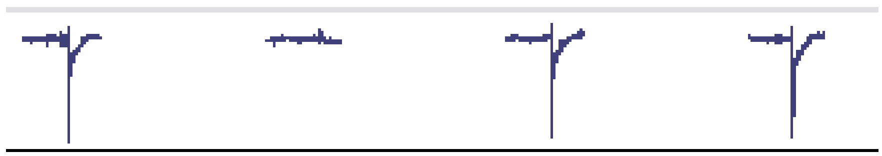
You can view the data associated with each synch pulse by clicking on the green numbered squares. The numbers are the marker values for each event. In this case, the markers are the video frame numbers. In other circumstances the markers might be a code representing, for example, test and control stimuli where these were alternated during an experiment.

We have identified the three main components of a sigTOOL channel.

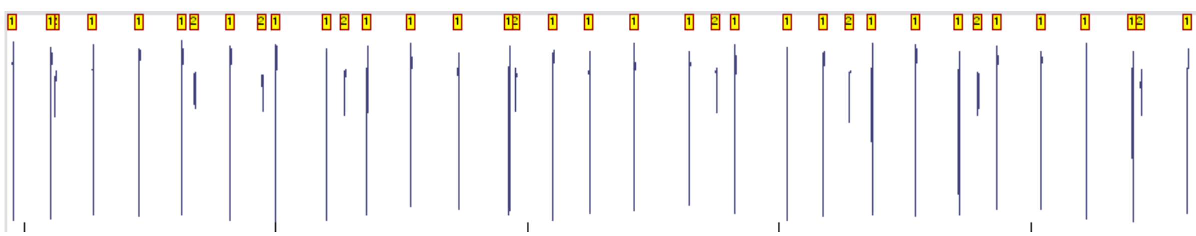
1. Timestamps associated with events
2. Data associated with each event
3. Marker data to classify the events

We can also associate more than one timestamp with each event:

In this case we record both the rising and the falling edges and the channel type is ‘Pulse’. Each pair of timestamps defines an episode of time. We can then associate an epoch of data with this period e.g. waveform data from a single oscilloscope sweep. In sigTOOL, the waveform data will then be plotted rather than the pulse. This is an ‘Episodic Waveform’

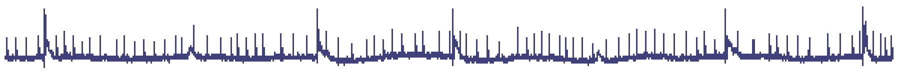


We can also associated markers with each epoch. In the case below these markers identify the spike of a single neurone in a multi-unit recording:



The markers can be single values or vectors of numbers, usually integers. The marker data can be more complex however, and include strings or structures perhaps containing metadata.

What if data are sampled continuously? That is just a special case where we have only one data epoch with one timestamp at the start of sampling and another at its end.



Finally we may need to associate a specific trigger time with the episodic data. In that case we just store three timestamps: the first marks the start of sampling for an epoch, the second marks the trigger time and last marks the end of sampling for each epoch. These channels are of two types:

1. Where there is no fixed temporal relationship between the three timestamps in different epochs. The epochs may vary in length and the length of sample before and after the trigger can be variable. These are labeled as ‘Episodic Waveform’ channels as before.
2. The temporal relationship is constant. All epochs are of the same length and there is a constant pre- and post- stimulus sample length. In this special case, the channel is labeled as a ‘Framed Waveform’.

Some file types allow multiplexed data to be stored on a single channel. In sigTOOL, these channels are stored just as above and the subchannels are interlaced in the data array. Most sigTOOL analysis routines do not support multiplexed data but the subchannels can be extracted to a channel of there own as described below.

The very simple channel organization described above is versatile enough to represent pretty much any waveform data. As shown above, it can also be used to represent other data formats such as video. To customize the treatment of non-waveform data the ‘Custom’ keyword is include in the channel type description and the file import function associates a MATLAB m-file function with the channel to process the data. For the video channel in demo.kcl this is the scViewImageData function. This displays the image when you click on the markers in the sigTOOL data view. Other functions could be defined to process any other type of data. For example, if you wanted to embed electronic notes in a file you could associate them with a timestamp placed at some appropriate point in the file. Place the text associated with the note in the data field and define a custom function to display the text in an editor when the marker is clicked. If the text was in a mark-up language, you could open it in the system web browser and include graphics or hyperlinks to web sites or other documents e.g. PDF files.

In practice, the channel structures are incorporated into custom designed objects of the scchannel object class. This has the advantage that methods can be written that are specific to the scchannel class and these can handle much of the work e.g. converting between time and matrix indices. These methods will be described later. For the present we will concentrate on the contents of the scchannel objects. There are five basic fields:

tim: which stores the times of events or the start and stop time for sampling a waveform together with an optional trigger time
mrk: which stores a marker value associated with the time(s) in *tim*
adc: which stores the data associated with timestamp(s) in *tim*. The data in *adc* may be a MATLAB vector, or a matrix including multi-dimensional matrices. *adc* is so named because it will commonly contain data from an analog-to-digital convertor but it is much more versatile and may be used to store many other data types.
hdr: which stores information used to interpret and display the data in the other fields.

Three additional properties are added when the scchannel objects are constructed. These are the

EventFilter:
which is used to select timestamps/data epochs through the scchannel object methods.

CurrentSubchannel:
which is used to select a subchannel from multiplexed data

channelchangeflag:
a structure of flags that indicate whether the channel has been altered since it was loaded. This is used to control file updates when data are saved.

The scchannel objects are assigned to cell arrays that have one element for each channel of data. In the examples that follow we ignore this for clarity, but remember that *tim*, *adc* and so on should normally be *channel{x}.tim*, *channel{x}.adc* etc.where *x* is the channel number.

Note in particular that different channel types are built in a coherent way. Events are stored in *tim*, if markers are available they are added to *mrk* and if additional data, such as waveform data are available they are added to *adc*. The *channeltype* description in the *hdr* field determines whether sigTOOL should focus on the *tim*, *mrk* or *adc* fields for handling the data.

Representing timestamps in sigTOOL

Timestamps are generally stored as integers in the source data files that are imported into sigTOOL. This offers an advantage because data processing of integer data types is not subject to rounding errors[6]. It has the disadvantage that non-integer values can not be represented. sigTOOL overcomes this by representing timestamps where possible as “flints” i.e. as floating point representations of integers. sigTOOL makes use of the property that all integers between 0 and 2^{52} are represented exactly in IEEE double precision floating point and that all algebraic operations involving two flints that should produce a flint will do so, i.e. no rounding errors occur as long as flints between 0 and 2^{52} are involved. To take advantage of this, timestamps are associated with two scaling factors, Scale and Units. These are related as follows:

Timestamp x Scale x Units = time in seconds

The Timestamp and Scale are typically flints so their product is also a flint without any rounding errors. These values are used in the sigTOOL analysis routines to avoid rounding errors. The Units factor allows these results to be converted to a standard time unit (seconds). As a practical example, take a source data file where events have been timed to an accuracy of 25µs. Timestamp will contain the number of clock ticks for an event (an integer or flint). Scale would be set to 25 giving a flint result in microseconds. Units would be set to 10^{-6} to convert the result to seconds.

To represent a timestamp between clock ticks simply set the value of timestamp accordingly e.g for 62.5µs, set Timestamp to 2.5 (i.e. 2.5*25=62.5µs). In practice, Timestamps are usually stored as integers to save memory so these values will need to be cast to floating point before assigning a non-integer value.

[6] These rounding errors are generally too small to be worth considering when analyzing waveforms but can become a nuisance in spike train analysis.

A consequence of using both Scale and Units settings is that, in two- or multi-channel analyses, we need to ensure that timestamps from all channels are scaled in the same way. To simplify this, the sigTOOL ImportXXX functions always use the same value for Units when importing a data file. sigTOOL analysis functions *assume* that Units will be equal on all channels.

Note that, for convenience, the examples below show timestamps scaled to seconds.

- **Event data**

Timestamps representing discrete events e.g. Rising and falling TTL edges
Timestamps are stored in the *tim.tstamps* field. This can be a column vector in which each row represents the time of a single event. These events may be rising or falling edges. The channel type is distinguished by the *channeltype* field in the *hdr* field which should be a string: ‘Edge’, ‘Rising Edge’ or ‘Falling Edge’. The example below shows a Rising Edge with three timestamps.

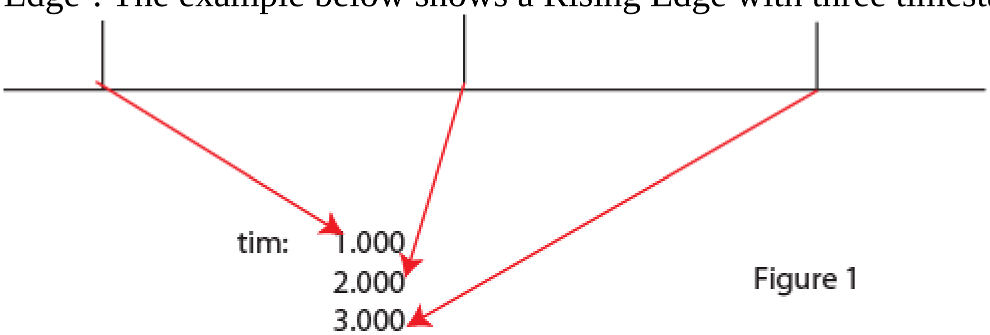


Figure 1

Note here the timestamps are represented in seconds as a double precision vector. As detailed below, they will normally be represented in other units of time e.g. multiples of 25µs.

Timestamps are normally represented in a *tstamp* objects. This is a MATLAB class defined within sigTOOL which provides a memory efficient way to represent timestamps that are mapped to a file on disc. The *tstamp* class is discussed further below.

- **Pulses**

We may be interested in both the rising and falling edges of a pulse. In this case the timestamps for each are stored as a vector. Each row of *tim* contains the time for the rising edge in column 1 and the falling edge in column 2.

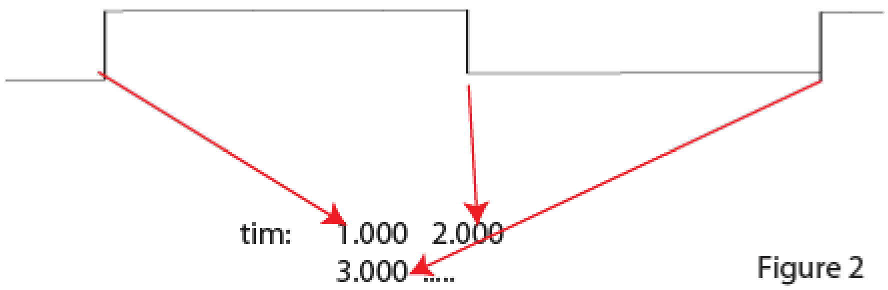


Figure 2

If the first recorded event is the Falling Edge of a pulse, tim(1,1) should be set to be zero (or negative – but this is not recommended).

General note: When stored in a tstamp object, the timestamps will often (but not invariably) be an integer data class on disc. If that class is unsigned, it will not be possible to store a negative stamp.

- **Markers**

A marker can be associated with an event or pulse (or any other channel type) by using the *mrk* field. Each row of *mrk* contains the markers associated with the timestamps in each row of *tim*.

This feature can be used to classify timestamps and to group those that represent similar events at different times.

If *mrk* is a simple MATLAB matrix, it can contain single or multiple values in each row and can be any numeric data type (though it makes sense to stick to integers).

e.g.:

tim:	1.000	3.000	mrk:	1	0	0	0
	5.000	7.500		2	0	0	0
	9.000	10.000		1	0	0	0
	.						

.

In this case *mrk* has 4 markers per row of *tim*, but only the first is used. In the illustrated case, *mrk* could be a *uint8* matrix.

Alternatively, *mrk* can contain a cell structure array or cell array of structures. In this case complex metadata can be used.
mrk can be left empty (*mrk=[]*) if it is not needed.

- **Analog data**

Digitized analog data is stored in the adc field. sigTOOL supports analog data that are sampled continuously or periodically and also supports multiplexed data.

Continuously sampled data

For continuously sampled data, put the start time and end time of sampling in the first row of *tim*. Then add the digitized analog data as a column vector to *adc*. e.g.

tim: 0.000 20.000
adc: 1.500
1.6
1.7
.
.

The *adc* field should contain an *adcarray* object. The *adcarray* is a sigTOOL-defined MATLAB class that allows digitized data on disc to be mapped into memory efficiently and allows large volumes of data to be represented in the *adc* field through virtual memory. The *adcarray* class is described more fully below – for most purposes the *adcarray* behaves as though it were a double precision array. In fact, the *adc* field can contain a double precision array but this is not recommended – some sigTOOL core functions will fail if *adc* does not contain an *adcarray* object).

Periodically sampled data

Remember that each row of *tim* can contain a pair of timestamps. To associate periodically sampled analog data with each row of *tim*, organize the data in columns in *adc*. Each row of *tim* should contain the start time and stop time for sampling, e.g. for three periods of sampling:

tim: 0.000 1.500
3.000 4.000
9.000 12.00

adc:	1.5	4.5	7.2
	1.6	4.7	7.4
	1.7	4.6	7.7
	.	.	.
	.	.	.

Note that the sampling periods do not have to be the same length.

General note:
In the example above, the data in *adc* associated with each row of *tim* is placed in the columns of *adc*. In this case, *adc* is a 2-dimensional matrix and different data periods are represented in the highest (2^{nd}) dimension of the matrix. This is used as a general rule in sigTOOL when dealing with higher dimensional matrices: data associated with the *nth* row of *tim*, are always placed in the *nth* element of the highest dimension of *adc*. If we associate each timestamp (i.e. row of *tim*) with a video frame containing a 300x300x3 RGB image, *adc* will be a 300x300x3xN matrix, where N is the row of *tim*, and the number of the frame in the video.

It may be that the periods of sampling are also associated with a trigger event. Often, there will be a fixed relationship between the start and end of sampling and the trigger e.g. with an oscilloscope sweep with a 10% pre-trigger time. However, some commercial data capture packages buffer data in such a way that there may be variable pre-trigger and post-trigger sampling periods. sigTOOL accommodates these possibilities by allowing each row of *tim* to contain three entries (columns): the start time, the trigger time and the end time for sampling e.g.

tim:	0.000	1.000	1.500
	3.000	3.100	4.000
	9.000	9.600	12.00
adc:	1.5	4.5	7.2
	1.8	4.7	7.4
	1.9	4.6	7.7
	.	.	.
	.	.	.

Note that column 1 always contains the start time and the final column always contains the end time for sampling. This simplifies programming. In many cases the length of the entry in *tim* can be ignored using the MATLAB *end* function. For the *nth* sampling period:
To find the start time use:

tim(n,1)

To find the end time use:

tim(n,end)

This syntax can be used without error whether *tim* contains one, two or three columns. Only if we need to test for a trigger time is there a need to examine the length of the row vectors in *tim*:

if size(tim,2)= = 3……

Multiplexed data

Multiplexed data are stored by interleaving:

tim:	0.000 1.000 1.500			
	3.000 3.100 4.000			
adc:	1.5	4.5		subchannel 1
	2.7	-1.1		subchannel 2
	4.8	7.5		subchannel 3
	-0.1	12.2		subchannel 4
	1.6	4.7		subchannel 1
	2.6	-0.9		subchannel 2
	4.9	7.6		subchannel 3
	-0.05	12.7		subchannel 4
	.	.	.	etc.
	.	.	.	

Once a column of data has been extracted, it may sometimes be convenient to use the MATLAB *reshape* function to separate the channels into columns.

The ChannelChangeFlag field

Each sigTOOL data channel has a *ChannelChangeFlag* field added when the data is loaded from disc. This is a structure with four logical fields indicating whether data in the *tim*, *mrk*, *adc* or *hdr* have been changed:

```
>> chan{1}.channelchangeflag
ans =

    hdr: 0
    adc: 0
    tim: 0
    mrk: 0
```

When a file is loaded, these fields are set to false (0). If you alter the data in any of the data fields, you should set the appropriate flag in *ChannelChangeFlag* to true to indicate that the file needs to be updated.

Event Filtering

Event filtering is used to select the events or epochs of data that you wish to analyze. With a spike train, you might choose only those spikes that have a marker value of 2. With episodically sampled data, you could choose to average only those epochs that are odd-numbered.

Event filtering is implemented through the EventFilter field of an scchannel object. This is a structure with two fields:

> Mode, which is either ‘on’ or ‘off’
> Flags, which is vector containing a logical flag for each event or epoch in the channel.

Event filtering distinguishes between ‘physical’ events or data epochs and ‘valid’ events or epochs. Physical epochs are all those present in the channel. If EventFilter.Mode is ‘on’, valid epochs are the subset of physical epochs for which EventFilter.Flag is true. If EventFilter.Mode is ‘off’, all physical epochs are valid epochs.

Event filtering is applied through the methods of the scchannel objects (not those of adcarray or tstamp objects).

Creating your own file import functions

sigTOOL supports the import of many proprietary neuroscience data formats as well as multimedia files as standard. To support a new format you need to create a ImportXXX file for that format and a menu_ file to include the format in the sigTOOL GUI. This section runs through that process using the ImportWAV file as an example. It is assumed that you have read “Representing channel data in sigTOOL” above.

Note that full details of the sigTOOL data file format, and of its data and header structures are given in Appendices A and C in this manual. When loading data, sigTOOL uses some custom-defined classes: adcarray and tstamp. These are described in Appendix D. If you have very lengthy data files, and can not fit all data from each channel into memory at one time, you may need to use the MAT-file utilities described in Appendix B to create the data file.

First, create a menu_ImportXXX function as detailed above. In this case:

```
function varargout=menu_ImportWAV(varargin)

% This sets up the menu in the GUI as detailed in the previous
section
if nargin==1 && varargin{1}==0
    varargout{1}=true;
    varargout{2}='Microsoft wave file (WAV)';
    varargout{3}=[];
    return
end

% This is the code that is run when the menu item is selected
if nargin>=2
    scImport(@ImportWAV, '*.wav');
end

return
end
```

Note that the ImportWAV file is not called directly. Instead, the menu_ file calls the scImport function passing the handle of the ImportWAV file to it, together with a string that specifies the file extension to be loaded: in this case ‘*.wav’. Among other things, scImport will display a file selection box to allow the user to select a file and then open a sigTOOL data view once data have been loaded.

Second, create the ImportXXX file.

Step 1:
Create the file using a standard calling format. For ImportWAV this is

```
matfilename=ImportWAV(filename, targetpath)
```

The inputs are:
filename
A string. The name of the file to import (together with its full path)
e.g. C:\WINDOWS\Media\chord.wav
targetpath
A string. The path of the folder to which the imported data file will be written. If empty, the file will be written to the same folder as the source file.

The output is:
matfilename
A string. The name of the generated file together with its full path.

Step 2:
Include code to generate the sigTOOL kcl data file. This is simple using the scCreateKCLFile function

```
matfilename=scCreateKCLFile(filename, targetpath);
if isempty(matfilename)
    return
end
```

If file creation fails, simply return. scImport will handle the failure.

Step 3:
Run through each channel in the source file loading and saving the data to a sigTOOL data file.
In the case of ImportWAV, we load all channels with a single call to the MATLAB standard wavread function then save them individually. To load the data

```
[audio, Fs]=wavread(filename);
```

In this case audio has one column for each channel so

```
for chan=1:size(audio,2)
```

will loop over these.

Next, create an empty header for the channel

```
hdr=scCreateChannelHeader();
```

and fill in the fields we need.

```
hdr.channeltype='Continuous Waveform';
hdr.channel=chan;
hdr.title=['Audio' num2str(chan)];
```

Next, deal with the data. We need to create a structure with 3 fields, one for each of the waveform data, the event data and the marker data: data.adc, data.tim and data.mrk.

In the present case the data.adc field is easily dealt with by putting the relevant column from audio into data.adc

```
data.adc=audio(:,chan);
```

If we had episodic data, we would put a 2-dimensional matrix into data.adc with one column for each data epoch. Next, create the header information to interpret these data. These go in the adc field of the header. We have one dimensional data, so need only to give one label. This would usually be time. With 2-D data, we might set Labels to {'Time' 'Epoch'}

```
hdr.adc.Labels={'Time'};
```

Next, set the sample interval. The sample rate was returned by wavread so the interval is just its reciprocal. In sigTOOL, two numbers are used to represent the interval and these are stored as a 2-element vector. The interval in seconds is there product. We will convert the interval to microseconds by multiplying by 10^6 and set a scaling factor of 10^-6 in the 2nd element to convert back to seconds.[7]

```
interval=1/Fs;
hdr.adc.SampleInterval=[interval*10^6 1e-6];
```

Now set the number of data points and the limits of the data for the display

```
hdr.adc.Npoints=length(audio);
hdr.adc.YLim=[min(audio(:,chan)) max(audio(:,chan))];
```

Finally, set the target class to use when loading the data through the scOpen function. scOpen will memory map the file and load the data via a sigTOOL-defined custom class called an adcarray so set TargetClass to that.

```
hdr.adc.TargetClass='adcarray';
```

[7] Appendix A explains why this is done.

Next, we need to set the event data. The waveform is continuous so there is only one epoch. This is set in microseconds as before[8]:

```
data.tim=[0 (length(audio)-1)/Fs]*1e6;
```

and we need to add some details to the hdr.tim field

```
hdr.tim.Units=1e-6;
hdr.tim.TargetClass='tstamp';
```

Again, scOpen may memory map the data so we specify a custom class again as for the adc field. This time it is ‘tstamp’.

Step 4:

Assign the markers. In this case we have no marker data, but need to specify this explicitly

```
data.mrk=[];
```

and set up the header information. In this case the markerclass is empty. Typically it might be ‘uint8’.

```
hdr.markerclass='';
```

To save these data call scSaveImportedChannel specifying the file name returned by scCreateKCLFile above and the channel number. Supply the data and hdr structures on input.

```
scSaveImportedChannel(matfilename, chan, data, hdr);
```

Finally, clear the imported data and repeat the loop for the remaining channels

```
clear('data','hdr');
```

Step 5:

With all the data loaded, add the sigTOOL version number to the file

```
sigTOOLVersion=scVersion('nodisplay');
save(matfilename,'sigTOOLVersion','-v6','-append');
```

Note that if audio is returned empty from wavread, the code above will create a kcl data file with no data. We can tidy that by slightly changing the order of the original code. Here is the entire function with some tidying.

```
function matfilename=ImportWAV(filename, targetpath)

% Call MATLAB builtin wavread
[audio, Fs]=wavread(filename);
```

[8] Always use the same value for hdr.tim.Units for each channel that is imported. Many sigTOOL functions assume this equality.

```
if isempty(audio)
    matfilename='';
    return
else
    % Set up MAT-file giving a 'kcl' extension
    matfilename=scCreateKCLFile(filename, targetpath);
    if isempty(matfilename)
        return
    end
end

% Save data to sigTOOL file
% One sigTOOL channel for each audio channel

    for chan=1:size(audio,2)
        hdr=scCreateChannelHeader();
        hdr.channeltype='Continuous Waveform';
        hdr.channel=chan;
        hdr.title=['Audio' num2str(chan)];

        data.adc=audio(:,chan);

        hdr.adc.Labels={'Audio'};
        interval=1/Fs;
        hdr.adc.SampleInterval=[interval*10^6 1e-6];
        hdr.adc.Npoints=length(audio);
        hdr.adc.YLim=[min(audio(:,chan)) max(audio(:,chan))];
        hdr.adc.TargetClass='adcarray';

        data.tim=[0 (length(audio)-1)/Fs]*1e6;
        hdr.tim.TargetClass='tstamp';
        hdr.tim.Units=1e-6;

        data.mrk=[];
        hdr.markerclass='';

        scSaveImportedChannel(matfilename, chan, data, hdr);
        clear('data','hdr');
    end


sigTOOLVersion=scVersion('nodisplay');
save(matfilename,'sigTOOLVersion','-v6','-append');

return
end
```

Adding custom data to an imported file

scImport allows you to specify a function that will be invoked after the import is complete and that can add data to the imported file. This might be used, for example, to add metadata to the sigTOOL data file. It can also allow you to add information that has not been included by the ImportXXX command. To use this feature, call scImport with three input arguments e.g.

```
scImport(@ImportNS, '*.mcd', @LocalPostProcess)
```

The function LocalPostProcess should take two input arguments which will be supplied automatically by scImport. These are the names of the source file being imported, and the name of the sigTOOL data that has been generated. Both need full folder paths.

```
function LocalPostProcess(source, target)
CustomVariable.SourceFileName=source;
save(target, 'CustomVariable', '-append', '-v6');
return
end
```

This example simply adds a custom variable to the sigTOOL data file. Note that you need to specify the –v6 and –append options.

LocalPostProcess can also take addition arguments. Specify these by passing a cell array to scImport:

```
scImport(@ImportNS, '*.mcd', {@LocalPostProcess, opt1, opt2});
```

and supporting the extra arguments in LocalPostProcess:

```
function LocalPostProcess(source, target, opt1, opt2)
    etc
```

Setting up the adcarray and tstamp fields

In the example above, wavread returns double precision data and there are no units. More usually, integer data will be returned when reading data sampled with an analogue-to-digital convertor. You will then need to set up the adc and tim fields of the header to allow the scOpen function to construct the required adcarray and tstamp objects correctly when loading the data. This is straight-forward and full details are given in Appendices A and D.

For the adcarray, you need to supply a scaling factor and offset to convert the data to the required units and, if required, the handle of a function to transform the data. Take an example where a 16-bit analog-digital-convertor was used and the input range was ±5V. One bit is used for the sign, so 1 LSB represents $5/2^{(16-1)} = 1.5259\times10^{-4}$V. A 1000x amplifier was used, so the Scale is set to 1.5259×10^{-4} and the Units to ‘mV’ to convert the ADC values to real-world values. No DC offset was applied so this is set to zero:

```
hdr.adc.Scale=1.5259e-4;
```

```
hdr.adc.DC=0;
hdr.adc.Units='mV';
hdr.adc.Func=[];
```

Timestamps will, similarly, usually be stored on disc in an integer format and the tstamp requires similar inputs to an adcarray:

The Scale property is used to multiply the data on disc to return a time stamp in base clock ticks. This will be returned as a floating point number but will usually be a "flint" i.e. a floating point representation of an integer.

The Units property specifies the length of the base clock tick and is used as a scaling factor to scale the output to seconds such that:

timestamp*Scale*Units=time in seconds

The *DC* property is replaced by Shift. For the present, Shift should be set to zero (the default) so that times will be expressed relative to the start of sampling. A value of -10 in Shift will return times relative to 10s so timestamps before 10s will be negative.

Suppose timestamps are stored as 32-bit unsigned integers from a counter clocked at 25microsecond intervals. Then set

```
hdr.tim.Scale=25;
hdr.tim.Units=1e-6;
hdr.tim.Shift=0;
```

Episodic and framed data

For episodic and framed waveform data, each epoch is represented by a column in the data.adc field. You need to

1. create the matrix in data.adc
2. add the number of data points in each epoch to hdr.adc.Npoints as a row vector e.g.
3. place the beginning and end times for each frame in data.tim together with the optional trigger time. Each row of data.tim contains the times for one epoch in data.adc.

Edge and pulse data

Set up data.tim and hdr.tim as above. Set data.adc and hdr.adc empty:

```
data.adc=[];
```

```
hdr.adc=[];
```

Multiplexed data

Multiplexed data are represented as above as a column, or 2-D matrix in data.adc with the values for each subchannel interleaved in the columns.

Then

1. Set hdr.adc.Npoints to (Number of SubChannels x Number of Samples per SubChannel) for each epoch (as a row vector).
2. Set hdr.adc.Multiplex to the number of subchannels
3. Set hdr.adc.MultiInterval= to a two-element row vector containing the interval between samples on successive subchannels using the same format as with hdr.adc.SampleInterval above. You may set this to [0 0] by default as standard sigTOOL functions ignore this value and assume simultaneous sampling on all subchannels.

Adding markers

Markers are added simply by placing them as a column vector or matrix in data.mrk. Each row gives the marker values for each data epoch. The class of the marker data needs to be declared in in hdr.markerclass e.g.

```
data.mrk=uint8(zeros, 100, 4);
hdr.markerclass='uint8';
```

This assigns 4 markers, all of which are zero for each of 100 data epochs. It is good practice to assign markers for episodic data even if they are unused as here, because it will reserve space in the data file to set them later.

Methods associated with scchannel objects

Data in scchannel objects can be accessed using standard MATLAB dot notation e.g.

```
channels{1}.adc(1:10000);
channels{1}.tim(1,:);
```

In addition, standard methods such as get and set can be used. A number of custom methods have also been defined. These fall into several groups:

conv	methods are used to convert between time, matrix indices and subscripts
get	methods return data
find	methods are similar to get methods but have constraints on their inputs which means that the programmer needs to test the contents of the scchannel object, perhaps through a get method, to pass appropriate inputs to the find method
is	methods test the contents of the scchannel object returning a true/false flag
extract	methods return blocks of data from the scchannel object
others	some other methods are provided, e.g. inspect can be used to convert the scchannel object to a structure that can be viewed in the MATLAB array editor.

Code for many of these custom methods have been profiled and optimized. In subsequent releases of sigTOOL, some of the more time consuming code in these methods may be implemented through a mex-file to provide further speed enhancements.

As an example, the code below shows how and average of a Waveform channel on channel 1 might be constructed using the triggers on an event channel (channel 2 in this case). Assume for the moment that `duration` and `pretime` have already been set and we want to use all valid triggers from channel 2.

Step 1:
Retrieve the triggers:

```
      trig=getValidTriggers(channels{2}, 0, Inf);
```

Note the time range from 0 to infinity – this is a get method so there are few constraints on the input.

Step 2:
Extract the data from the waveform channel

```
[data tb epochs]=…
      extractValidFrames(channels{1}, trig, duration, pretime);
```

The rows of `data` contain the relevant waveform data at each of the times relative to the triggers stored in the timebase `tb` (so we need to average the columns). The numbers of the data epochs that the data were drawn from are returned in `epochs`.

Step 3:
Average the data using the MATLAB built-in mean function

```
average=mean(data);% Take the average of each column
```

Step 4.
Plot the result, remembering to convert `tb` to seconds using the factor in `channels{1}.tim.Units`:

```
figure;
plot(tb*channels{1}.tim.Units, average);
```

How should we set up `duration` and `pretime`? Ideally, we would like these to be user settable in seconds. That means converting to the same time units as the channel data in software.

```
duration=duration*(1/channels{1}.tim.Units);
pretime=pretime*(1/channels{1}.tim.Units);
```

[Note the use of 1/channels{1}.tim.Units: Units will often have an integer exponent e.g. 10^{-6}. $1/10^{-6}$ is 10^{6} exactly in IEEE, i.e. a flint. We may still end up with a rounding error depending on the value of `duration` or `pretime`, but we have avoided them for integer values of `duration` and `pretime`].

Finally we need to check that the values are valid if we have an episodic waveform. Is there enough data in each epoch to return given these settings?

```
duration=min(duration, findMaxPostTime(channels{1}, trig));
```

```
pretime=min(pretime, findMaxPreTime(channels{1}, trig));
```
will restrict `duration` and `pretime` to valid values.

For further details see the function wvAverage which uses code similar to that above and also places the result in a sigTOOLResultData object.

Alphabetical list of methods

Note that, except where stated, times specified as input to, or output from, these methods are in base clock ticks for the channels i.e. multiples on *channel.tim.Units*.

convIndex2Time

convIndex2Time converts array indices to times

Examples:
Using indices
```
time=convIndex2Time(channel, SampleNumber)
```
or using subscripts:
```
time=convIndex2Time(channel, Epoch, SampleNumberWithinEpoch)
```
where:
channel is a sigTOOL channel object

When SampleNumber is supplied, this is the 1-D index into the channel adc field (adc need not be 1-dimensional so this method can be used regardless of the dimensions of the adc matrix).

Alternatively, subscripts may be used where Epoch and SampleNumberWithinEpoch are the subscripts of the element that the sample time is required for e.g. convIndex2Time(channel, 10, 8) returns the time of the 8th sample in epoch 10. This is limited to vectors and 2-D matrices.

To convert between subscripts and indices use ind2sub and sub2ind

See also convTime2ValidIndex, ind2sub, sub2ind

SampleNumber, Epoch and SampleNumberWithinEpoch may be column vectors with multiple indices. In that case time will a vector of sample times.

convTime2PhysicalEpochs

convTime2PhysicalEpochs returns physical epoch numbers within a time range

Example:
```
epochs=convTime2PhysicalEpochs(channel, start, stop)
```

where

channel is a sigTOOL channel object
start & stop are the beginning and end times

Returns physical epoch numbers where
start <= channel.tim(:, 1) <= stop

convTime2PhysicalIndex

convTime2PhysicalIndex converts time to linear indices into a waveform matrix

Example:
```
idx=convTime2PhysicalIndex(channel, time)
idx=convTime2PhysicalIndex(channel, start, stop)
```
channel is a sigTOOL channel array cell element
where:

time or start & stop are the times to convert

idx contains the start and stop indices in columns 1 and 2 respectively

When a single time is specified, idx is the index into adc for the sample at the specified time or the first sample afterwards
When start and stop are given, idx are indices into the adc field such that sampling occurred between the limits
start <= t < stop.

With episodic or framed waveforms, idx will be a matrix of indices with one row for each period contained in the interval start to stop.
e.g.
```
idx=convTime2PhysicalIndex(channels{1}, 0, 500000);
```
might return
```
idx =
1 16001
16002 32002
32003 48003
48004 64004
64005 80005
```
Access the first period with:
```
data=channels{1}.adc(idx(1,1):idx(1,2));
```

convTime2PhysicalIndex differs from findVectorIndices in that the returned indices are linear indices into the adc matrix, not indices into the column vector representing a specific epoch

See also ind2sub, sub2ind

convTime2ValidEpochs

convTime2ValidEpochs returns valid epoch numbers within a time range

Example:
`epochs=convTime2ValidEpochs(chan, start, stop)`
where
chan is a sigTOOL channel object
start & stop are the beginning and end times for the search.

Returns valid epoch numbers where
start <= chan.tim(:, 1) < stop

convTime2ValidIndex

convTime2ValidIndex converts time to linear indices into a waveform matrix

Only indices for valid data epochs will be returned

Example:
```
idx=convTime2ValidIndex(channel, time)
idx=convTime2ValidIndex(channel, start, stop)
```
channel is a sigTOOL channel object
time or start & stop are the times to convert

idx contains the start and stop indices in columns 1 and 2 respectively

When a single time is specified, idx is the index into adc for the sample
at the specified time or the first sample afterwards
When start and stop are given, idx are indices into the adc field such
that sampling occurred between the limits
start <= t < stop.

With episodic or framed waveforms, idx will be a matrix of indices with
one row for each period contained in the interval start to stop.
e.g. if channels{1} contains episodes of length 16001,
idx=convTime2ValidIndex(channels{1}, 0, 5);
might return
idx =
1 16001
16002 32002
32003 48003
48004 64004
64005 80005
Access the first period with:
```
data=channels{1}.adc(idx(1,1):idx(1,2));
```

convTime2ValidIndex differs from findVectorIndices in that:
1. It returns indices only for valid epochs
2. It returns indices that are linear indices into the adc matrix,
not indices into the vector representing a specific epoch.

See also findVectorIndices, ind2sub, sub2ind

display

display method overloaded for the scchannel class

extractPhysicalEpochData

extractPhysicalEpochData returns the adc data in an episodically sampled scchannel object

```
[data npoints epochs]=…
                    extractPhysicalEpochData(channel, epoch)
[data npoints epochs]=…
                    extractPhysicalEpochData(channel, epoch1, epoch2)
[data npoints epochs]=…
                    extractPhysicalEpochData(channel, epoch1, step, epoch2)
```

Inputs:
channel is a scchannel object containing episodic data
epoch is the required epoch (e.g. 1, 2) or range of epochs (e.g. 1:10)
Note that the 'end' statement can not be used with this form
epoch1 and epoch2 allow the use of the end statement but it
must be included as a string e.g.
getEpochData(channel, 2, 'end') returns epochs 2:end
where 'end' refers to the last valid epoch.
step if specified sets the increment e.g.
getPhysicalEpochData(channel, 2, 2, 'end')
returns epochs 2:2:end where 'end' refers to the last
valid epoch as above.

Ouputs:
data contains the scaled adc data in double precision with each epoch
represented in columns
npoints and epochs are optional outputs. Each is a row vector.
npoints gives the number of valid data points in each column
epochs gives the physical epoch number of the returned data.

extractPhysicalEpochTimes
extractPhysicalEpochTimes returns the tim data in a channel object

[data epochs]=extractPhysicalEpochTimes(channel, epoch)
[data epochs]=extractPhysicalEpochTimes(channel, epoch1, epoch2)
[data epochs]=extractPhysicalEpochTimes(channel, epoch1, step, epoch2)

extractPhysicalFrames

extracts framed adc data from valid epochs

Example:

```
 [data tb epochs trig]=…
              extractPhysicalFrames(channel, trig, duration, pretime)
```

where

channel is a sigTOOL channel object
trigger is a vector of trigger time
duration is the duration of the sweeep
pretime is the pre-trigger time

All times are in units defined by getTimeUnits(channel) [usually seconds]

Returns

data a double matrix. Each coloumn is a frame of data
tb the timebase for each frame of data (pretime to duration-pretime) in seconds
epochs the physical numbers of the epochs from which data was taken for each frame
trig an updated copy of the input, with invalid trigger times omitted

Note that, in the case of multiplexed channels, extractValidFrames returns data for the currently selected subchannel as set in channel.CurrentSubchannel.

extractValidEpochData

extractValidEpochData returns the adc data in an episodically sampled scchannel object

```
[data npoints epochs]=extractValidEpochData(channel, epoch)
[data npoints epochs]=extractValidEpochData(channel, epoch1, epoch2)
[data npoints epochs]=…
              extractValidEpochData(channel, epoch1, step, epoch2)
```

If EventFilter.Mode is 'on' the specified epoch numbers will be translated to valid epochs for which EventFilter.Flag==true.
Thus with EventFilter.Flags=[0 1 0 1 0 1 0 1 0 1], passing epochs 1:3 on input would return data from the first 3 valid epochs i.e 2,4 and 6.

Inputs:
channel is a scchannel object containing episodic data
epoch is the required epoch (e.g. 1, 2) or range of epochs (e.g. 1:10)

Note that the 'end' statement can not be used with this form
epoch1 and epoch2 allow the the use of the end statement but it must be included as a string e.g.
getEpochData(channel, 2, 'end') returns epochs 2:end
where end refers to the last valid epoch.
step if specified sets the increment e.g.
getEpochData(channel, 2, 2, 'end') returns epochs 2:2:end
where 'end' refers to the last valid epoch as above.

Ouputs:
data contains the scaled adc data in double precision with each epoch represented in columns
npoints and epochs are optional outputs. Each is a row vector.
npoints gives the number of valid data points in each column
epochs gives the physical epoch number of the returned data.

extractValidEpochTimes

extractValidEpochTimes returns the tim data in a channel object

```
[data epochs]= extractValidEpochTimes(channel, epoch)
[data epochs]= extractValidEpochTimes(channel, epoch1, epoch2)
[data epochs]= extractValidEpochTimes(channel, epoch1, step, epoch2)
```

extractValidFrames

extractValidFrames extracts framed adc data from valid epochs

Example:
```
[data tb epochs trig]=…
     extractValidFrames(channel, trig, duration, pretime)
```
where
channel is a sigTOOL channel object
trigger is a vector of trigger time
duration is the duration of the sweeep
pretime is the pre-trigger time
All times are in units defined by getTimeUnits(channel) [usually seconds]

Returns
data a double matrix. Each coloumn is a frame of data
tb the timebase for each frame of data (pretime to duration-pretime) in seconds
epochs the physical numbers of the epochs from which data was taken for each frame
trig an updated copy of the input, with invalid trigger times omitted

Note that, in the case of multiplexed channels, extractValidFrames returns data for the currently selected subchannel as set in channel.CurrentSubchannel.

See also scchannel/getTimeUnits

findMaxPostTime
findMaxPostTime scchannel method

findMaxPostTime returns the maximum post-trigger time for which data is available in all epochs given a set of trigger times

Example:
```
duration=findMaxPostTime(chan, trig)
```

chan is an scchannel object
trig is a time or vector of times

duration will be the maximum available post-trigger time based on the times in trig. This is the minimum of the times available from all epochs

findMaxPreTime

findMaxPreTime returns the maximum pre-trigger time for which data is available in all epochs given a set of trigger times

Example:
```
duration=findMaxPreTime(channel, trigers)
```

channel is an scchannel object
triggers is a time or vector of times

duration will be the maximum available pre-trigger time based on the times in triggers. This is the minimum of the times available from all epochs

findPhysicalEpochs

findPhysicalEpochs returns the physical epochs that a time falls within

Example:
```
epochs=findPhysicalEpochs(chanel, time);
```
where
channel is a sigTOOL channel object
time is a scalar or vector of timestamps
and

epochs is a size(time) vector, containing the relevant epoch for each timestamp in time. If a timestamp does not fall within an epoch, epochs will contain zero. A +/- 1 sample interval jitter is allowed to account for sequentially sampled waveform channels where samples are not simultaneous.

findValidFrameIndices

returns the indices of valid frames

Example:

```
[idx epochs trigger]=…
    findValidFrameIndices(channel, trigger, duration, pretime)
```

where

channel is a sigTOOL channels object
trigger is a vector of trigger times
duration is the duration of the sweep
pre-time is the pre-trigger time

All times are in the same units (as returned by getTimeUnits(channel))

Returns

idx a 2-column vector with the start and end row indices for each trigger
epochs the epochs (columns) that idx refers to
trigger an updated copy of the input, with invalid trigger times omitted

findValidEpochs

findValidEpochs returns the valid epochs that a time falls within

Example:

```
epochs=findValidEpochs(chanel, time);
```

where
channel is a sigTOOL channel object
time is a scalar or vector of timestamps
and
epochs is a size(time) vector, containing the relevant epoch for each timestamp in time. If a timestamp does not fall within an epoch, epochs will contain zero

findValidFrameIndices

findValidFrameIndices returns the indices of valid frames

Example:

```
[idx epochs trigger]=…
    findValidFrameIndices(channel, trigger, duration, pretime)
```

where
channel is a sigTOOL channels object

trigger is a vector of trigger times
duration is the duration of the sweep
pre-time is the pre-trigger time
All times are in the same units (as returned by getTimeUnits(channel))

Returns
idx a 2-column vector with the start and end row indices for each trigger
epochs the epochs (columns) that idx refers to
trigger an updated copy of the input, with invalid trigger times omitted

findVectorIndices

findVectorIndices converts time to the indices into a waveform vector

Example:
```
[n1 n2 epoch]=findVectorIndices(channel, start, stop)
```
matrix=findVectorIndices(channel, start, stop)

where:
channel is a sigTOOL channel object
start is the time of the first sample and must fall within a valid data epoch for the channel or an error will result
stop is valid the time to search to for valid data

n1 and n2 are the indices into the vector (epoch) corresponding to the times:
start <= t < stop
for continuous waveforms or
start <= t <= stop
for episodic waveforms and will be limited to (n1 >= 1) and (n2<=epoch length).

start and stop may be vectors, in which case n1, n2 and epoch will be vectors with one entry for each of the specified data periods.

If only one output is requested, this will be a 3-column matrix containing n1, n2 and epoch in each row.

Continuous waveforms

If channel contains a continuous waveform (i.e. a single vector of adc data), n1 and n2 are simply linear indices into the the vector and epoch

will always be equal to 1.
n1 and n2 will always be aligned on subchannel of multiplexed data.

Episodic sampled waveforms

Epoch and n1, n2 can be used for subscripted indexing into the adc field.
n1 and n2 give the rows and epoch the columns. Thus, n1 and n2 are the indices into the column vector of data representing the epoch e.g.
[n1 n2 epoch]=findVectorIndices(channels{1}, 0.2, 0.3)
The relevant adc data may be extracted with:
data=channels{1}.adc(n1:n2, epoch)

The exact sample times may be retrieved using convIndex2Time e.g
t1=convIndex2Time(channels{1}, n1, epoch)
With episodically sampled multiplexed data, n1 will always be aligned on subchannel 1. n2 will be aligned on subchannel 1 unless stop exceeds the epoch time in which case n2 will be aligned on the highest numbered subchannel and will be limited to the length of the data. This makes data extraction simpler, e.g. to extract subchannel 2 of 4:
[n1 n2 epochs]=findVectorIndices(channels{1}, 0.2, 0.3);
data=channels{1}.adc(n1+1:4:n2, epochs)

See also convIndex2Time, ind2sub, sub2ind

get

get method for overloaded for the scchannel class

getData

getData returns the data for a specified time period

Data are returned for the period START <= t < STOP

Example:
```
channelout=getData(channel, start, stop)
channelout=getData(channel, [start stop])
```

channel is an scchannel object
start and stop are the times marking the beginning and end of the required time window.

channelout on output is an scchannel object.
Waveform data in channelout will be trimmed to the limits START <= t < STOP.
When the channel is a 'Custom' channeltype, all epochs with

START <= tim(:,1) < STOP will be returned.
The epochs are defined by the highest dimension of adc.

getPhase

getPhase returns the phase of an event during a cycle

Examples:

```
phase=getPhase(trigchannel, eventchannel)
phase=getPhase(trigchannel, eventchannel, start, stop)
```

returns the phase of the events in eventchannel in relation to the cycles defined by the events in trigchannel

If defined, start and stop give the time period to use. Otherwise these default to start=0 and stop=Inf.

The output, phase, is a double precision vector. For each element, the fractional part represents the phase while the non-fractional part represents the number of the valid cycle that each event occurred in. Thus phase=2.75 indicates that an event occurred 3/4 of the way through the second valid cycle in the period start to stop.

Only only valid events in eventchannel will be used. For triggers, only valid events will be used to mark the start of a cycle but all physical events will be searched for the end of cycle marker (i.e for the start of the subsequent cycle). This is useful when there are breaks in the data e.g. if there are 10 cycles but the 10th was interrupted, mark 1-9 as valid. Only these will be used for triggers but the onset of the interupted 10th cycle will be used to determine the length of the 9th cycle for calculating phase correctly.

getPhysicalTriggers

getPhysicalTriggers returns all trigger times over a time period

Example:

```
trig=getPhysicalTriggers(channel, start, stop)
```

where
channel is an scchannel object
start is the start time for the search
stop is the stop time for the search

trig is a vector of trigger time

Returns all triggers where
start <= channel.tim(:, 1) <= stop

Trigger times are those in column 1 of channel.tim if it has 1 or 2 columns or those in column 2 if it has 3 columns (i.e. an explicit trigger time is present

getSampleInterval

getSampleInterval returns the sampling interval in a scchannel object

Example:
```
interval=getSampleInterval(channel)
```
interval is returned in seconds

getSampleRate

getSampleRate returns the sample rate in a scchannel object

Example:
```
Fs=getSampleRate(channel)
```
Fs is returned as samples/second

getTimeUnits

getTimeUnits returns the units used to represent time in the tim field of an scchannel object as a string

Example:
```
str=getTimeUnits(channel)
```

getTimeVector

getTimeVector generates a vector or matrix of sample times for a waveform channel

Example:
```
t=getTimeVector(channel)
```

where:

t the output matrix containing the timebase, one value if t for each sample in the waveform channel

channel a waveform channel as an scchannel object

:

getValidEpochNumbers

getValidEpochNumbers returns the physical numbers for valid epochs

```
epochs=getValidEpochNumbers(channel)
```
returns all valid epoch numbers
```
epochs=getValidEpochNumbers(channel, n)
```

returns the nth valid epoch number
`epochs=getValidEpochNumbers(channel, n1, n2)`
returns the n1th through n2th valid epoch numbers
`epochs=getValidEpochNumbers(channel, n1, step, n2)`
returns every step valid epoch between the n1th and n2th
valid epoch numbers

getValidTriggers

getValidTriggers returns all valid trigger times over a time period

Example:
`trig=getValidTriggers(channel, start, stop)`
where
channel is an scchannel object
start is the start time for the search
stop is the stop time for the search

trig is a vector of trigger time

Returns triggers for all valid events/epoochs where
start <= channel.tim(:, 1) <= stop

Trigger times are those in column 1 of channel.tim if it has 1 or 2 columns or those in column 2 if it has 3 columns (i.e. an explicit trigger time is present

inspect

inspect method for scchannel objects

Examples:
`s=inspect(obj)`
returns a structure that can then be inspected using the MATLAB array editor. Custom defined objects in each field of obj are also cast to structures (and the fieldname changed to indicate this). memmapfile objects remain as objects and are not editable within the array editor
inspect(obj)
places the structure in 'ans' in the base workspace and opens it in the array editor

isInSynch

isInSynch method for scchannel objects

Example
`TF=isInSynch(chan1, chan2)`
where chan1 and chan2 are scchanel objects

isInSynch returns true if the adc data in the channels share the same sampling rate and the beginning and end of each epoch are within 1 sample interval of each other

isMultiplexed

isMultiplexed returns true if any of the channels contain multiplexed adc data

Example:

```
TF=isMultiplexed(chan1, chan2,...);
```

isSwapNeeded

isSwapNeeded method for scchannel objects

Example

```
TF=isSwapNeeded(obj)
```

returns true if data in the Map.Data.Adc need to be byte swapped on the current platform, false otherwise

isSwapNeeded should rarely be needed. If data are accessed through the scchannel, adcarray or tstamp subsref methods, byte swapping will be done automatically as required. Only, if you extract the memmapfile object (e.g. using get) will you need to know the byte order.

plot

plot method overloaded for scchannel class

scchannel

scchannel constructor for sigTOOL channel object

Example:

```
obj=scchannel(s)
```

returns an scschannel object given a sigTOOL channel structure as input

size

size method for overloaded for the scchannel class

subsasgn

subsasgn method for overloaded for the scchannel class

subsref

subsref method for overloaded for the scchannel class

Building graphical user interfaces

A menu_ function that is added to the sigTOOL menu as described above will typically display a graphical user interface to prompt for user input. There are several ways to design these interfaces including:

1. Using GUIDE, the MATLAB GUI development environment.
   For an example of interfacing MATLAB to a GUIDE developed GUI see the menu_Interface_To_Waveclus function
2. Manually writing code calling MATLAB's uicontrols
3. Using the sigTOOL GUI development functions. These provide low-, medium- and high-level functions that give access to Java Swing GUI components in MATLAB.

Low level GUI tools: the jcontrol object

A jcontrol is a custom designed sigTOOL object that provides access to the Java AWT and Swing graphical components. A jcontrol object is associated with:

1. A MATLAB container. This is a MATLAB handle graphics object that contains the Java component
2. The Java component
3. A MATLAB handle that points to the MATLAB container.

The MATLAB container has a MATLAB figure as an ancestor. The Java component it contains must therefore be lower in the hierarchy than a Java frame.

With jcontrol objects, you can gain access to all the Java component's callbacks rather than just a few of them as with MATLAB uicontrols (which are also Java components underneath). The medium- and high- level routines described below make use of jcontrol objects. Most users will be able to rely on these routines rather than creating jcontrols directly – the following description of jcontrol objects can therefore be skipped.

Creating jcontrol objects

Note: The jcontrol class methods have been modified for compatibility with MATLAB R2008b onwards. See the jcontrol help at the ML command window for full details. Jcontrol objects now always have a MATLAB uipanel as an ancestor.

```
Example:
obj=JCONTROL(Parent, Style);
obj=JCONTROL(Parent, Style, PropertyName1, PropertyValue1,...
             PropertyName2, ProeprtyValue2....);
Inputs:
Parent: the handle of a Matlab figure or other container for the resulting
     component
Style: string describing a java component e.g. 'javax.swing.JPanel',
     'javax.swing.JButton' or a variable containing a java object
PropertName/PropertyValue pairs: these are automatically assigned to the
     HG container or the java component as appropriate.

Pre-create the java object if you need to pass arguments to the
constructor e.g.
javaobj=javax.swing...(...............);
obj=jcontrol(Parent, javaobj)

By default, JCONTROLs are returned with Units set to 'normalized'.

USE:
Build a GUI with repeated calls to JCONTROL in much the same way as with
MATLAB's uicontrol function e.g.:
            h=jcontrol(gcf,'javax.swing.JPanel',...
              'Units','pixels',...
              'Position',[100 100 200 200]);
             h(2)=jcontrol(h(1),'javax.swing.JComboBox',...
                      'Position',[0.1 0.8 0.8 0.1]);
             h(2).addItem('Item1');
             h(2).addItem('Item2');
             h(3)=jcontrol(h(1),'javax.swing.JCheckBox',...
                      'Position',[0.1 0.1 0.1 0.1],...
                      'Label','My check box');
See the jcontrolDemo() for a fuller example.

A JCONTROL aggregates the MATLAB handle graphics container and the Java
component (as returned by MATLAB's JAVACOMPONENT function) into a single
object.
Access to the JCONTROL's properties is provided by GET/SET calls.
```

These automatically determine whether the target property is in the HG container or java object.

```
myobject=jcontrol(gcf,'javax.swing.JPanel',...
        'Units', 'normalized',...
        'Name', 'MyPanel');
set(myobject, 'Position', [0.4 0.4 0.4 0.2],...
        'Enabled', 0);
pos=get(myobject,'Units');
```

Note that you can mix HG container properties (e.g. Units, Position) and java component properties (e.g. Name, Enabled) in single calls to JCONTROL and SET.
Use the HG container to control the Units, Position, and Visible properties

MATLAB dot notation may also be used. This notation also provides access to the java object's methods

```
pos=myobject.Position;
sz=myObject.getSize;
myobject.setEnabled(1);
myobject.setToolTipText('My tip');
myobject.setOpaque(1);
```

UNITS, POSITION and VISIBLE properties
Set these by accessing the JCONTROL or its container (not the hgcontrol).
MATLAB links these properties between the container and the java control, but unidirectionally.
Note that JCONTROL methods always act on/return the Visible property of the container ('on' or 'off') which will also update the java control.
Do not use the setVisible() methods.

Overloaded class methods are case-insensitive for properties but case-sensitive for java methods

CALLBACKS
Setting up callbacks
The simplest way to set up a callback is through the SET method

```
myPanel=jcontrol(gcf,'javax.swing.JPanel',...
        'Units','normalized',...
        'Position',[0.3 0.3 0.5 0.5]);
set(myPanel, 'MouseClickedCallback', 'MyCallback')
or
set(myPanel, 'MouseClickedCallback', @MyCallback);
or
set(myPanel ,'MouseClickedCallback', {@MyCallback A B C...});
```

The callback then takes the usual MATLAB form, e.g.

```
function MyCallback(hObject, EventData)
function MyCallback(hObject, EventData, varargin)
```

Accessing JCONTROL objects in callbacks:
The handle received by a callback will be that of the java control object contained in the JCONTROL, not the JCONTROL itself. In addition, GCO will return empty and GCBO will not return the parent figure handle. However, the JCONTROL constructor adds the HG container handle to the java component's properties. This can be used to access the container and its parent figure from within the callback e.g.

```
get(hObject.hghandle);% gets the HG container
ancestor(hObject.hghandle,'figure')% gets the parent figure handle
```

To cross-reference from the container, JCONTROL places a reference to the java control in the container's UserData area e.g.

```
hgc=findobj('Tag','MyCustomTag')
javacontrol=get(hgc, 'UserData');
```

Accessing data in callbacks
Data can be passed to a callback, as above, with optional input arguments. In addition, data that is specific to the control can be stored in the application data area of the control e.g. to return values dependent on the selection of a popup menu

```
data=getappdata(hObject,'data');
returnvalues=data(hObject.getSelectedItem+1);
```

Note: +1 because the item numbering is zero based for the java object.
The HG container has a separate application data area.

R2006a or higher only:
GETAPPDATA, SETAPPDATA ISAPPDATA and RMAPPDATA methods have been overloaded for JCONTROL objects. These place/return data from the application data area of the java control. Take care if removing the whole application data area - TMW may place data in there too. The HG container has a separate application data area.

Notes:
If a property name occurs in both the HG container and the java object, the JCONTROL methods can not unambiguously identify the source/target and it must be defined explicitly by the user e.g.

```
get(myobject.hgcontainer,'Opaque');
set(myobject.hgcontrol, 'Opaque',0);
```

The JCONTROL methods test for ambiguity and issue an error message when it arises. Note that the test uses MATLAB's isprop and is case insensitive.
It may also detect properties not listed by the MATLAB builtin GET

function for the hgcontrol such as Visible. The JCONTROL methods always act on the Visible property of the hgcontainer, letting MATLAB update the object automatically (see above).

The DeleteFcn property of the hgcontainer is set by the JAVACOMPONENT function. If this property is changed, the new callback must explicitly delete the hgcontrol.

Middle-level GUI functions

Middle and high- level GUI functions in sigTOOL all have names beginning with jv. Quite complex GUIs can be developed pretty easily using these functions using the following steps:

Step 1: Create a panel structure

To create a GUI panel call jvPanel which returns a structure:

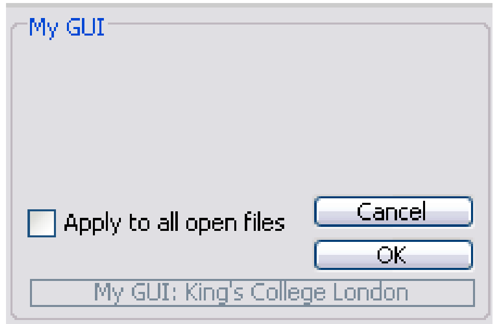
```
>> s=jvPanel('Title', 'My GUI',...
'Position', [0.4 0.4 0.2 0.2],...
'ToolTipText', 'This is My new GUI',...
'AckText','My GUI: King''s College London');

>> s
s =
      Panel: [1x1 struct]
    AckText: 'My GUI: King's College London'
         OK: []
     Cancel: []
>>
```

We have not added anything useful to the GUI yet but can nevertheless display it by calling jvDisplay, passing the structure s as an input:

```
>> fhandle=figure();
>> h=jvDisplay(fh,s);
```

This displays the panel in the figure:



Note that jvDisplay has added some standard features: OK and Cancel buttons together with the Apply to all open files checkbox. Standard callbacks have also been activated for these components by the jvDisplay function.

jvDisplay returns a cell containing a structure in h. To examine this:

```
>> h{1}
ans =
         Panel: [1x1 jcontrol]
       AckText: [1x1 jcontrol]
    ApplyToAll: [1x1 jcontrol]
            OK: [1x1 jcontrol]
        Cancel: [1x1 jcontrol]
          Help: [1x1 jcontrol]
```

jcontrol objects have been created for the panel itself and for each of the component controls that it contains. You can access these at a low-level through their properties/methods as described above e.g.

```
>> get(h{1}.Panel)
ans =
    hgcontainer: [1x1 hgjavacomponent]
      hgcontrol: [1x1 javahandle_withcallbacks.javax.swing.JPanel]
       hghandle: 390.0012
```

Ignore the Help jcontrol for now.

Step 2: Adding Useful elements

The GUI above does not ask anything useful. To create a real GUI, you need to add further elements to the structure s returned by jvPanel. To add a channel selector:

```
>> s=jvPanel('Title', 'My GUI',...
'Position', [0.4 0.4 0.2 0.2],...
'ToolTipText', 'This is My new GUI',...
```

```
'AckText','My GUI: King''s College London');

>> s=jvElement(s, 'Component', 'channelselector',...
'Label', 'Channel A' ,...
'Position', [0.1 0.7 0.8 0.1],...
'DisplayList', {'Channel 1', 'Channel 2'}, ...
'ReturnValues', {1, 2});
```

The jvElement function creates the channel selector. Note that the DisplayList contains the strings to display and ReturnValues contains, in this case, numeric values corresponding to each string.

At the time of writing, the following Java Swing items are supported via the jvElement function:

javax.swing.JComboBox
javax.swing.JList
javax.swing.JCheckBox
javax.swing.JButton
javax.swing.JPanel
javax.swing.JTextField

In addition, sigTOOL provides two purpose-designed JComboBoxes
channelselector
timermenu

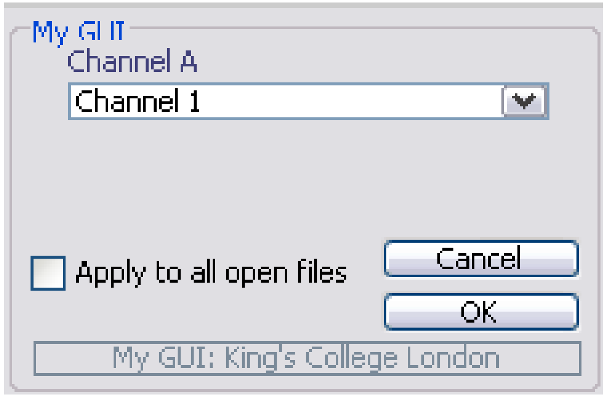
ChannelSelector was described above. A TimerMenu provides the Start and Stop times for an analysis adding support for cursors etc, automatically.

Step 3: Display the GUI

Note here that channelselector is a sigTOOL defined control type based on the javax.swing.JComboBox. To look at the new GUI:

```
h=jvDisplay(fhandle, s);
```

displays the following



If we look again at h{1} returned by jvDisplay we will find that the handles for the channel selector (and its label) are now included:

```
>> h{1}
ans =
            Panel: [1x1 jcontrol]
         ChannelA: [1x1 jcontrol]
    ChannelALabel: [1x1 jcontrol]
          AckText: [1x1 jcontrol]
       ApplyToAll: [1x1 jcontrol]
               OK: [1x1 jcontrol]
           Cancel: [1x1 jcontrol]
             Help: [1x1 jcontrol]
```

Note that the name of the field in h{1} is the same as the label string but without any spaces. We could also add text to the label in parentheses. That text will not contribute to the field name in h, so characters that are illegal in field names can still be used in the label as long as they are in parentheses.

Step 4: Getting the user selections

To make use if the GUI, we need to halt code execution until the user clicks OK or Cancel. Do this using MATLAB's standard uiwait() command:

```
h=jvDisplay(fhandle, s);
uiwait();
```

The OK or Cancel button callbacks will issue the needed uiresume() to cause the calling routine to start running again. We still need to find what options the user had selected when OK was pressed. The OK button callback takes care of this, placing the results in the application data area of the figure. Note that values are returned empty for those controls that have no meaningful data associated with them.

```
>> getappdata(2,'sigTOOLjvvalues')
ans =
            Panel: []
         ChannelA: 1
    ChannelALabel: []
          AckText: []
       ApplyToAll: 0
               OK: []
           Cancel: []
             Help: []
```

High-level GUI functions

In most instances, you will need to call only two higher level GUI functions: jvDefaultPanel and jvAddPanel.

- **jvDefaultPanel**

Take the following example from the menu_Average function in the Waveform processing menu:

```
h=jvDefaultPanel(fhandle, 'Title', ‘Waveform
Average’,...
    'ChannelType', {'All' 'Waveform'},...
    'ChannelLabels', {'Trigger' 'Waveforms'});
```

fhandle is the figure handle for the sigTOOL data view

We add a title, in this case ‘Waveform Average’ and define the channel types to be displayed in the Channel A and Channel B selectors. The channel types will be passed to the scGetChannelsByType function which will return a list of relevant channels. Supported strings (at the time of writing) are:

All
Empty
Episodic
Multiplexed
Triggered

and

None

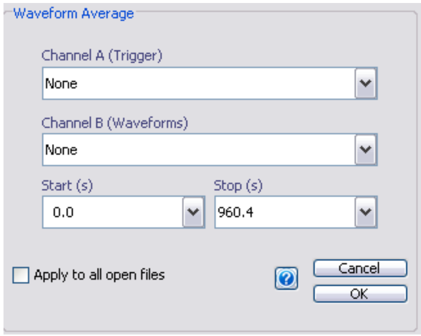
Alternatively, you can provide any string to use as a pattern that will be searched for in the channel type description in the channel header, using:

```
strfind(channels{i}.hdr.channeltype, pattern);
```

For details of channel types in sigTOOL see Representing channel data in sigTOOL

The channel labels are the strings added in brackets after the Channel A and B text.

The resulting default GUI panel is shown below. Note that default options for Start, Stop, Cancel, OK and Apply to all open files have been added automatically.

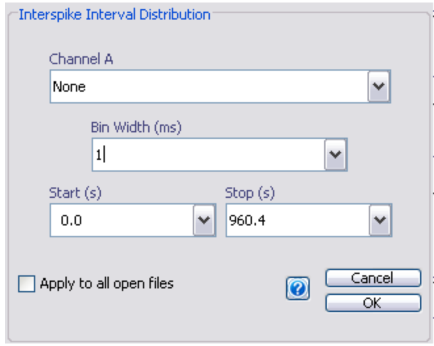


The cell h returned by the call to jvDefaultPanel contains the jcontrols associated with each component. In many instances, you will be able to customize this menu by altering the underlying jcontrols using the low-level function described above. Here is an example from the menu_InterspikeInterval function:

```
h=jvDefaultPanel(fhandle, 'Title', 'Interspike Interval Distribution',...
    'ChannelType', {'All' 'none'});

% Re-use the Channel B list for the bin width selection
h{1}.ChannelBLabel.setText('Bin Width (ms)');
h{1}.ChannelB.removeAllItems;
h{1}.ChannelB.addItem('1');
h{1}.ChannelB.addItem('2');
h{1}.ChannelB.addItem('5');
h{1}.ChannelB.addItem('10');
h{1}.ChannelB.setEnabled(true);
h{1}.ChannelB.Position(1)=h{1}.ChannelB.Position(1)+0.1;
h{1}.ChannelB.Position(3)=h{1}.ChannelB.Position(3)-0.2;
h{1}.ChannelBLabel.Position(1)=h{1}.ChannelBLabel.Position(1)+0.1;
h{1}.ChannelBLabel.Position(3)=h{1}.ChannelBLabel.Position(3)-0.2;
```

This displays:



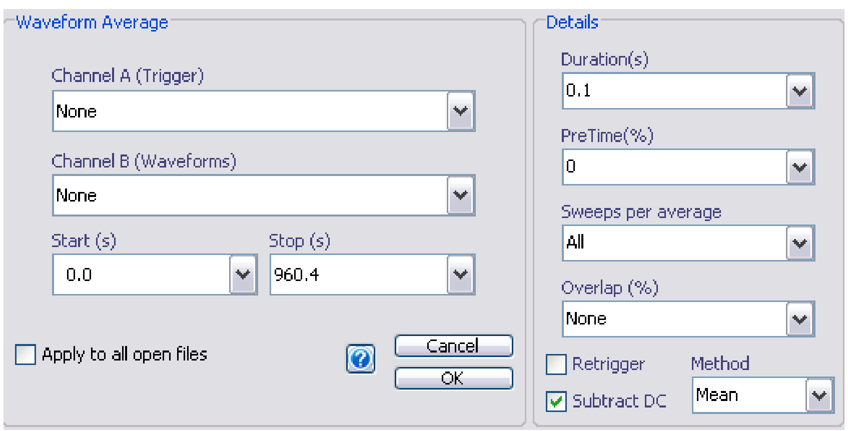
The Channel B selector has been re-used to represent the Bin Width for the analysis (Note: If you want to change the name of that field in the in h{1} you can do so, but you must also save the new handles in the data view sigTOOLjvhandles application data area).

- **jvAddPanel**

Most analysis GUIs present a supplementary panel for entering analysis options. These are added using the jvAddPanel function. For the menu_Average function the panel is added by:

```
h=jvAddPanel(h, 'Title', 'Details',...
    'dimension', 0.6);
```

h on input is typically from a preceding call to jvDefaultPanel. On output, h has a second cell added with details of the new panel. This is populated by calls to jvElement as before. In the case of the averaging function, all this is done in a function called jvAddAverage. This displays:



In this case h{2} contains:

```
K>> h{2}

ans =

                   Panel: [1x1 jcontrol]
                Duration: [1x1 jcontrol]
           DurationLabel: [1x1 jcontrol]
                 PreTime: [1x1 jcontrol]
            PreTimeLabel: [1x1 jcontrol]
        Sweepsperaverage: [1x1 jcontrol]
   SweepsperaverageLabel: [1x1 jcontrol]
                 Overlap: [1x1 jcontrol]
            OverlapLabel: [1x1 jcontrol]
                  Method: [1x1 jcontrol]
             MethodLabel: [1x1 jcontrol]
               Retrigger: [1x1 jcontrol]
              SubtractDC: [1x1 jcontrol]
```

Access to these controls will be as before by calling

```
s=getappdata(fhandle,'sigTOOLjvvalues')
```

but s will now be a cell array with one set of returned values for each element of h e.g.

```
s =

    [1x1 struct]    [1x1 struct]

K>> s{2}

ans =

                   Panel: []
                Duration: 0.1000
           DurationLabel: []
                 PreTime: 0
            PreTimeLabel: []
        Sweepsperaverage: 0
   SweepsperaverageLabel: []
                 Overlap: 0
            OverlapLabel: []
                  Method: 'mean'
             MethodLabel: []
               Retrigger: 0
```

```
SubtractDC: 1
```

The sigTOOL/CORE/utils/uifunctions folder contains jvAddAverage and many similar functions for adding supplementary panels to the other Waveform and Spike processing menus. The quickest way to develop a GUI is to find one of these that does (almost) want you want, copy it to a new file and edit it to meet your needs more exactly.

Linking channel selections

The Channel B selector may be disabled by default and activated only following selection of Channel A. In this case, the channels displayed in the Channel B selector list may be determined by the Channel A selection. To activate this call

```
jvLinkChannelSelectors(h, linkmethod)
```

`after calling jvDisplay.` Linkmethod may be ‘All’, ‘Fs’, ‘Synchro’ or ‘EqualEpochs’ for selecting all channels, those with matched sample rate, those with matched sampling rates and synchronized epochs or the same number of epochs respectively.

Adding help to the menus

The jvSetHelp function activates the help button on the main panel of any menu displayed by jvDisplay. The help files should be written in a markup language (HTML is assumed by default) for display in the web browser. To ensure that the file will be found correctly, place it in a ….\private\help folder where …. represents the folder containing the calling routine. jvSetHelp works relative to the folder of the calling routine so this should ensure that your help file will be found on any computer that the folder is moved to irrespective of absolute path. Using the \private folder also makes sure that the contents of that folder does not appear on the MATLAB path (see the MATLAB help for details of how MATLAB handles private folders). Example:

```
jvSetHelp(h, 'Waveform Average.html');
```

where h is the cell array of handles from a prior call to jvDisplay (or equivalent such as jvDefaultPanel etc.).

sigTOOL result objects

sigTOOL analysis functions return results as sigTOOLResultData objects. Much of the sigTOOL functionality is implemented as methods of these objects: it is therefore desirable that programmers should use this object class when returning results from user–written analysis functions.

sigTOOLResultData objects have the following properties:

acktext

An acknowledgement string. Programmers can add their own text here, which will be included on print-outs of the analysis results

data

A cell array containing the data. This is described further below.

datasource[9]

The handle of the source sigTOOL data view.

datasourcetitle[7]

A string describing the data source – typically the name of the file.

description

A string describing the type of result e.g. 'Waveform Average'. This will be added to the title of result views.

displaymode

A string: this identifies the default display style as set in the sigTOOL Result Manager.

options

A handle to a uicontextmenu. This will be added to the general-purpose uicontextmenu of a sigTOOL result figure. See below for further details.

plotstyle

A handle to a function to use to plot the data.

title

A string: the name to give the result figure

userdata

[9] sigTOOL version 0.91: Use the datasourcetitle field in preference to the datasource field when programming. The handle in datasource is not reliable as the figure may be closed and the handle re-used after creating a result object. For backwards compatability, if datasource is specified on construction of the result object but datasourcetitle is not, sigTOOL will place the name of the datasource figure in the datasourcetitle field.

Typically empty, users can use this property to store custom data/details

viewstyle

No longer used

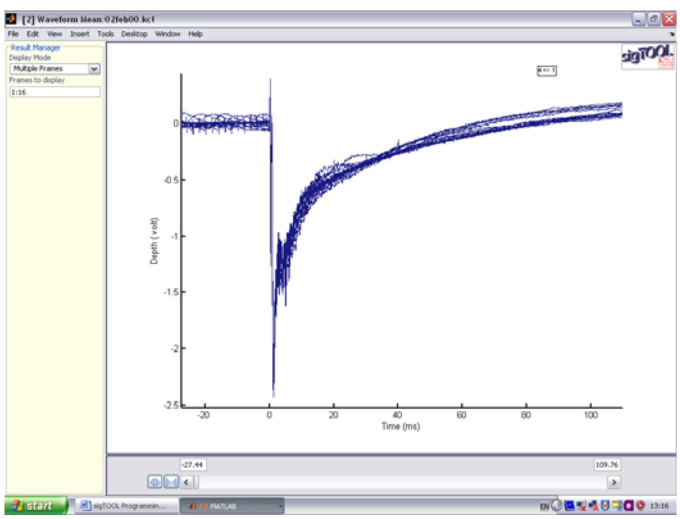
zzID

A unique identifier**.** Not used at present and likely to be removed.

When sigTOOLResultData objects are plotted, the plot routine creates, in addition, a sigTOOLResultView object. This is stored in the result figure application data area. This object class is used to overload figure-related methods such as print and printpreview which have been customized to give best results with sigTOOL result figures e.g.

```
obj=getappdata(resultfigurehandle, ‘sigTOOLResultView’);
printpreview(obj);
```

As an example, take a look at the following set of waveform averages calculated in sigTOOL. Each average is of 20 sweeps and a total of 16 averages have been calculated i.e. a total of 20*16 triggers were used.



The figure title tells us that this figure has a handle of 2, so

```
>> x=getappdata(2, 'sigTOOLResultData')
            acktext: ''
               data: {2x2 cell}
         datasource: 1
    datasourcetitle: 'demo.kcl'
        description: []
            details: []
        displaymode: 'Multiple Frames'
            options: []
```

```
                plotstyle: {[1x1 function_handle]}
                    title: 'Waveform Mean'
                 userdata: []
                viewstyle: '2D'
                     zzID: 'tp28f19552_ec73_4080_bacd_fb80ea5c0351'
```

- **The data field**

The data field is a cell array that contains the data to plot. In this case we have only one graphic in the figure

```
>> x.data
ans =
    'Channel'    '1'
    '4'          [1x1 struct]
```

The first column contains the numbers of the channels that were used as a reference or trigger. The first row contains the numbers of the source data channels. The remaining fields are either empty, contain a standard structure as detailed below, or (from sigTOOL 0.89 onwards) they may contain a custom object.

The standard sigTOOL result structure

In the case illustrated above, there are no empty fields and just one standard data structure

```
>> x.data{2,2}
ans =
      tdata: [1x6861 double]
      rdata: [16x6861 double]
      odata: [1x16 double]
    errdata: [1x1 struct]
    details: [1x1 struct]
     tlabel: 'Time (ms)'
     rlabel: 'Depth ( volt)'
     olabel: 'Time (s)'
```

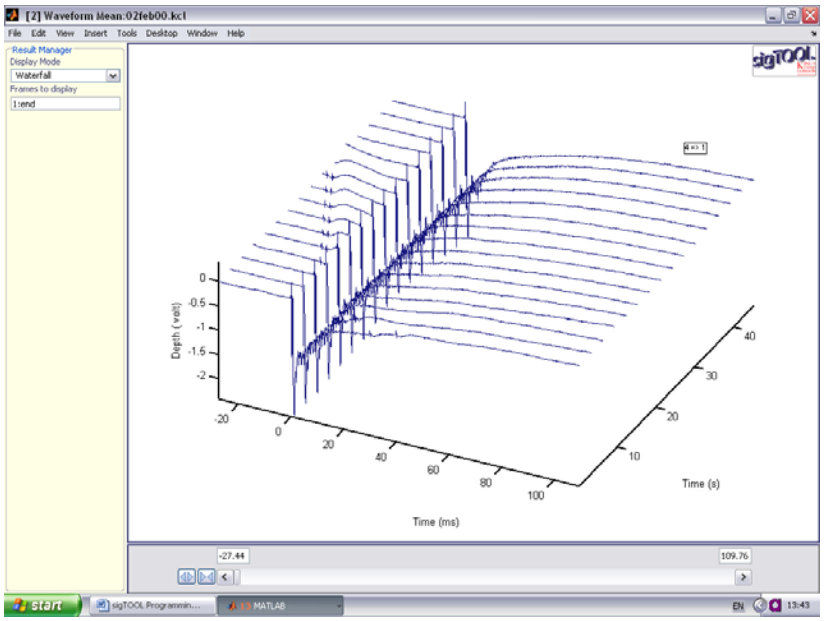
tdata

The tdata field contains the x-axis values for the plot. In this case it is a 6861 element row vector which provides the timebase for the averages. In other cases it could contain the frequencies for a power spectrum.

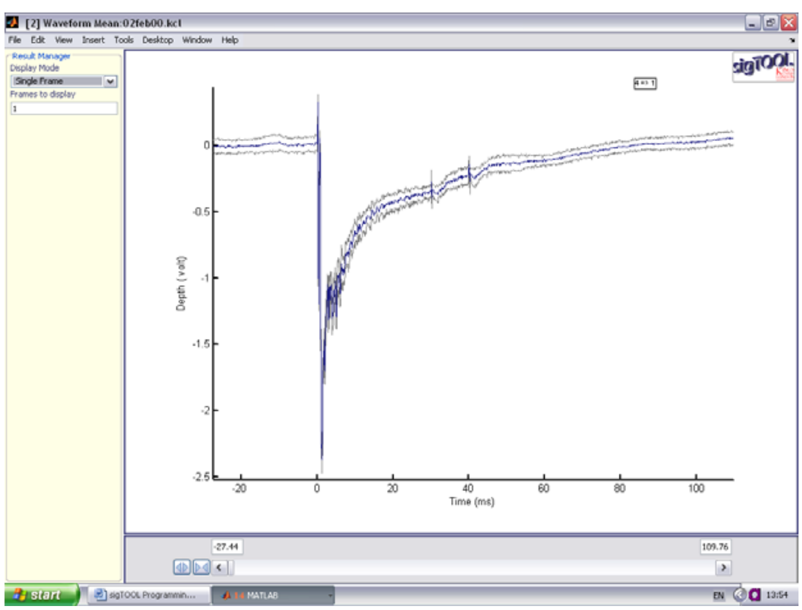
rdata

The rdata field contains the main data for the result. In this case, there are 16 averages each with 6861 data points arranged a row vectors. In 2-D plots as here, these data are plotted on the y-axis. In 3-D plots they will be the z-axis data.

odata
The odata field contains the offsets to be used for the y-axis in 3-D plots (the field is ignored for 2-D plots). In the example case, odata contains 16 values, i.e. one for each average and these are the times of the start of each block of data used to construct the 16 averages. If we swap to a 3-D presentation the odata values are used to offset the data on the y-axis e.g.



errdata
The errdata field contains the data to use for error bars. For clarity, these data will usually only be plotted when we view a single average:



In this case errdata contains standard deviations:

```
>> x.data{2,2}.errdata
ans =
       r: [16x6861 double]
    type: 'std'
```

These are stored in errdata.r which is a matrix of standard deviations with 16 rows, one for each average, and 6861 data points in each row corresponding to each point in the averages. The errdata.type field specifies the nature of the error estimate, in this cast std for standard deviation. Had median averaging been used, the errors would have been percentiles[10]:

```
>> y.data{2,2}.errdata
ans =
       r: [1x1 struct]
    type: 'prctile'
```

In this case errdata.r is a structure with the upper and lower percentiles:

```
>> y.data{2,2}.errdata.r
ans =
    upper: [16x6861 double]
    lower: [16x6861 double]
```

details

The details field contains anything that the programmer chooses. Usually it will list the options used to calculate the result. For the average details contains:

```
>> x.data{2,2}.details
ans =
        frames: {1x16 cell}
       nsweeps: 16
    codesource: 'wvAverage'
        method: @mean
```

This tells us that there are 16 averages, that the wvAverage function was used to calculate them and that the averaging method was the MATLAB built-in mean function, represented here as a function handle. In this case the source data were episodically sampled and the frames field tells us which episodes of data were used to construct each of the 16 averages:

```
>> x.data{2,2}.details.frames{1}'
ans =
  Columns 1 through 12
```

[10] Percentiles require the MATLAB Statistic Toolbox to be present

```
     1     4     7    10    13    16    19    22    25    28
31    34
  Columns 13 through 20
    37    40    43    46    49    52    55    58
```

The triggers fell in epochs 1, 4, 7 etc. Non-overlapping averages were used and the next average was formed from epochs:

```
>> x.data{2,2}.details.frames{2}'
ans =
  Columns 1 through 12
    61    64    67    70    73    76    79    82    85    88
91    94
  Columns 13 through 20
    97   100   103   106   109   112   115   118
```

tlabel, rlabel and olabel

These fields each contain a string that will be used in the result figure to label the axes. In this case ‘Time (ms)’, ‘Depth (volt)’ and ‘Time (s)’.

tdir, rdir and odir

If present, these are set to ‘normal’ or ‘reverse’ and control the direction of the axes.

Custom-defined objects[11]

Support for custom objects was introduced in sigTOOL version 0.89. These objects must be supported by an overloaded plot method specific to their class. The ‘plotstyle’ field in the sigTOOLResultData object should be set to @plot to invoke this method.

The matrix of axes normally present for a standard result view will be replaced by a matrix of uipanels. The plot method for the object will then be invoked to draw the object within a uipanel.

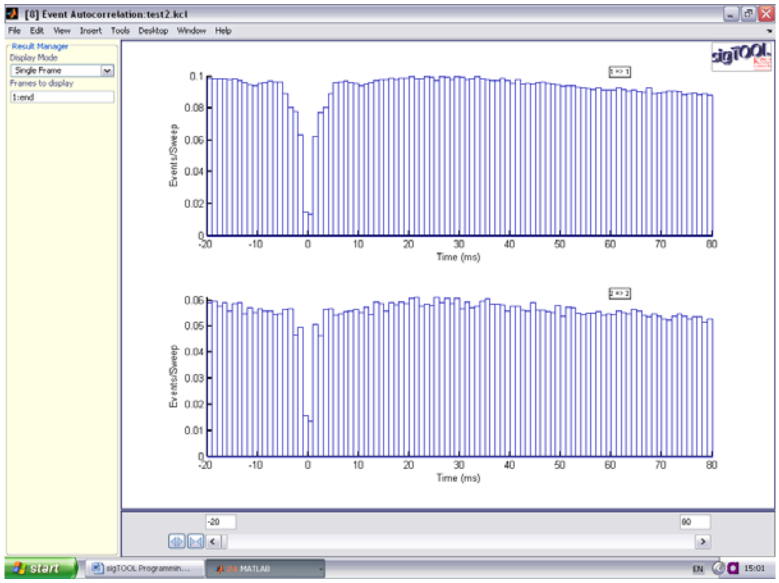
The standard sigTOOL context menu for the axes will be replaced, not supplemented, by the uicontextmenu contained in the “options” property of the parent sigTOOLResultData object.

For an example of using custom objects, see the menu_JPETH and spJPETH functions and the associated jpeth class methods.

[11] Note that the techniques for dealing with custom objects are likely to evolve through future releases of sigTOOL

Empty data fields

Some data field may be left empty if no result was calculated for a particular channels combination. For example, in calculating an event autocorrelation:



The data field of the sigTOOLResultData object is as follows:

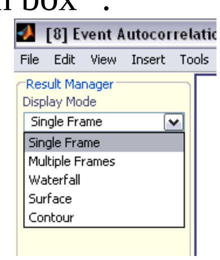
```
>> x=getappdata(6, 'sigTOOLResultData');
>> x.data
ans =
    'Channel'    '1'               '2'
    '1'          [1x1 struct]              []
    '2'                    []      [1x1 struct]
```

There is an empty entry for channels 1->2 or 2->1 as we are interested only in the autocorrelations.

- **The displaymode, plotstyle and viewstyle fields**

These fields determine how the data will be plotted by default. There is some overlap of meaning in these fields.

displaymode is a string that generally matches one of the settings in the Result Manager Display Mode selection box[12]:



[12] More options are likely to be added

plotstyle is a handle to a function that will do the plotting. sigTOOL's standard plotting routines are

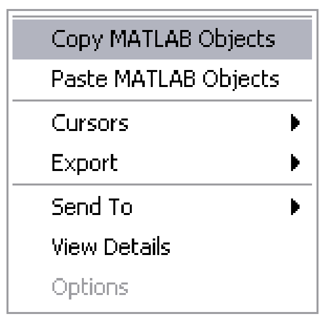
scFrames	which plots single and multiple lines (e.g. waveform averages)
scBar	for histograms (e.g. PETHs)
scSurf	for surfaces
scContour	for contour plots
scScatter	for scatter diagrams (e.g. rasters)

When custom-defined objects are used, plotstyle should be set to @plot and a plot method should be overloaded for that class (see above).

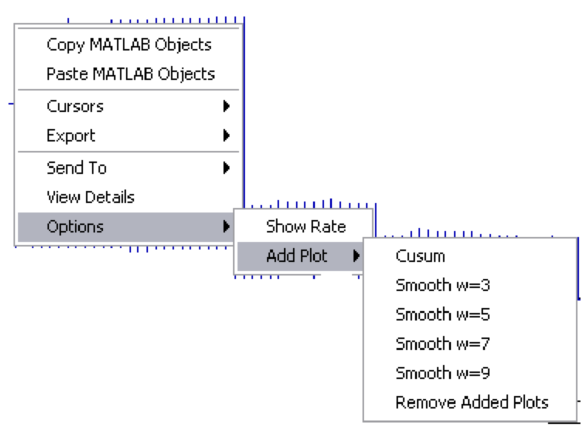
viewstyle is a string, either '2D' for 2-D graphics or '3D' for 3-D graphics.

- **the options field**

sigTOOL adds context-sensitive menus to the result plots. The standard menu is[13]:



In this case, the options menu is not enabled. To enable it, create a MATLAB uicontextmenu object and place its handle in the sigTOOLResultData object's options field. Here is an example for an event correlation:



[13] At the time of writing. You may get a more extensive menu in sigTOOL

This menu allows us to swap between showing spike counts and rates and to add additional plots such as cusums or smoothed versions of the correlations. To create these menus, you need to create your own uicontextmenu and program the callbacks (type help uicontextmenu at the MATLAB prompt for details). To add a plot, call the scAddPlot function from your callback.

Appendix A: Detailed specification of sigTOOL data representation

Waveform channels

Waveform channels are represented within a scchannel object as follows:

```
>> chan{1}

ans =

                  adc: [48028475x1 adcarray]
    channelchangeflag: [1x1 struct]
                  hdr: [1x1 struct]
                  mrk: [0 0 0 0]
                  tim: [0 960.5695]
```

In this case, we have a ~48Msample adcarray.

- The *hdr* field

```
>> chan{1}.hdr

ans =

               adc: [1x1 struct]
           channel: 1
       channeltype: 'Continuous Waveform'
    channeltypeFcn: ''
           comment: 'No comment'
       markerclass: 'uint8'
            source: 'C:\Program Files\MATLAB\R2006b\work\12oct04.smr'
               tim: [1x1 struct]
             title: 'Depth'
```

1. The hdr.adc field
   This field contains the information needed to interpret the contents of chan{1}.adc:

```
>> chan{1}.hdr.adc
ans =
                  DC: 0
                Func: []
              Labels: {'Time'}
       MultiInterval: [0 0]
           Multiplex: 1
             Npoints: 48028475
      SampleInterval: [20 1.0000e-006]
               Scale: 1.5259e-004
         TargetClass: 'adcarray'
               Units: 'mV'
                YLim: [-1.9820 0.8620]
```

- **DC, Func, Labels, Scale and Units fields**

The DC, Func, Labels, Scale and Units fields contain the values required to create the adcarray in chan{1}.adc (see adcarray objects above). In this case, a 16-bit analog-digital-convertor was used and the input range was ±5V. One bit is used for the sign, so 1 LSB represents $5/ 2^{(16-1)} = 1.5259\times10^{-4}$V. A 1000x amplifier was used, so the Scale is set to 1.5259×10^{-4} and the Units to 'mV' to convert the ADC values to real-world values.

- **Npoints**

Npoints is a row vector with one element for each column of waveform data. In this case, data were sampled continuously and there is only one column of data and hence only one element in Npoints This is 48028475: the number of samples on the channel. If data are multiplexed, Npoints is set to (Number of SubChannels x Number of Samples per SubChannel).

- **SampleInterval**

The sample interval is the period (in seconds) between samples. *SampleInterval* is represented by two numbers, their product is the sample interval: in this example $20 \times 1*10^{-6}$. The first element of *SampleInterval* should be an integer (if this is possible given the sample rate), the second is the multiplier to convert to seconds. In the example case sampling was at 50000Hz so the sample interval is 20µs.

General note: The reason for having two elements in *SampleInterval* is to avoid rounding errors. 20×10^{-6} is not exactly representable in binary. The sampling rate here is $1/20\times10^{-6}$Hz, but in IEEE double precision, that yields 49999.999999999993Hz because of rounding. However, 1/(1e-6)/20 yields 50000Hz exactly. The error is trivial, but it can be handy to have an exact solution when programming to analyze possible out-by-one errors. Hence the use of two elements for *SampleInterval*.

- **TargetClass**

This is the MATLAB class to which the data will be cast when loaded, and may be different to the class on disc. In this case, data are cast to the sigTOOL-defined *adcarray* class while data on disc are stored in *int16*.

- **YLim**

YLim contains the minimum and maximum values observed on the channel after scaling, offsetting by DC etc. *YLim* is used to set the limits if the y-axes in sigTOOL.

- **MultiInterval and Multiplex**

Multiplex is simply the number of subchannels of data that are stored interleaved in this channel (see Multiplexed data above).

MultiInterval is similar to *SampleInterval* but contains the interval between samples on each subchannel. In the example, the data are not multiplexed. *MultiPlex* is therefore set to 1 and *MultiInterval* to [0 0] which are the default values.
None of the file types supported to date provide the information needed for MultiInterval and sigTOOL core routines always assume values of [0 0] even if non-zero values are entered i.e. synchronous sampling.

2. **hdr.channel**
   hdr.channel is not used in sigTOOL. This field is set to the channel number in the original source file e.g. if data were imported from channel 2 of a pClamp file, hdr.channel would be set to 2.

3. **hdr.channeltype**
   hdr.channeltype is string describing the data type. For a waveform channel, the string must contain ‘Waveform’. In this case, data have been continuously sampled so hdr.channeltype is ‘Continuous Waveform’. Periodically sampled data would be ‘Episodic Waveform’ or, if the periods of sampling were identical with the same pre- and post-trigger sample numbers, ‘Framed Waveform’.

General Note: sigTOOL looks for the ‘Waveform’ keyword in the string. ‘Continuous’, ‘Framed’ and ‘Episodic also serve as keywords to identify the way in which data should be treated by sigTOOL. Keywords are case sensitive.
You can add other words to the string, including user-defined keywords to be interpreted by user-written functions as part of a package to apply sigTOOL to a particular research field e.g. a neurophysiologist storing a series of action potentials or spikes in a ‘Frame Waveform’ could label the *channeltype* as ‘Framed Waveform

(Spike)'. The core sigTOOL functions will recognize only the 'Waveform' and 'Framed' keywords. The neurophysiological analysis package could react to the presence of the 'Spike' keyword.

- **hdr.channeltypeFcn**

hdr.channeltypeFnc is the name of a user-written function that will be executed from sigTOOL:

1. When the mouse is used to select a marker from this channel in the sigTOOL data view.
2. When a user-written function needs to access data in a custom designed way for a custom designed *channeltype*.

This increases the flexibility of sigTOOL to deal with data types that have not been predicted in the core functions.
In the example used here, the string is empty. There is no custom function associated with the data.

- **hdr.classifier**

Introduces in sigTOOL version 0.9, this field will be used to support information theoretic analyses of spike trains where stimuli of different classes have been delivered.
When the present channel is the spike:
hdr.classifier.By contains the channel number of the stimulus channel
When the present channel is the stimulus:
hdr.classifier.For indicates the spike channel associated with this stimulus. The markers on the stimulus channel classify the stimuli

- **hdr.comment**

hdr.comment is a user-supplied string describing the channel contents.

- **hdr.markerclass**

hdr.markerclass contains the class of the markers stored in the mrk field of the channel. In this case '*uint8*'. There are no limits on the marker class.

- **hdr.source**

hdr.source is a string containing the file name of the file from which this channel was derived. In this case, from a proprietary file format: Cambridge Electronic Design's Spike2 for Windows software,

- **hdr.tim**

hdr.tim is similar to *hdr.adc* but contains the data needed to interpret the *tim* field of the channel rather than the *adc* field.

```
>> chan{1}.hdr.tim

ans =

    Class: 'tstamp'
     Func: []
    Scale: 5.0000e-006
    Shift: 0
    Units: 's'
```

- **The Func, Scale, Shift and Units fields**
  These fields contain the values required as input to the *tstamp* constructor function (see *tstamps* above) or to scale the values on disc (which will normally be in clock ticks) to real-world values. In this case, each clock tick is 5µs.

- **Class**
  The *hdr.tim.Class* field specifies the class to cast the data to from disc (Note: this is normally *tstamp*, but if there are few data, sigTOOL may cast the *tstamp* to type double if that will save space. This is presently done if there are fewer than 1000 timestamps).

- **hdr.title**

hdr.title is the title of the channel. This should be a short string as it will be used to label the channel in the sigTOOL graphics displays.

General note: No limits are placed on the length of strings in sigTOOL. You can set any of the string fields above to any value you choose but if you have very long strings, e.g. in the title field, this may cause the sigTOOL GUI displays to become cluttered.

Event and marker channels

Marker channels are stored in the same way as Waveform channels but the *adc* field, and the *hdr.adc* field that describes it, are left empty.

Note: If adc data are present, the channel will be a ‘Episodic’ or ‘Framed’ waveform channel. An additional, application specific keyword might then be added e.g. ‘Spike’ in neurophysiology.

For event channels, in which there is also no marker data, the *mrk* and associated header fields will be empty.

Both marker and event channels are described by the same keyword in *hdr.channeltype* which will contain the ‘Edge’ keyword which may be qualified as ‘Rising Edge’ or ‘Falling Edge’. Alternatively, if both edges are of interest, *hdr.channeltype* is set to ‘Pulse’.

sigTOOL distinguishes between event and marker channels by whether *mrk* is empty or not. In general, when importing data from external file formats, it may be simplest to fill the *mrk* field with zeros. That ensures that space is reserved for the markers in the sigTOOL data file should they be needed at some point in the future.

General note: If markers are present but all the same value, sigTOOL generally ignores them. sigTOOL also provides other options that tell it how to interpret the markers, e.g. how many to display in a data view. These options are described fully elsewhere (#TODO: insert link).

Custom channels
You can define a custom channel type by including the ‘Custom’ keyword in *hdr.channeltype*. In the sigTOOL data display, custom data channels will be treated as though they were edge channels. However, if the *hdr.channeltypeFcn* function is defined, any markers associated with the events will be highlighted and will be active to mouse selection.

Keyword precedence
Where appropriate, e.g. to display data in the standard data view, sigTOOL treats the keywords in *hdr.channeltype* according to the following precedence

Custom>Edge>Pulse>Waveform

e.g. if *hdr.channeltype* is ‘Custom Waveform’, the channel will be treated as a ‘Custom’ channel.

Appendix B: MAT-file Utility Functions

These MAT-file functions can be used to manipulate large data sets in a standard version 6 MAT-file including the sigTOOL .kcl data files. These MAT-file utilities for writing data to a MAT-file and modifying variables mostly require that you are working in the final variable in the MAT-file i.e. the last variable saved using MATLAB's *save* function. The write functions work only on standard matrices, not structures objects etc, and require the variable to be real valued (not complex).
Note that the inline help for these functions provides far more detail than this document (type e.g. "help where" at the MATLAB prompt – after running sigTOOL).

- **Reading data files**

▪ **where**

Where acts similarly to whos but in addition provides information about the class of the data on disc and the byte offsets into the file. This can be used to read the file using low level I/O or memmapfile
For all variables in a file

```
s=where(filename)
```

For a specified variable

```
s=where(filename, varname)
```

For a particular field of a structure

```
s=where(filename, varname, fieldname)
```

or

```
s=where(filename, varname, fieldname1, fieldname2…)
```

if you have structures within structures
Replace fieldname with propertyname for objects

`[s,swap]=where(…)` sets swap to 0, if the MAT-file endian format is the default for the platform you are using , or to 1 if byte swapping will be needed.

▪ **endian**

```
endian(filename)
```

returns 'ieee-le' for a little-endian MAT-file and 'ieee-be' for big-endian

- **Writing large data sets**

The output of where can be used to help with reading subsets of data from a variable. The following routines are to assist with writing data:

- **MATOpen**

MATOpen creates a new MAT-file, or if it exists, opens an existing MAT-file in the appropriate endian mode and returns a MATLAB file handle

```
fh=MATOpen('myfile', permission)
```

for valid strings for permission see MATLAB's *fopen*.

All the routines below require that the target variable is the last variable in the MAT-file. This can be checked with **CheckIsLastEntry**(filename, varname) which returns true or false. If unknown, the name of the last variable in a file can be determined with **GetLastEntry**(filename) which returns a string.

In addition, all the routines require that data is stored on disc as the same class as the target variable. MATLAB's *save* command casts data to the smallest compatible data type e.g. a double array with values all between 0 and 255 will be cast to uint8. Run **RestoreDiscClass**(filename, varname) before calling the **AppendXXXX** functions to restore the class of the data on disc to that of the target variable in memory.

The AppendXXXX function can only be used with real valued variables (not complex data)

- **AppendVector**

Adds data to the end of a row or column vector

```
AppendVector(filename, varname, vector)
```

- **AppendColumns**

Adds columns to the end of a column vector or 2D matrix.

```
AppendColumns(filename, varname, matrix)
```

Varname and matrix may be column vectors or a 2D matrices. They must have the same number of rows.

- **AppendMatrix**

Equivalent to AppendColumns but adds data to the final dimension of an N-dimensional matrix where N is >=2.

Suppose we have a 100x100x3 RGB image stored in variable img in myfile.mat. We can add a second 100x100x3 image from variable newimg using

```
AppendMatrix('myfile', 'img', newimg);
```

>load myfile img
will then return a 100x100x6 matrix with the two images.
Had newimg been 100x100x6, varname would have become 100x100x9.

To organize the data into a higher dimesional matrix, use AppendMatrix in combination with AddDimension (see below)

- **AddDimension**

Can be used with AppendMatrix to save data to a higher dimension. Using the example above we could use

```
AddDimension(filename,'img'); % Convert img to a 4D matrix
                              %(100x100x3x1)
AppendMatrix(filename, 'img', newimg);% Add the 3D newimg to the
                                      %4th dimension
```

Now

```
load myfile img
```

will return a 100x100x3x2 matrix, the 4th dimension is the image number

In general, for

```
AppendMatrix(filename, varname, matrix)
```

- If varname and matrix have the same number of dimensions, matrix will be added to the highest dimension of varname (e.g. if varname points to a 100x100x9 matrix and we add a 100x100x6 matrix, we will end up with a 100x100x15 result. The element ordering on disc will be the same as if we had *save*d a 100x100x15 matrix in the first place. MATLAB's *load* command can be used to access the matrix.
- If varname has 1 dimension more than matrix, matrix will be treated as a submatrix or (set of submatrices) and added to varname whose final dimension will be incremented by:
  size of ultimate dimension of matrix / size of penultimate dimension of varname
  which must be integer (e.g. suppose varname points to a 100x100x3x22 matrix and we add a 100x100x9 matrix, we will produce a

100x100x3x25 result – matrix is assumed to contain three 100x100x3 matrices e.g. three RGB image frames).

Note that AddDimension adds the dimension to the data on disc. Most MATLAB functions including *load*, *whos* etc strip away any trailing singleton dimensions so the effects of AddDimension will not show until the final dimension is >=2.

- **RestoreDiscClass**

MATLAB's *save* command casts data to the smallest compatible data type e.g. a double array with values all between 0 and 255 will be cast to uint8. Run **RestoreDiscClass**(filename, varname) before calling the **AppendXXXX** functions to restore the class of the data on disc to that of the target variable in memory.

- **CheckIsLastEntry & GetLastEntry**

The AppndXXX functions require that the target variable is the last variable in the MAT-file. This can be checked with **CheckIsLastEntry**(filename, varname) which returns true or false. If unknown, the name of the last variable in a file can be determined with **GetLastEntry**(filename) which returns a string.

Appendix C: sigTOOL data file specification

Channel data are stored in sigTOOL data (*.kcl) files in a format that is similar to their storage in memory as described above. As the *.kcl file is simply a MAT-file that follows a specific variable naming convention, data can be saved to the file using the MATLAB builtin *save* command (although sigTOOL includes a number of functions to assist in writing large data matrices to the files).

Mode 0 (Standard format)

This is the standard *.kcl file format.

- Channel header

  The channel header is contained in the variable named headxxx where xxx is the channel number (head1, head2 etc.).The contents of the header variable is identical to the *hdr* field of a sigTOOL channel cell array element (as described above).

- Channel data

  The *tim*, *adc* and *mrk* data are stored in fields with those names in a structure named chanxxx (where, as before, xxx is the channel number: chan1, chan2 etc.).

  For *tim*, only the data that will be represented in the *tim.Data.Stamps* field is stored on disc, while for *adc*, only the data to be represented in the *adc.Data.Adc* field is stored on disc (the relevant *adcarray* or *tstamp* objects will be constructed using the scale, offsets etc. from the corresponding header variable). The data for the *mrk* field is stored directly.

Mode 1 (Alternative format)

Occasionally, there may be too much data on a channel for the chanxxx structure described above to be formed in MATLAB memory. This is likely to occur only with the *adc* field as timestamps and markers occupy relatively little memory. Nevertheless, for generality, each of the *adc*, *tim* and *mrk* fields are separated into separate entries in the data file. In these cases, an alternative – but similar- format is used:

- **Channel header**

  The headxxx structure is the same as above.

- **Channel data**

  Instead of being saved in a structure named chanxxx, the *tim*, *adc* and *mrk* data are stored as variables named timxxx, adcxxx and mrkxxx (where xxx is the channel number as above). sigTOOL includes functions to allow standard matrices to be written in stages e.g. a column at time.

Mixed mode

In mixed mode chanxxx contains the adc and tim fields only. Markers are saved in a separate variable called mrkxxx. This allows complex markers such as cell arrays of structures to be loaded separately from the remaining data e.g. where mrk is used to represent complex metadata.

General note: User-specified variables can also be added to a *.kcl file as long as there are no variable name clashes.

Appendix D: Memory mapping of data

sigTOOL stores data on disc and uses the standard MATLAB *memmapfile* function to map the data into memory through the host operating system. A *memmapfile* object is a member of standard class defined in MATLAB. To simplify programming, sigTOOL aggregates the *memmapfile* object into one of two sigTOOL-defined classes: *tstamp* for representing timestamps in the *tim* field and *adcarray* for storing digitized analog signals, video clips etc in the *adc* field.

The *adcarray* and *tstamp* classes, and the methods associated with them, are defined in the *@adarray* and *@tstamp* folders in the sigTOOL utils folder.

The description that follows assumes some knowledge of MATLAB class constructors and method definitions and of the standard *memmapfile* class.

adcarrays

The *adcarray* constructor function is used to create an *adcarray* object. You can create an empty *adcarray* by calling *adcarray* with no input:

```
>> a=adcarray();
```

You can also load a standard numeric matrix into an *adcarray* by passing the matrix as an input to it, e.g:

```
>> a=uint8([1 2 3 4])

a =

    1    2    3    4
>> a=adcarray(a);
```

There are instants where this is useful e.g. to represent data temporarily in RAM but for the most part the *adcarray* is used to store *memmapfile* objects e.g.

Create a *memmapfile* object:

```
        map=memmapfile(filename,...
            'Repeat',1,...
            'Format',{‘int16’[10000 100] 'Adc'},...
            'Offset',272);
```

In this case, a 10000x100 *int16* matrix stored at a byte offset of 272 into the file *filename* is mapped into the *map.Data.Adc* property of the *map memmapfile* object.

Here is a further example showing the contents of map

```
map =

Filename: 'C:\kcl\Program Files\MATLAB\R2006b\work\12oct04.kcl'
    Writable: false
      Offset: 272
      Format: {'int16' [48028475 1] 'Adc'}
      Repeat: 1
        Data: 1x1 struct array with fields:
              Adc
```

In this case the Adc property contains a 48028475 element column vector. The MATLAB workspace memory saved by using a memmapfile object can be assessed with:

```
>> whos map
  Name      Size            Bytes  Class         Attributes

  map       1x1               456  memmapfile
```

where we see that only 456 bytes are used to represent this vector in the MATLAB workspace. 48028475*2 bytes of virtual memory address space will be allocated when the adcarray is first accessed (note accessed, not created). When you have finished using a channel, you can call the scRemap() function to free-up this virtual memory space allowing other channels to use it. Many of the sigTOOL GUI functions call scRemap automatically as needed.

The *map.Data.Adc* data can then be accessed using the standard MATLAB matrix syntax e.g. *map.Data.Adc(:,2)* returns the 10000 *int16* elements of the second column of the matrix. The advantage of using *memmapfile* is that the data are represented through your system's virtual memory.

Commonly in sigTOOL, these *int16* values will be the samples from an analog-digital convertor. The memmapfile can be aggregated into an *adcarray* object by calling:

```
>>channel{n}.adc =adcarray(map,...
                Scale,...
                DC,...
                Func,...
                Units,...
                Labels,...
                swap);
```

Note in this case, that the *adcarray* object has been placed in the *adc* field of the structure stored in the *nth* element of a cell array of such structures. This is how sigTOOL stores channel data.

The arguments to the adcarray constructor function are:

map: the *memmapfile* object created above.

Scale: the value to multiply the values by to convert to real-world numbers. Scale is a double precision float[14], and the values in the *map.Data.Adc* field will also be cast to double before doing the multiplication.

DC: A DC offset to add to the result after scaling (double precision float[6]; default zero).

Func: A handle to a function that will be used to transform the data after scaling and adding the offset. Func is usually empty i.e. Func=[]. Otherwise it will be the handle of a simple function such as @abs, in which case the data will be rectified before being returned.

Units: A string: the real world units e.g. mV.

Labels: A cell array of strings describing the real-world meaning of each dimension in the data matrix. In the example above we have a 2-dimensional adc matrix: the rows represent time and each period of sampling is stored in a separate column so Labels might be {'Time' 'FrameNumber'}.

swap: Swap is a logical flag (true or false) which indicates whether the data stored in the *memmapfile* object need to be byte swapped before scaling offsetting etc. Byte swapping allows files to be mapped when the source file has a different endian format than the host platform. It therefore allows sigTOOL data files to be moved between operating systems.

The methods defined for the *adcarray* class do all the scaling, offsetting etc for you once an *adcarray* object has been created. Thus channel{n}.adc(:,2) returns the contents of column 2 (i.e. frame 2) as a double precision column scaled to the real-world units.

For further details of the *adcarray* class, type "help adcarray" at the MATLAB command prompt after running sigTOOL. The following functions have been overloaded for use with adcarrays:

[14] The adcarray constructor will accept values that are not double precision for scale and offset but specifying non-double values will result in mixed precision arithmetic being used and non-double results being returned by the adcarray methods.

SUBSREF, GET, SET, DISPLAY, END, LENGTH & SIZE.

SUBSASGN, HORZCAT & VERTCAT are also overloaded but return double precision results rather than *adcarrays*.

Use helpwin for further details of these methods applied to *adcarrays*.

tstamps

The *tstamp* class is designed to hold timestamps but is, in most respects, identical to the *adcarray* class.
The differences are:

1. The *Scale* property is used to multiply the data on disc to return a time stamp in base clock ticks. This will be returned as a floating point number but will usually be a "flint" i.e. a floating point representation of an integer.
2. The *Units* property specifies the length of the base clock tick and is used as a scaling factor to scale the output to seconds such that:
   timestamp*Scale*Units=time in seconds
3. The *DC* property is replaced by *Shift*. When *Shift* is zero (the default), times will normally be expressed relative to the start of sampling. A value of -10 in *Shift* will return times relative to 10s so timestamps before 10s will be negative (assuming here that *Units* is set to seconds)[15].
4. The data are stored in the *Map.Data.Stamps* property, rather than *Map.Data.Adc*
5. There is no *Labels* property for *tstamps*

General note: The memory maps stored in adcarray and tstamp objects have their Writable property set to false by default. If you attempt to write to an *adcarray* or *tstamp* object they will be converted to double precision matrices. However you can write directly to the memmapfile object that they contain. Note though, that if you do this, you will overwrite the data stored on disc.

To overwrite the disc data set Map.Writable to true. Remember that any data you write must be suitable for subsequent scaling, offsetting etc. by the

[15] Shift is presently always set to zero in sigTOOL.

adcarray and *tstamp* class *subsref* method, and for being passed through the function pointed to by the handle in *Func*: You must therefore:

1. Apply the inverse function to *Func*
2. Subtract the *DC* offset (or Shift for *tstamps*)
3. Divide the data by *Scale*
4. Cast the data to the appropriate class e.g. *int16*. Note that you will also need to check the class on disc.

It will generally be easier to set up a temporary channel where the data in the memmapfile object are already in double precision so that *Scale*=1, *DC*=0 and *Func* is empty. This is what sigTOOL does with temporary channels.

Appendix E: Managing virtual memory

No virtual memory is allocated when you load a file by calling scOpen. Instead, virtual address space is allocated only when you access a channel. Many functions in sigTOOL dynamically manage this memory by calling the scRemap function when necessary. However, this can only work properly if certain conditions are met. Note that, scRemap will only be needed for very large files and/or if you simultaneously load many files.

Normally, MATLAB passes variables between functions using pass-by-value. Each function has an independent copy of the data[16]. However, this is not true for some objects, including the memmapfile objects used to represent data in sigTOOL. Memmapfile objects are passed by reference, so there is only ever a single master copy maintained in memory.

For scRemap to work properly, you need to ensure that the only active reference to the data on a channel is that in the application data area of the relevant sigTOOL data view. This is easily achieved by:

1. Passing a handle rather than a channel cell array as the argument to functions
2. Clearing unwanted channel references in a function before calling another function,

e.g.

```
function myFunc(fhandle,……)
.
.
channels=getappdata(fhandle, 'channels');
thischan=channels{4};
.
.
.
clear('channels');
clear('thischan');
MyNextFunc(fhandle, ……);
return
end
```

[16] Internally, recent versions of MATLAB use pass-by-reference where possible combined with copy-on-write memory management but these processes are transparent to the MATLAB user

The code below shows how scRemap can be used:

```
try
    data=channels{10}.adc(1:1000);
catch
    % Likely to be Out of Memory error.
    % Clean up virtual memory if so
    if isOOM()
        % Out of Memory error.
        clear('channels', 'data');
        scRemap();
        channels=getappdata(fhandle, 'channels');
        data=channels{10}.adc(1:1000);
    else
        % Not an Out of Memory error.
        rethrow(lasterror());
    end
end
```

If the

```
data=channels{10}.adc(1:1000)
```

causes an out-of-memory error the catch sequence will be executed[17]. This tests the type of error by calling isOOM (a sigTOOL function that returns true if the last error was an out-of-memory error and false otherwise). We need to clear any local references to the channel data with

```
clear('channels', 'data');
```

(in this example data should be empty but let's clear it anyway).

If isOOM returns true, we call scRemap(). This creates a local copy of channels that has scope only within scRemap, deletes the copy in the application data area and overwrites the memory map object for each channel with a completely new memory map. This deletes the pre-existing map releasing all virtual memory associated with it. Not uncommonly, this process will free-up 1-1.2Gb of virtual address space[18].

Next, we reload channels and try to allocate data again:

```
channels=getappdata(fhandle, 'channels');
data=channels{10}.adc(1:1000);
```

Hopefully, this will execute without error. It not, a new error will be thrown and we leave MATLAB to handle that in the usual way.

[17] Note we are taking only 1000 data points here but virtual address space will be allocated for all data on the channel. That may be many hundreds of Mb – hence the out-of-memory error

[18] Note: If you do not clear all references to channels in the calling stack scRemap can make matters worse by creating two separate but identical memmapfile objects each needing its own virtual memory space.

Calling scRemap with arguments

In the examples above, scRemap was called with no inputs. This is the sledge-hammer approach that releases all channels on all open sigTOOL files.

The Remap button in the sigTOOL Channel manager calls scRemap in this way. If you encounter out-of-memory errors they will usually be solved by hitting this button (but if you are in debug mode, make sure you have cleared the channel reference stack).

The following calling conventions are also supported:

```
scRemap(fhandle)
```

Releases only channels for the specified figure

```
scRemap(fhandle, list)
```

Releases the channel specified in list for the specified figure

```
scRemap(fhandle, list, 'exclude')
```

Releases all channels except those in the list for the specified figure

Calls of these types are made by many of the sigTOOL core functions.

If scRemap does not solve your out-of-memory errors, a simple way to load multiple files