

MATLAB SON Library

Malcolm Lidieth
King's College London
Updated October 2005 (version 2.0)

This pack now contains two libraries:

SON2 is a backwards-compatible update to the previous library. It provides a set of low-level routines that read from a SON file. These use standard MATLAB code, so they can be run under any operating system supported by MATLAB. The new version includes better online help, improved error handling and a number of optional input arguments.

SON32 is a Windows only package. It allows MATLAB to read and write SON files (and to create them from scratch) by calling the functions in the son32.dll file that is supplied by CED with Spike2 for Windows. This is the application extension that Spike2 uses to read/write files.

Comparison of the two libraries:

SON2	SON32
Use under any operating system	Windows only
Reads SON files.	Reads/writes/creates SON files
Requires MATLAB version 5 or higher	Requires MATLAB version 7.1 or higher
Easy to use. Can be used with minimal knowledge of SON file structure	Designed for programmers – knowledge of the SON filing system is expected
Stand-alone, low-level code	Requires son32.dll which is supplied with Spike2 for Windows (Cambridge Electronic Design)

Both libraries include detailed online help.

USING THE SON2 LIBRARY

- **Opening a file**

The SON file should be opened using the MATLAB 'fopen' command e.g.

```
fid=fopen('demo.smr');                      reading only
```

or

```
fid=fopen('demo.smr','rb+');                reading and writing
```

If you are working on a non-Windows system you may also need to set the machine format in the call to fopen.

- **Reading a channel**

Data is read using the SONGetChannel command. A call to SONGetChannel takes the form:

```
data=SONGetChannel(fid,chan);
```

or

```
[data, header]=SONGetChannel(fid,chan);
```

Where *data* will be returned with the data stored on the channel, *fid* is the file handle from fopen() and *chan* is the channel number - numbered from 1 to (maxchan-1) as in Spike2. *header* is an optional output argument and provides details of the channel type, sample rates etc.

SONGetChannel identifies the data type in the channel and calls one of a set of routines depending on the data type. These routines can also be called directly but this is discouraged – in the future, code may be added to SONGetChannel but not included in the other routines.

New in Version 2

The SONGetChannel command now also accepts optional arguments. These:

- Allow parts of the channel data to be loaded instead of all data
- Allow you to choose the time units for returning timestamps
- Let you choose whether to scale ADC data
- Allow a progress bar to be displayed when reading a file

- **Reading parts of the data**

Within a SON file, data is stored in arbitrarily sized blocks. For continuously sampled data, you can read one, or a subset of the blocks as follows

```
data=SONGetChannel(fid, chan, 10);
```

reads block 10 only

```
data=SONGetChannel(fid, chan, 10, 20);
```

reads blocks 10 to 20 inclusive.

If sampling on the relevant channel was done in triggered-mode, there will be gaps between the epochs of data. The SON library detects these gaps automatically. The calls shown above would then return epoch 10 or epochs 10-20 inclusive.

All remaining optional arguments are supplied as string inputs. The strings may appear in any order but must be the last arguments. Arguments that are not relevant to a particular channel type are ignored.

- Setting the timeunits

By default, the library returns times in seconds but you can choose microseconds, milliseconds, seconds or clock ticks e.g.:

```
data=SONGetChannel(fid, chan, 'microseconds');
```

returns all data on the channel scaled in μ s.

```
data=SONGetChannel(fid, chan, 10, 'ticks');
```

returns block 10 only scaled in clock ticks

```
[data, header]=SONGetChannel(fid, chan, 10, 20, 'seconds');
```

returns blocks 10-20 inclusive scaled in seconds.

The timeunits affect the *data* and the values returned in *header* (except the *header.sampleinterval* field which is always in seconds for backwards compatibility).

- Scaling data

For ADC data, the SON file stores a scale and offset to use to convert 16-bit integer values to floating point real world values. The scale option causes data to be returned appropriately scaled e.g. :

```
data=SONGetChannel(fid, chan, 'scale');
```

```
[data, header]=SONGetChannel(fid, chan, 10, 20, 'seconds', 'scale');
```

- Progress bar

Selecting progress displays a progress bar during a read operation. If you do not specify the number of blocks (or epochs) to read, the progress bar will only appear if the read request involves loading more than 10 blocks.

```
[data, header]=SONGetChannel(fid, chan, 1, 200, 'seconds', 'scale', 'progress');
```

You can put the optional arguments into a cell array e.g.:

```
Options{1}='scale';
```

```
Options{2}='milliseconds';
```

```
Data1=SONGetChannel(fid, 1, Options{:});
```

```
Data2=SONGetChannel(fid, chan, 1, 100, Options{:});
```

For convenience, the examples below do not use optional input arguments.

ADC data

data is returned either as a single row vector for continuously sampled data or as a matrix of row vectors for triggered sampling i.e.,

either

Sample 1	Sample 2	Sample 3	Sample 4	Sample 5 etc..
----------	----------	----------	----------	----------------

or

Epoch 1 Sample 1	Epoch 1 Sample 2	Epoch 1 Sample 3
Epoch 2 Sample 1	Epoch 2 Sample 2
Epoch 3 Sample 1etc

When the source file is derived from frame based acquisition software (such as CED's Signal), the Epochs in each row of *data* should be aligned with the trigger. However, this will not generally be the case with Spike2 derived files. The triggered sampling mode introduced in Spike 2 Version 5 produces sweeps of variable length and the trigger will occur at variable distances into each row of *data* (the sweep time and pre-trigger time set in Spike2 are treated as minimum values for each sweep – not constant values from sweep to sweep).

The *transpose* flag is intended to keep track of the orientation of waveform *data*. The row vector organization is best for some functions while column vectors are better for others. With large data arrays, frequently transposing them will waste too much time. It may be better to transpose the data in the main workspace and set the transpose flag so that you know the orientation of the data. (Note: *transpose* is defined in SONGetChannel. A direct call to SONGetADCChannel, or similar channel-type specific functions, will leave *transpose* undefined).

Integer data can be converted to floating point simply by using `double(data)` or `single(data)` but this leaves the values unaffected. The SONADCToDouble and SONADCToSingle functions do the conversion and scale the output according to the values of scale and offset stored in the SON file. Both functions take the same form e.g

```
[dataout {,headerout}]=SONADCToDouble(datain {,headerin});
```

headerin is a structure such as the header returned by SONGetChannel. Only the scale, offset and kind fields need be present.

In both cases the output is

$$(\text{datain} * \text{headerin.scale} / 6553.6) + \text{headerin.offset}$$

Note that setting the 'scale' option in the call to SONGetChannel causes SONADCToDouble to be called automatically.

RealWave data

RealWave data is read and stored in 32 bit floating point (single precision) format. The inputs and outputs are as for ADC data (above) except that *data* is single precision floating point and the *header* does not contain a *scale* or *offset* field. Instead a *min* and *max* field are present that give expected minimum and maximum values in *data* (Note: RealWave channels written in Spike2 may contain *scale* and *offset* of the source channel in *max* and *min*.). For RealWave data *header.kind* is set to 9.

Event data

Event data is read as 32-bit integers giving the time of event in clock ticks and returned in 64-bit floating point (double precision) format with the times given in seconds (by default) or in the timeunits selected by an optional argument to `SONGetChannel` (see above).

```
[data {,header}]=SONGetChannel(fid, chan);
```

where *data* is a column vector containing the times of each event in sequence expressed in seconds since the start of sampling.

If *header.kind*=4 data is of EventBoth type. The header will then contain a field called *h.initLow*. This defines the initial state of the channel: 1=low, 0=high.

If *initLow*=1 the first event will be a low to high transition. *data* will contain the times of low-to-high transitions in its odd numbered elements and high-to-low transitions in its even numbered elements.

If *initLow*=0 the first event will be a high to low transition. *data* will contain the times of high-to-low transitions in its odd numbered elements and low-to-high transitions in its even numbered elements.

The *h.nextLow* field is a flag used during sampling. Although copied to the header it is not used here.

Marker data

Marker data is treated in the same way as event data except that the returned *data* is a MATLAB structure containing two fields.

```
[data {,header}]=SONGetChannel(fid, chan);
```

returns a *data* structure containing for example

timings: [1x38 double]

markers: [4x38 char]

with a channel with 38 markers.

The *data.timings* field is a column vector containing the marker times just as with event data. In the SON system, each marker event has associated with it four single byte markers. Each marker makes up a column vector in *data.markers* .

To access data for individual events use subscripting e.g.

data.timings(10) gives the time of the tenth event marker and *data.marker(10,3)* gives the value of marker 3 for event 10. You can access all markers in one go using range subscripts e.g. for channel 31 of the demo.smr file supplied with Spike2:

```
>> [d,h]=SONGetChannel(fid,31);
>> data.markers(:,1)'

ans =

thisisanexampleofdatasampling.Goodbye!

>>
```

ADCMark (WaveMark) data

ADCMark data is treated in the same way as marker data except that the returned *data* is a MATLAB structure containing three fields.

```
[data {,header}]=SONGetChannel(fid, chan);
```

returns a *data* structure containing for example

```
timings: [279x1 double]           % 279 events
markers: [279x4 char]             % each with four markers...
adc: [279x32 int16]               % ...and 32 adc values
```

The *data.timing* and *data.marker* fields are the same as with ordinary marker data. The *data.adc* field contains the ADC data associated with each event. The ADC data is organized as a series of row vectors, with one vector for each marker event. Access data for individual marker events as before. For the ADC data you can access individual values e.g

```
>> data.adc(5,15)
ans =
-65
```

gives the value for the 15th adc sample of the 5th marker event.

```
>> plot(d.adc(5,:))
```

plots all the data associated with the 5th event.

```
>> plot(d.adc(5:10,:))
```

plots all the data associated with the 5th through 10th event.

Markers are generally numbers rather than characters. Use int8 or uint8 to inspect them e.g.

```
>> int8(d.markers)
```

```
ans =
```

```
1  0  0  0
1  0  0  0
1  0  0  0
1  0  0  0
1  0  0  0
```

```
--more--
```

RealMark data

RealMark data is treated in the same way as ADCMark data but returns real valued data in *data.adc*.

```
[data {,header}]=SONGetChannel(fid, chan);
```

returns a *data* structure containing for example

```
timings: [13481x1 double]           % 13482 events
markers: [13481x4 char]             % each with four markers...
real: [13481x1 single]              % ...and one single precision value in this
                                     case
```

TextMark data

Text mark data allows a string of characters to be attached to a marker event.

```
[data {,header}]=SONGetChannel(fid, chan);
```

returns a *data* structure containing e.g.:

```
timings: [41x1 double]
markers: [41x4 char]
text: [41x80 char]
```

The length of the strings in *data.text* is variable and determined when reading the SON file.

```
title: 'Comment'
transpose: 0
```

CONVERTING DATA

Wave data

Routines are included that convert between 16-bit integer, and single and double precision floating point representations.

- **Converting to floating point.**

```
[dataout {,headerout}]=SONADCToSingle(datain {,headerin});
```

and

```
[dataout {,headerout}]=SONADCToDouble(datain {,headerin});
```

convert 16-bit integer data to single and double precision floating point respectively. If present, the values of *headerin.scale* and *headerin.offset* are used scale *dataout* into units of *header.units*.

headerin is copied to *headerout* and values of the minimum and maximum values in *dataout* are added as fields to *headerout* as with RealWave data in the SON system.

Output is assigned a channel type of 9 corresponding to RealWave data in SON regardless of whether *dataout* is single or double precision.

(Note: if *scale* is 1 and *offset* is 0 there is no advantage to these routines over the MATLAB single or double conversions).

- **Converting to 16-bit integer**

Converting from floating point to 16-bit integer is more difficult. Data needs to be scaled and an offset applied to make maximum use of the 16 bits available. This is done with:

```
[dataout {,headerout}]=SONRealToADC(datain {,headerin});
```

datain can be single or double precision. Regression is used to find values of *headerout.scale* and *headerout.offset* and *dataout* is a newly scaled version of *datain*.

dataout is rounded to the nearest integer and always fills the range -32768 to 32767.

Expect a quantization error of about $(10 \cdot \text{header.scale}) / (2 \cdot 65536)$ i.e $\pm 1/2$ LSB.

dataout and *headerout* can be written as a new ADC channel using SONCreateChannel.

Scale and offset will be converted to single precision on disk.

- **Event and marker data**

To convert between event or marker times in clock ticks and second.

```
dataout=SONTicksToSeconds(fid, datain, option);
```

option is a string or cell array of strings. Valid options are ticks (does nothing), microseconds, milliseconds or seconds (default).

MISCELLANEOUS

F=SONFileHeader(fid);

Returns a MATLAB structure containing the SON file header. See CED SON documentation for details.

L=SONChanList(fid)

Returns a MATLAB structure array with one element for each active channel in the SON file. Each element is a structure giving some details about the channel:

number: 1 % the channel number as it would appear in Spike2
kind: 1 % channel type
title: 'Sinewave'
comment: 'sinewave'
phyChan: 0 % physical port if appropriate

I=SONGetChannelInfo(fid,chan);

Returns the disk SON channel header as a MATLAB structure. See CED SON documentation for details.

B=SONGetBlockHeaders(fid,chan)

Returns the disk block headers as a MATLAB matrix. See CED SON documentation for details but note that *B* will contain a pointer to the start of each block not to the previous and succeeding block e.g.

B =
5120 % disk pointer to start of block
100 % start time (clock ticks)
2549100 % end time (clock ticks)
1 % channel number
2550 % number of data points

SONGetSampleInterval and SONGetSampleTicks

[interval {,start}]=SONGetSampleInterval(fid, chan)

or

[interval {,start}]=SONGetSampleTicks(fid,chan)

return the sample interval and the time of the first sample in seconds and clock ticks respectively for channel *chan* in file *fid*.

SONTest('filename') or SONTest(fid)

Runs the CED SONFix.exe test utility on the file. If filename is e.g. 'c:\data*.smr' all files in the directory are tested. Files that are open for writing (in MATLAB or e.g. Spike2) will not be tested. (Note: always leave SONFix using its Exit button).

SONVersion

Displays a box with the version number of this SON library and the SON versions it supports

USING THE SON32 LIBRARY

To use this library you need a copy of son32.dll (supplied as part of Spike2 for Windows). This file has to be in the c:\Spike5 folder.

The 'Additional Documentation' included with Spike2 describes the functions contained in son32.dll.

To access these function using the SON32 library

1. Load the library by running SONLoad
2. Open the file using SONOpenOldFile or SONCreateFile (not fopen)

Most of the functions in the dll are called using the MATLAB built-in calllib. If you use calllib directly, you must pre-allocate the output arrays. If this is not done properly, segmentation errors will occur and MATLAB will crash and/or become unstable. The supplied m-files take care of memory allocation for you.

Some functions can not be called with calllib as it can be used only with scalar structures. Gateway functions have been written in C to call these functions (the compiled files have the mex32 file extension).

. The main difference between the MATLAB calls and those made directly from C is that markers are returned as a structure of arrays instead of as an array of structures. This allows the arrays to be used as arguments to the MATLAB built-in functions more easily. The online help provides further details.

To unload the library (and free memory) call unloadlibrary('son32').

Pitfalls:

Remember that SONOpenOldFile (and SONCreateFile) return handles that are internal to son32.dll. They are not valid MATLAB file handles.

Channels are numbered from 0 to maxchan-1 in the SON32 library (not 1 to maxchan as with the SON2 library).