
# CEDS64ML interface library

# CEDS64ML interface library

## MATLAB® interface to the 64-bit SON library as a manual

*by Cambridge Electronic Design Limited*

# CEDS64ML interface library



Printed: June 2017 in Cambridge, England

## Revision History

Version 1.00 May 2014

Published by:

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

Telephone: Cambridge (01223) 420186
International: +44 1223 420186
Fax: Cambridge (01223) 420488
International Fax: +44 1223 420488
Email: info@ced.co.uk
Home page: www.ced.co.uk

## Acknowledgements

# Table of Contents

# 1: Introduction

# Introduction

This documentation describes a MATLAB® interface to the CED 64-bit Son library, as used by Spike2 version 8 and later versions. This library can read and create both the older format 32-bit Son files (`smr` file extension) and the newer 64-bit files (`smrx` file extension).

This interface does not give you access to the entire API of the CED 64-bit filing system, but it should be sufficient for any reasonable use from MATLAB®. In some cases, particularly when dealing with Marker and extended Marker channels, we have to manipulate/transform data to make it conform to the expectations of MATLAB®. This can degrade performance. The MATLAB® code interfaces to the library through a C language wrapper around the underlying SON64 library (which is written in C++). This C wrapper can be used from C and other languages that can interface to C to read and write CED SON format files.

## A quick overview of SON files

### Data file general properties

A SON data file is a container for multiple channels of data. Each channel of data holds multiple items, all of the same type and stored in ascending time order. Time in a SON file is measured in indivisible time units, referred to as ticks. Every item in the file has a time, which is a 64-bit integer number of ticks. The file has a timebase, which is the length of each tick, and this timebase is typically 0.000001 seconds (1 microsecond), but can be set to any value (within reason), but will likely be set in the range 1e-9 to 1e-3 seconds (1 nanosecond to 1 millisecond) per tick.

### Channels

The minimum number of channels in a file is 32, the maximum number is currently limited to 1000, but this is an arbitrary limit and the file structure would allow up to 65535 channels. However, if you want to be compatible with Spike2, you should limit files to 400 channels. When you create a new file you declare the number of channels and file header space is allocated for each channel. You can then use as many or as few of these channels as you desire. There is currently no mechanism to increase the number of channels beyond the number declared when the file is created. The file structure would allow us to do this but the currently defined programming interface does not support it.

Channels are identified by a channel number. Within Spike2 and in this library, the first channel is 1. If you read the low-level SON library documentation for users who program in C++, channels there start at 0; the glue code that links MATLAB® (and Spike2) to the low-level library deals with subtracting 1 or adding 1 to channel numbers, as required.

There are two basic channel types:

#### Waveform channels

Waveform channels hold data items that occur at fixed tick intervals, for example an ECG waveform sampled at 500 Hz. The sample interval is a fixed, integral multiple of the file tick. If the file tick were 1 microsecond, the available sample rates in the file would be 1000000/n where n is know as the channel divider. To achieve 500 Hz, with a 1 microsecond tick, the divider would be 2000.

Waveform data need not be continuous. It is stored in the file as data blocks with a start time (in ticks), followed by continuous data items. If we had a channel sampled from time 0 at 500 Hz with a gap at 10 seconds of 1 second, this would be stored as a block of 5000 items starting at tick 0 followed by a block starting at 11 seconds (11000000 ticks with a time resolution of 1 microsecond).

Waveform data blocks may not overlap in time. Further, they are usually written so that if the first data item is at tick `Tfirst`, all subsequent data blocks start at times that can be represented as `Tfirst + n * ChanDivide` ticks.

Currently, there are two types of waveform channels: Adc and RealWave.

##### *Adc channel type (code 1)*

These channels are designed to be efficient in data file space and store the data as 16-bit signed integers. You can set a `scale` and an `offset` value to convert these values into real, user units:

user value = (16-bit value) * `scale` /6553.6 + `offset`

With a scale of 1.0 and an offset of 0.0, the user values span the range -5.0 to 4.99985 user units. This scaling came about because the first 16-bit ADC (Analogue to Digital Converter) we used spanned a range of -5 to 4.99985 Volts.

You can choose to read these channels as either 16-bit integer values or as 32-bit floating point values (converted with the `scale` and `offset`). You write these channels as 16-bit integers.

*RealWave channel type (code 9)*
These channels store data as 32-bit floating point values. They also have a `scale` and `offset` that is used if you want to read the data back as 16-bit integers.

16-bit value = (32-bit floating point value - `offset`) * 6553.6/`scale`

If the result would exceed the range -32768 to 32767, it is limited to the range.

### Event-based channels

Event-based channels hold items that have a time in ticks (a 64-bit quantity). The simplest event channel types have no other information. The more complex types have a set of 4 8-bit marker codes and a 32-bit value attached to them (currently the 32-bit value is reserved for future use). These Marker types allow the data to be filtered on the codes; for example, read all data with specific codes in a time range. The full range of event-based channels currently defined is:

*EventFall (code 2) and EventRise (code 3)*
These types are essentially identical and interchangeable. The names come about because they are timed by either a rising edge or a falling edge of a digital input. They are stored as 64-bit tick counts.

*EventBoth (code 4)*
This type is used to record both edges of a digital input. In the original SON library (with 32-bit tick counts), this was stored as if it were EventFall or EventRise data, and the direction of each edge was deduced by counting edges from the start of each data block. In the 64-bit SON library, this type is implemented using marker codes to indicate the rising and falling edges. This occupies twice as much space, but has the advantage of making it easy to select just falling or rising edges by setting a filter on the codes.

*Marker (code 5)*
A Marker type is a tick count plus 4 marker codes as described above. Marker data is manipulated in MATLAB® using `CEDMarker` objects.

*WaveMark (code 6)*
Each data item is a Marker, followed by 16-bit waveform data. This is typically used to record nerve spikes. The attached 16-bit data can be scaled into user units as for Adc channel data. There can also be more than one channel of 16-bit data (Spike2 supports 1, 2 or 4 channels of data), in which case the data is interleaved (consecutive values cycle round the channels). The waveform sample interval per (interleaved) channel is set by the channel divide, as for an Adc channel. WaveMark data is manipulated in MATLAB® using `CEDWaveMark` objects.

*RealMark (code 7)*
Each data item is a Marker, followed by a list of 32-bit floating point values. RealMark data is manipulate in MATLAB® using `CEDRealMark` objects.

*TextMark (code 8)*
Each data item is a Marker, followed by a text string. TextMark data is manipulated in MATLAB® using `CEDTextMark` objects.

## Writing data

The library has facilities for efficient data writing, expecting to be doing this in real time. However, most use from MATLAB® will be offline to write processed data to a file, or to export data from MATLAB® for use by Spike2.

### Reading data

The library is optimised to read data efficiently (even from very large files) using time ranges as the key to locating data. A typical read routine requests data starting at or after a start time, up to but not including an end time with a limit on the number of items returned. When reading from a Marker-based channel you can also apply a marker filter (only items that match the filter). You can also read any Event-based channel as if it were event data (just times), or any Marker-based channel as if it were a Marker (ignoring the additional attached data).

### SON file versions

There are two major versions of the SON library: the original version used by Spike2 versions up to 7 that uses 32-bit times and the new version used by Spike2 version 8 onwards that uses 64-bit times. The new library can read and write old format files with the same software interface as the new format. However, there is the obvious limitation that tick ranges are limited to 32-bits.

## SON file versions

Spike2 data is stored in files with the extensions `.smr` (for the original 32-bit times format) and `.smrx` (for the 64-bit times format). The 32-bit file format was designed around 1987 and has been much extended since. Every son file contains a version code. The version code changes each time we make a change to the file format that would prevent older software from reading it correctly.

There were 9 major version changes to the 32-bit format; we are still on the first version of the 64-bit format. We have always made sure that more modern versions of Spike2 will read older file versions. Unless you are interested in history, or you have an old version of Spike2 and cannot read a data file, you can stop reading this page now.

If you are interested in the technical details of the library, or need to interface to it, there is documentation and header files on the CED web site (follow links to **Downloads** and then **Son library**). The `CEDS64Version()` command returns the version of the file.

| Version | Change history |
|---|---|
| 1 | The original SON filing system, written in Pascal, supported waveform and `Event` data only. |
| 2 | New `Marker` data type. Added extra space to the file header for future expansion. The library was much faster reading large data files as it remembered the last accessed data. |
| 3 | Added the `FilterMarker` function. Changed the meaning of the waveform channel divide to prevent an apparent change of sampling rate if a waveform channel was added or deleted. |
| 4 | Added the WaveMark data type. Changed the use of `FilterMarker`. The library now caches the current data block to avoid re-reading when the required block is already in memory. |
| 5 | Added the `TextMark` and `RealMark` data types and the `MaxTime` and `ChanMaxTime` functions. C library versions written for the IBM PC in DOS and Windows and for the Macintosh. The version 5 libraries write version 3 and 4 files if the data does not need new features.<br><br>In 1998, we extended the C version to support read-only files and to allow more data to be written per call. We added lookup tables to speed up data access in long files and write buffering to remove the need for `FastWrite`, allow peri-triggered sampling, and speed up writing). |
| 6 | Documented for C/C++ use only with information on use as a DLL by other languages. Added the `RealWave` channel type. `AdcMark` channels have multiple traces. Waveform rate divider is now 32-bits. Added support for time and date stamp, basic time unit and an application identifier. Files still compatible with versions 3, 4 or 5 if they do not use version 6 features. |
| 7 | You can choose to round the sizes of `AdcMark` and `TextMark` extended marker types up to a multiple of 4 bytes so that we can build the library on systems that insist on aligned data access. |
| 8 | Altered storage of channel numbers to avoid the bit used to hold the initial state for a level channel, allowing the maximum number of channels to be increased to 451. Extended the lookup table so that the table for a channel starts small and grows up to a limit (larger than the old fixed size). |
| 9 | Added support for big files (up to 1 TB), rewrote the internal lookup table system to improve speed and saves the table as part of the file in Big file mode. Macintosh (big-endian format) support was |

removed. Linux support is added. In a version 9 file, pointers to disk space that were previously byte offsets (but multiples of 512) are now block pointers (a pointer value 10 means offset 5120). This increases the maximum file size by a factor of 512. There are changes to the file header to support a lookup table on disk and to channel headers to track the block counts. There were also type changes to allow compiling as 64-bit code on Windows and Linux.

256 0x100 **64-bit times**. A complete redesign of the filing system to remove the time and disk space limitations of the original while leaving a system that is similar enough to the original to store the same data and behave in similar ways. Times are stored as 64-bits and the files can theoretically be of any size up to 2 to the power 64 bytes (around 10 to the power 19 bytes).

Practical considerations, such as the need to copy files in reasonable time will limit file sizes to a small number of TB for the next few years. Times are stored to 64-bit accuracy, but some operations within Spike2 use floating point numbers to manipulate file times, and a floating point number has some 53 bits of precision, so this also limits the available maximum time. However, with 1 microsecond ticks, you can still sample for tens of years before this becomes an issue.

You can access both 32-bit and 64-bit files using the same API, so although there is software effort required to use the new format, once done you can read both old and new formats with the same code. Also, if the son64.dll file is placed in the same folder as a recent son32.dll file, old code that reads old 32-bit files can also read a new file (as far as the first 32-bits of clock ticks).

## What we supply

The set of files contained in the MATSON installer allow you to open .smr and .smrx files (or create new files) in MATLAB® and then read and write data to those files.

When correctly installed you will find the following inside the target folder:

The compiled help file `ceds64int.chm`.

The folder `CEDS64ML`, which should contain:

57 '.m' MATLAB function files of the form `CEDS64AbcXyz.m`

4 MATLAB class files (`CEDMarker.m`, `CEDRealMark.m`, `CEDTextMark.m`, `CEDWaveMark.m`)

a folder `x86` holding `ceds64int.dll`, `ceds64int.h` and `son64.dll`. This code is for 32-bit versions of MATLAB®. The folder also holds the `.lib` files needed to link to the DLLs from languages other than MATLAB.

a folder `x64` holding `ceds64int.dll`, `ceds64int.h` and `son64.dll`. This code is for 64-bit versions of MATLAB®. The folder also holds the `.lib` files needed to link to the DLLs from languages other than MATLAB.

a folder `Data`, which should be empty and which is used by the examples to create data files

The folder `Examples` which should contain seven `.m` MATLAB files: `ExampleCopy`, `ExampleCreateFile`, `ExampleDownSample`, `ExampleFFT`, `ExampleFilter`, `ExampleMask` and `ExamplePeakFind`. These are basic MATLAB script files demonstrating most of the functionality of the ceds64int library.

Before you start your own work it is recommended you run these scripts (running `ExampleCreate` first) to ensure that the interface is working correctly on your machine. Note that the installer will have set the environment variable CEDS64ML to the path to the `CEDS64ML` folder. This allows our example scripts to locate the library files. If you have not used the installer (for example, you have copied another installation), you will need to set the environment variable by hand.

## Requirements and limitations

This interface is standalone, it does not require Spike2 to be installed on the same machine as MATLAB®. The library has been tested with MATLAB® R2013b (32 and 64-bit), while we cannot be sure that it is compatible with earlier versions, it does not use any recent MATLAB® features, so it should work with all 'reasonably' recent versions of MATLAB®.

This document does not cover the structure of SON file format (e.g. what information is stored in the file header, how the files are saved to disc). The library is intended to be user friendly and safe as possible, which has meant using two levels of abstraction, the `ceds64int` library and the `CEDS64AbcXyz()` MATLAB®

functions. The AbcXyz part of the function names are mainly taken from the underlying function names in the son library. For a more detailed explanation of the SON file structure you will need access to the documentation for the current son32 and son64 libraries (not provided here and not currently released). You do not need this information to use this library as documented here.

# Using the ceds64int library

You must tell MATLAB® where to find the CEDS64ML folder that contains the library. You can use it from the installation folder, or you can copy the CEDS64ML folder somewhere convenient. When using the CED code (either from the command window or a script) you will need to start the code with:

```
cedpath = 'c:\Your\Install\Path\CEDS64ML';
addpath( cedpath );
CEDS64LoadLib( cedpath );
```

This tells MATLAB® where the CED code is and loads the correct library DLL for the version of MATLAB® (32-bit or 64-bit). Then you can use any of the functions in the library. When you have finished, remember to call

```
unloadlibrary ceds64int;
```

to unload the DLL, although this is not strictly necessary as the DLL will be unloaded when MATLAB® closes.

However, having programs with fixed paths embedded in them is not very flexible, so we strongly recommend that you follow the practice of the example programs, and use an environment variable to locate the folder holding the library.

### Environment variable CEDS64ML

Rather than require you to edit them, our example programs expect you to have set the environment variable CEDS64ML to be the path to the folder. The MATSON installer will have set this environment variable for you when you installed the SON library support, so you only need set or edit the environment variable if you have moved or renamed the folder. This can be done as follows:

Locate the System Properties dialog, click the Environment Variables button, then set a new User variable (to set it for a particular user) or set a new System variable to make it accessible to everyone on the machine.

**Windows 7**: to locate this dialog I opened the start menu, selected Computer, System properties, then Advanced System Settings.

**Windows 10**: Type the word 'Environment' (without the quote marks) into the *Ask me anything* box next to the Start menu, and the system will suggest "Edit the environment variables" in the Control Panel, which is what you want.

Once you have the environment variable set to the path to the folder, you can then use:

```
cedpath = getenv('CEDS64ML');  % should hold the path to the CEDS64ML folder
addpath( cedpath );
CEDS64LoadLib( cedpath );
```

If the system environment does not hold the CEDS64 environment variable, you can use the MATLAB® setenv function to set the path for the duration of the session. However, this only sets environment variables in the copy of the environment that is passed to MATLAB® when it is run; the value will be lost after MATLAB® exits. You would do something like:

```
setenv('CEDS64ML', 'c:\Your\Install\Path\CEDS64ML');
```

### Direct calls to the interface DLL

Advanced users may be tempted to call methods directly from the ceds64int library using the calllib command, without going through the CEDS64AbcXyz() functions.

However we strongly advise against this, as it will often require you to pass blocks of memory between MATLAB® and the library. If any of these blocks are the wrong sizes it can crash the program, and potentially corrupt your data. The CEDS64AbcXyz() functions calculate the required size of the memory blocks and should be safe. Further, we may wish to modify the details of the DLL interface; if you make direct calls your code could be broken if we make any changes.

# Writing a script with the ceds64int library

In this section we explain how to write a simple script that opens an existing file, reads data from it, filters it and writes it to a new file. Start by creating a new MATLAB® script, it doesn't matter where.

## Initialise the library

Add the following lines of code:

```
clear;                                  % clear workspace

% add the path to the CED code (choose one of the two methods)
cedpath = 'c:\your\path\to\CEDS64ML'; % if no environment variable (NOT recommended)
cedpath = getenv('CEDS64ML');          % if you have set this up (this is the recommende

addpath( cedpath );                     % so CEDS64LoadLib.m is found
CEDS64LoadLib( cedpath );               % load ceds64int.dll
```

You must set `cedpath` to wherever the `CEDS64ML` folder is located. This code tells the script where to find the CED code and loads the correct DLL for your build of MATLAB®. The MATSON installer will have set the `CEDS64ML` environment variable for you.

## Open a file to read

Then add:

```
% Open a file
fhand1 = CEDS64Open( 'C:\Spike8\demo.smr' );
if (fhand1 <= 0); unloadlibrary ceds64int; return; end
```

This attempts to open the file `demo.smr` in `C:\Spike8` (change this to the relevant directory and or file) and returns an integer file handle (`fhand1`) which allows us to access the file. At the moment there are a limited number of file handles (100) and therefore a limited number of files that can be open at once. If it fails to open the file, `fhand1` will be negative, at which point there is no point in continuing so we end the script and unload the DLL. Most `CEDS64AbcXyz()` functions return negative error messages if the they fail, which allows you to terminate the program before an error and work out exactly what has gone wrong.

## Read some waveform data

Add the following:

```
% get waveform data from channel 1
maxTimeTicks = CEDS64ChanMaxTime( fhand1, 1 )+1;    % +1 so the read gets the last poin
[ fRead, fVals, fTime ] = CEDS64ReadWaveF( fhand1, 1, 100000, 0, maxTimeTicks );
```

The first line gets the time in ticks (not seconds!) of the final item in channel 1 of the file with handle `fhand1` (`demo.smr`). The second line reads the waveform data from this channel as a vector of 32-bit floats. `CEDS64ReadWaveF()` takes as inputs: a file handle, a channel number, the maximum number of points you want to read, the start time and the end time (both in ticks) (it is worth pointing out, almost all the times in `CEDS64AbcXyz()` commands are given as 64-bit ticks rather than seconds, there are two functions `CEDS64SecsToTicks()` and `CEDS64TicksToSecs()` to allow you to convert between ticks and seconds).

The function returns (in order from left to right): `fRead`, the number of items actually read, this will never be more the number you asked for but could be less, if it is negative there has been an error; `fVals`, a vector of 32-bit floats containing the value of the items; `fTime` the time of the first item in ticks.

## Process the data

Now add the following code to apply a MATLAB® filter to the read data:

```
% Set up filter parameters, see MATLAB documentation for more details
a = 1;
b = [1/10, 1/10, 1/10, 1/10, 1/10, 1/10, 1/10, 1/10, 1/10, 1/10];

% Apply the filter to fVals and save output as FiltVals
FiltVals = filter(b, a, fVals);
```

### Create an output file

Now we create a new file to save the filtered data:

```
% create a new file
fhand2 = CEDS64Create( 'c:\Spike8\Data\ExampleFilter.smrx', 32, 2 );
if (fhand2 <= 0); unloadlibrary ceds64int; return; end
```

This attempts to create a file `ExampleFilter.smrx` in `c:\Spike8\Data`, with room for 32 channels. The final input argument '2' tells the function to create a 64-bit `.smrx` file rather than a 32-bit `.smr` file. This is strictly redundant as we have already specified the file type in the name and you can omit this argument if you wish. By default, the new file is created with a time resolution (duration of each tick) of 1 microsecond. We need to change this to match the source file, otherwise when we set the output channel sample rate it will not match the source.

```
tbase = CEDS64TimeBase( fhand1 );
CEDS64TimeBase( fhand2, tbase );
```

### Create a suitable output channel in the new file

```
% set the chan divide and ideal rates to be the same as the original file
chandiv = CEDS64ChanDiv( fhand1, 1 );
rate = CEDS64IdealRate( fhand1, 1 );
```

and use it to create a RealWave channel in `ExampleFilter.smrx`

```
% create a new real wave channel
CEDS64SetWaveChan( fhand2, 1, chandiv, 9, rate );
```

This sets channel 1 in `ExampleFilter.smrx` to a RealWave channel with the same ideal rate and divide rate as channel 1 in `demo.smr`.

### Write data to the output file

Now we write the filtered data into channel 1 of `ExampleFilter.smrx` using:

```
% write filtered data to new channel
CEDS64WriteWave( fhand2, 1, FiltVals, 0 );
```

### Close all and tidy up

Finish off by closing both files and unload the DLL.

```
CEDS64CloseAll();          % close all the files
unloadlibrary ceds64int;   % unload ceds64int.dll
```

## Licence

A portion of the library is implemented using MATLAB® code. We have released this code under the GPL with the following licence statement:

```
Copyright (C) Cambridge Electronic Design Limited 2014
Authors: James Thompson, Greg Smith
Web: www.ced.co.uk email: softhelp@ced.co.uk

This file is part of CEDS64ML, an interface to the 64-bit SON data library.
CEDS64ML is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

CEDS64ML is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with CEDS64ML. If not, see <http://www.gnu.org/licenses/>.
```

## Support

This library is provided to you free of charge, and we make no undertaking to support your use of it. If you find a problem with the library that you think is a bug in our code, we will be pleased to hear about it and will endeavour to fix it. If the bug is in the MATLAB® interface, you may wish to send us a suggested fix. We are also happy to receive suggestions for improvements.

However, we do not undertake to debug your MATLAB® program. When reporting a potential bug, please reduce your problem to the smallest script that illustrates the defect. It is usually possible to do this with 10 lines or so of code. If the problem requires a data file and it cannot be demonstrated with a standard data file that we supply with Spike2, then either create the data file with your script or send us the file. When sending us a file, please be considerate. Most problems can be demonstrated with relatively small files. Spike2 includes facilities to export channels and time ranges from large files to make smaller ones.

# 2: Library calls by function

# Library calls by function

This section divides the library functions up into related groups.

## File operations

These are the library functions that deal with the data file rather than individual channels.

| | |
|---|---|
| CEDS64Open | Open an existing file and get the handle |
| CEDS64Create | Create a new file and get the handle |
| CEDS64Close | Close a nominated file |
| CEDS64CloseAll | Close all open files |
| CEDS64EmptyFile | Remove all channels and data, but preserve header information |
| CEDS64FileCount | Count the number of files that are open |
| CEDS64FileSize | Get the size of a file on disk |
| CEDS64FileComment | Get and set file comments |
| CEDS64TimeBase | Get and set the time base (seconds per tick) value for the file |
| CEDS64IsOpen | Test if a file handle refers to an open file |
| CEDS64MaxChan | Get the number of channels this file supports |
| CEDS64MaxTime | Get the time of the last data item in the file |
| CEDS64SecsToTicks | Convert seconds to ticks |
| CEDS64TicksToSecs | Convert ticks to seconds |
| CEDS64TimeDate | Get and set the time and date at which sampling started |
| CEDS64Version | Get file version (detect 32-bit file or 64-bit file) |

## Channel operations

These are the library functions that deal with channels. Operations that deal with specific channel types are delat with separately.

| | |
|---|---|
| CEDS64ChanComment | Get or set a channel comment |
| CEDS64ChanDelete | Delete a channel |
| CEDS64ChanDiv | Get the waveform rate divisor (determines the waveform rate) |
| CEDS64ChanMaxTime | Get last item time in a channel |
| CEDS64ChanOffset | Get or set the wave scaling offset value |
| CEDS64ChanScale | Get or set the wave scaling scale value |
| CEDS64ChanTitle | Get or set the channel title |
| CEDS64ChanType | Get the type of a channel |
| CEDS64ChanUnits | Get or set the channel units |
| CEDS64ChanUndelete | Undelete a channel that has been deleted but not reused |
| CEDS64ChanYRange | Get or set two values that can be used for a channel range |
| CEDS64EditMarker | Modify data attached to a Marker or extended Marker channel |
| CEDS64GetExtMarkInfo | Get information from an extended Marker channel |

| | |
|---|---|
| CEDS64GetFreeChan | Get the next unused channel number in the file |
| CEDS64IdealRate | Get or set the ideal waveform rate/expected event rate for a channel |
| CEDS64MaxChan | Get the number of channels allowed in the file |
| CEDS64PrevNTime | Search back N items on a channel |

## Waveform channels

These functions are used with waveform channels

| | |
|---|---|
| CEDS64ChanDiv | Get the sample interval in file tick units |
| CEDS64ChanOffset | Get or set the offset for scaling |
| CEDS64ChanScale | Get or set the offset for scaling |
| CEDS64ChanUnits | Get or set the units string |
| CEDS64IdealRate | Get or set the ideal channel rate (for information only) |
| CEDS64ReadWaveS | Read waveform data as 16-bit integers |
| CEDS64ReadWaveF | Read waveform data as 32-bit floating point values |
| CEDS64SetWaveChan | Create a new waveform channel |
| CEDS64WriteWave | Write waveform data |

## Event channels

Functions for EventFall and EventRise channels.

| | |
|---|---|
| CEDS64IdealRate | Get or set the expected sustained maximum rate |
| CEDS64PrevNTime | Search backwards N events |
| CEDS64ReadEvents | Read event times as ticks |
| CEDS64SetEventChan | Create a new event channel |
| CEDS64WriteEvents | Write event times |

## EventBoth channels (Level event)

In a 64-bit file, EventBoth data is stored as Markers using the first marker code to decide if the events are low or high with 0 meaning low and not zero meaning high.

| | |
|---|---|
| CEDS64PrevNTime | Search backwards N events with an optional marker mask |
| CEDS64ReadEvents | Read data as event times with an optional marker mask |
| CEDS64ReadLevels | Read data as event times with an optional marker mask and get the level of the first event read |
| CEDS64ReadMarkers | Read EventBoth data as Markers with an optional marker mask |
| CEDS64SetInitLevel | Set initial level before any data is written for a new channel |
| CEDS64SetLevelChan | Create a new EventBoth channel |
| CEDS64WriteLevels | Write EventBoth data |
| CEDS64WriteMarkers | Write data, taking responsibility for sequence of levels |

## Marker channels

Functions used for Marker channels.

| | |
|---|---|
| CEDS64EditMarker | Modify the codes attached to a marker in a channel of a file |
| CEDS64IdealRate | Get the expected sustained maximum rate for this channel |
| CEDS64PrevNTime | Search backwards N events with an optional marker mask |
| CEDS64ReadEvents | Read a marker channel as events with an optional marker mask |
| CEDS64ReadMarkers | Read a marker channel as Markers with an optional marker mask |
| CEDS64SetMarkerChan | Create a Marker channel |
| CEDS64WriteMarkers | Write marker data to a channel |

## Extended markers

Functions for generalised extended markers (RealMark, WaveMark, TextMark):

| | |
|---|---|
| CEDS64EditMarker | Edit the data attached to a Marker or extended Marker in a file |
| CEDS64GetExtMarkInfo | Get information about the extended marker channel |
| CEDS64PrevNTime | Search back N events with an optional marker mask |
| CEDS64ReadEvents | Read extended markers as event times with an optional marker mask |
| CEDS64ReadExtMarks | Read extended markers as their native type with an optional marker mask |
| CEDS64ReadMarkers | Read extended markers as markers with an optional marker mask |
| CEDS64SetExtMarkChan | Create an extended marker channel of any type |
| CEDS64SetTextMarkChan | Create a TextMark channel. |
| CEDS64WriteExtMarks | Write extended marker data |

## Marker masks

The SON library supports the concept of marker masks that can be used to filter Marker and Extended Marker data. A marker mask has three components: a 256 x 4 matrix indicating which marker codes are acceptable, a mode or use of the mask and a column selector for use with extended marker types with interleaved data (currently multi-trace WaveMark data). The ceds64int library provides 1000 of them that are manipulated through library functions:

| | |
|---|---|
| CEDS64ResetMask | Reset one or all marker masks to be empty and mode 0. |
| CEDS64MaskCodes | Exchange a marker mask between the internal format and a 256 x 4 numeric matrix. |
| CEDS64MaskMode | Get and/or set the mode of a mask. |
| CEDS64MaskCol | Get and/or set the WaveMark trace (column) to return when reading waveform data. |

### Masks

Marker masks are identified by a mask handle, which is an integer; the first mask is number 1. There is an overall limit on the number of masks that can exist that is set in the library (currently 1000). It is up to you to track your marker mask usage. Each time you refer to a new marker mask number, an empty mask is created. Most functions that use an optional mask handle will allow you to omit the handle (or use a handle number of

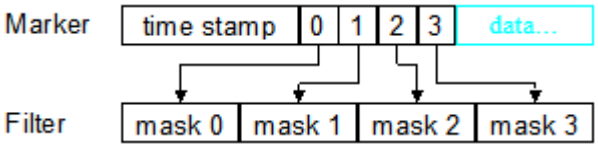
0) to not use a mask (that is, to accept all data). Some actions, such as CEDS64PrevNTime(), are significantly slower when a marker mask is used.

A Marker or extended Marker data item has a time stamp and 4 marker codes, each of which is an uint8 value (in the range 0 to 255). The purpose of a marker mask is to filter data items so that when reading you can collect only the data that matches some criterion. Marker masks are implemented in MATLAB® as a 256 x 4 matrix of uint8 data. For each of the 4 marker codes there are 256 values indicating which code values are acceptable to the mask; a zero in the mask means not acceptable, non-zero means acceptable.

A marker mask can also be operated in two modes: AND or OR:

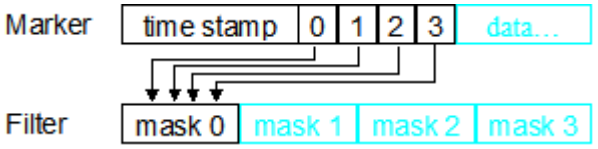
*Mode 0 (AND)*

A marker data item is allowed through the mask if each of the four codes in the data item is present in the corresponding mask. We think of this as *and* mode because for the filter to pass the marker, marker code 0 must be in mask 0 *and* marker code 1 in mask 1 *and* marker code 2 in mask 2 *and* marker code 3 in mask 3. In this mode, masks 1, 2 and 3 are usually set to accept all codes and masking is used for layer 0.




*Mode 1 (OR)*

A marker data item is allowed through the mask if any of the four marker codes is present in mask 0. A code 00 is only accepted for the first of the four marker codes. Masks 1 to 3 are ignored. We think of this as *or* mode because marker code 0 *or* 1 *or* 2 *or* 3 must be present in mask 0 for the filter to pass the marker for display or analysis. This mode can be used with spike shape data (WaveMark) where two spikes have collided and the marker represents a spike of two different templates.




**Column**

The underlying SON64 library allows extended Marker data types to have a grid of data attached to them. This feature is currently only used for the WaveMark data type, which can have interleaved waveform traces attached. The library allows you to read waveform data back from WaveMark channels, and if you do so with a multi-trace channel, by default you get the first trace. By setting a Marker Mask you can select which trace is returned with CEDS64MaskCol().

# Marker mask examples

This example creates a mask that only lets through markers that have at least one code set to 1, 2, 3, or 163 (in hex 01, 02, 03, A3). As we do not care which of the 4 marker codes has the value we will be in OR mode. the variable maskh is assumed to be set to one of the available marker handles (1-1000).

```
CEDS64MaskMode(maskh, 1); % set to OR mode
mask = zeros(256, 4, 'uint8'); % create a 256x4 matrix of zeroed 8-bit integers

% Both indices are offset by 1 to accommodate MATLAB® indexing conventions
mask(1+1, 0+1) = 1;      % this includes code 1 in layer 0.
mask(2+1, 0+1) = 1;      % this includes code 2 in layer 0.
mask(3+1, 0+1) = 1;      % this includes code 3 in layer 0.
mask(163+1, 0+1) = 1;    % this includes code 163 in layer 0.
CEDS64MaskCodes(maskh, mask);
```

You can now use maskh in a read function to filter the data.

```
% to use the mask include the mask handle in a read command e.g.
CEDS64ReadExtMarks( fhand, iChan, 100,  0, CEDS64SecsToTicks(15), maskh )
```

This example creates a mask that lets through markers that have the first marker code set to 01, and doesn't care about the remaining three codes, so the mask is set to accept all values for them. This is likely to be the most common sort of mask you will use.

```
CEDS64MaskMode(k, 0); % set to AND mode
mask = zeros(256, 4, 'uint8'); % 256x4 matrix of 8-bit integers set to zero
mask(2, 1) = 1;                % this includes code 1 in first layer
```

```
mask(1:end, 2:4) = 1;           % include all codes in remaining layers
CEDS64MaskCodes(k, mask);       % copy the codes to the mask
```

## Utility functions

Functions that do not fit the other categories:

| | |
|---|---|
| CEDS64ErrorMessage | Convert an error code into a message |
| CEDS64LoadLib | Load the relevant DLLs and initialise the library. Call this first. |

## Error codes

Many functions in the library return a negative code to indicate an error. You can convert these codes to a message with CEDS64ErrorMessage().

| | | |
|---|---|---|
| 0 | S64_OK | There was no error |
| -1 | NO_FILE | Attempt to use when file not open, or use of an invalid file handle, or no spare file handle. Some functions that return a time will return -1 to mean nothing found, so this is not necessarily an error. Check the function description. |
| -2 | NO_BLOCK | Failed to allocate a disk block when writing to the file. The disk is probably full, or there was a disk error. |
| -3 | CALL_AGAIN | This is a long operation, call again. If you see this, something has gone wrong as this if for internal SON library use, only. |
| -5 | NO_ACCESS | This operation was not allowed. Bad access (privilege violation), file in use by another process. |
| -8 | NO_MEMORY | Out of memory reading a 32-bit son file. |
| -9 | NO_CHANNEL | A channel does not exist. |
| -10 | CHANNEL_USED | Attempt to reuse a channel that already exists. |
| -11 | CHANNEL_TYPE | The channel cannot be used for this operation. |
| -12 | PAST_EOF | Read past the end of the file. This probably means that the file is damaged or that there was an internal library error. |
| -13 | WRONG_FILE | Attempt to open wrong file type. This is not a SON file. |
| -14 | NO_EXTRA | A request to read user data is outside the extra data region. |
| -17 | BAD_READ | A read error (disk error) was detected. This is an operating system error. |
| -18 | BAD_WRITE | Something went wrong writing data. This is an operating system error. |
| -19 | CORRUPT_FILE | The file is bad or an attempt to write corrupted data. |
| -20 | PAST_SOF | An attempt was made to access data before the start of the file. This probably means that the file is damaged, or there was an internal library error. |
| -21 | READ_ONLY | Attempt to write to a read only file. |
| -22 | BAD_PARAM | A bad parameter to a call into the SON library. |
| -23 | OVER_WRITE | An attempt was made to over-write data when not allowed. |
| -24 | MORE_DATA | A file is bigger than the header says; maybe not closed correctly. Use S64Fix on it. |

# 3: Library functions

# Library functions

This section deals with all the library functions. To avoid repeating argument details many times, we start by describing common arguments:

`fhand` An integer that represents a file (read this as fileHandle). You get this value returned when you call `CEDS64Create()` or `CEDS64Open()`. Currently, the `ceds64int` library allows you to have up to 100 files open with handles in the range 1 to 100. This is an arbitrary limit and we may choose to increase it (or even remove it) in the future.

`iChan` An integer that identifies a channel in a file. The first channel in a file is 1, matching the channel numbers used by Spike2. Data files have at least 32 channels, but can have substantially more. Spike2 supports files with up to 400 data channels and the library allows up to 1000 (though this may be increased in the future).

`i64Xxxx` A 64-bit integer that specifies a time in ticks.

`i64UpTo` A 64-bit time that marks the end of a time range, usually for reading. The time range for valid data extends up to *but not including* this time. You can usually set this to -1, meaning the range extends to the end of the file.

`iOk` An integer that is returned to indicate success (0 or greater) or failure (a negative error code).

`maskh` An integer handle to a marker mask, used to filter data. This can usually be omitted or set to 0 for no marker mask.

### Optional items

In many cases, you can omit arguments to functions and return values. Such items are indicated by enclosing them in curly braces `{...}`. You never type the curly braces into your code. For example:

```
[ dSecs ] = CEDS64TimeBase( fhand{, dNew} )
```

In this case, you can omit the `dNew` arguments.

```
dTimeBase = CEDS64TimeBase( fhand );  % get the current time base
CEDS64TimeBase(fhand, dTimeBase*2.0); % change the time base
```

Sometimes, as here, omitting an argument changes what the function does. In other cases, omitting an argument gives you a default value. See the individual descriptions for details. Where there are multiple items that can be omitted, for example:

```
[ marker ] = CEDMarker({Time{, Code1{, Code2{, Code3{, Code4}}}}})
```

You must omit from the right, that is you cannot omit an argument in the middle of the list.

```
mark1 = CEDMarker();
mark2 = CEDMarker(1);
mark3 = CEDMarker(1,1);
mark4 = CEDMarker(1,1,2);
mark5 = CEDMarker(1,1,2,3);
mark6 = CEDMarker(1,1,2,3,4);
```

## CEDS64AppID

Each data file has space within it for an application ID. This is 8 bytes of space where you can store anything that will fit, but it will usually be ASCII characters that identify the entity that created the file. This is for information only and you can safely ignore it, if you wish.

```
[ iOk, vIDOut ] = CEDS64AppID( fhand {, IDIn } )
```

`fhand` An integer file handle to identify the file.

`IDin` Optional. If present the application ID field will be changed. This is either a string (in which case the first 8 characters will be used) or a vector of integers (which will be converted to `uint8`). If the passed in data is less than 8 character/values the space is made up with zeros.

`iOk` Returned as 0 if the operation completed without error, otherwise a negative error code.

vIDOut Returned as a vector of 8 unsigned 8-bit integers holding the application ID before any change made by IDin.

**Example**
The first two examples set the application ID using a string and then read it back. The illustrate what happens if you use strings that are less than 8 characters or greater than 8 characters in length. Note that the returned value is not a string.

```
EDS64AppID( fhand, 'AppID!' );          % sets a 6 character ID
[ iOk, val ]= CEDS64AppID( fhand )      % val is returned as:
%                             [ 65, 112, 112, 73, 68, 33, 0, 0 ]

CEDS64AppID( fhand, 'ID is too long' ); % sets 'ID is to'
[ iOk, AppId ]= CEDS64AppID( fhand );   % AppId is returned as:
%                         [ 73, 68, 32, 105, 115, 32, 116, 111 ]
```

This example sets the values using integers that should be in the range 0 to 255. If you set an integer greater that 255 the value 255 is stored. If you set a negative value, 0 is stored.

```
CEDS64AppID( fhand, [ 2, -6, 24, -120, 720, 0, 0, -4 ] );
[ iOk, AppId ]= CEDS64AppID( fhand );   % AppId is returned as:
%                             [ 2, 0, 24, 0, 255, 0, 0, 0 ]
```

# CEDS64ChanComment

This gets and/or sets the comment associated with a channel. The 32-bit SON library set a limit of 71 characters on the length of a channel comment, so we suggest that your comments are similarly pithy, even though the new 64-bit format sets no limit.

```
[ iOk {, sCom} ] = CEDS64ChanComment( fhand, iChan{, sNew} )
```

fhand An integer file handle to identify the file.

iChan The channel number (starting at 1).

sNew If present, a string to set the new channel comment. If omitted, no change is made.

iOk 0 if the operation completed without error, otherwise a negative error code.

sCom Returned as the original comment (before any change if sNew is provided).

# CEDS64ChanDelete

This function deletes a channel from a file. This deletes a channel from the file. In a 64-bit file, channels are deleted in such a way that as long as you do not reuse the channel, it is possible to undelete them using CEDS64ChanDelete().

```
[ iOk ] = CEDS64ChanDelete( fhand, iChan )
```

fhand An integer file handle to identify the file.

iChan The channel number to be deleted (starting at 1).

iOk 0 if the channel was deleted correctly, otherwise a negative error code.

# CEDS64ChanDiv

This gets the channel divide (sample interval in ticks) for waveform and WaveMark channels. If used with a channel that does not use a channel divide, the result will be 0.

```
[ i64Div ] = CEDS64ChanDiv( fhand, iChan )
```

fhand An integer file handle to identify the file.

iChan The channel number to be deleted (starting at 1).

i64Div The sample interval in ticks, otherwise a negative error code.

**Example**
Get the channel 1 rate in Hz (assuming that channel 1 is a waveform channel):

```
SampleRateInHz = 1.0/(CEDS64ChanDiv(fhand, 1)*CEDS64TimeBase(fhand));
```

## CEDS64ChanMaxTime

This function returns the time in ticks of the last item in a channel.

```
[ i64Time ] = CEDS64ChanMaxTime( fhand, iChan )
```

fhand An integer file handle to identify the file.

iChan The channel number (starting at 1).

i64Time The time of the last item in a channel, or -1 if no items exist, otherwise a negative error code.

## CEDS64ChanOffset

Gets and/or sets the offset used to convert between 16-bit values and user units for a channel for a channel. This is only relevant for Waveform, WaveMark and RealWave channels. However, it is not an error to use this with any channel type.

```
[ iOk {, dOffset} ] = CEDS64ChanOffset( fhand, iChan{, dNew} )
```

fhand An integer file handle to identify the file.

iChan The channel number to be deleted (starting at 1).

dNew An optional argument. if present, it sets a new channel offset value.

iOk 0 if the operation completed correctly, otherwise a negative error code.

dOffset If present, returned holding the channel offset before any change made by providing dNew.

## CEDS64ChanScale

Gets and/or sets the scale used to convert between 16-bit values and user units for a channel for a channel. This is only relevant for Waveform, WaveMark and RealWave channels. However, it is not an error to use this with any channel type.

```
[ iOk {, dScale} ] = CEDS64ChanScale( fhand, iChan{, dNew} )
```

fhand An integer file handle to identify the file.

iChan The channel number to be deleted (starting at 1).

dNew An optional argument. if present, it sets a new channel scale value.

iOk 0 if the operation completed correctly, otherwise a negative error code.

dScale If present, returned holding the channel scale before any change made by providing dNew.

## CEDS64ChanTitle

Gets and/or sets the channel title. The 64-bit SON library does not impose limits on the length of the title string. However, the older 32-bit library only allows 9 characters and Spike2 does not expect longer titles so we suggest that you keep them short.

```
[ iOk {, sTitle} ] = CEDS64ChanTitle( fhand, iChan{, sNew} )
```

fhand An integer file handle to identify the file.

| | |
|---|---|
| iChan | The channel number to be deleted (starting at 1). |
| sNew | An optional argument. if present, it sets a new channel title. |
| iOk | 0 if the operation completed correctly, otherwise a negative error code. |
| sTitle | If present, returned holding the channel title before any change made by providing sNew. |

## CEDS64ChanType

This returns an integer code indicating the type of a channel in a file.

```
[ iType ] = CEDS64ChanType( fhand, iChan )
```

| | |
|---|---|
| fhand | An integer file handle to identify the file. |
| iChan | The channel number (starting at 1). |
| iType | Either the channel type or a negative error code. Channel types are: 0=channel unused, 1=Adc, 2=EventFall, 3=EventRise, 4=EventBoth, 5=Marker, 6=WaveMark, 7=RealMark, 8=TextMark, 9=RealWave. |

## CEDS64ChanUnits

Gets and/or sets the channel units. The 64-bit SON library does not impose limits on the length of the units string. However, the older 32-bit library only allows 5 characters and Spike2 does not expect longer unit strings so we suggest that you keep them short.

```
[ iOk {, sUnits} ] = CEDS64ChanTitle( fhand, iChan{, sNew} )
```

| | |
|---|---|
| fhand | An integer file handle to identify the file. |
| iChan | The channel number to be deleted (starting at 1). |
| sNew | An optional argument. if present, it sets a new channel units string. |
| iOk | 0 if the operation completed correctly, otherwise a negative error code. |
| sUnits | If present, returned holding the channel title before any change made by providing sNew. |

## CEDS64ChanUndelete

This function attempts to undelete a channel. For this to suceed, the channel must be part of a 64-bit data file (not an old-style 32-bit file) and the channel must not have been reused since being deleted by CEDS64ChanDelete().

```
[ iOk ] = CEDS64ChanUndelete( fhand, iChan )
```

| | |
|---|---|
| fhand | An integer file handle to identify the file. |
| iChan | The channel number to be recovered (starting at 1). |
| iOk | 0 if the channel was recovered, otherwise a negative error code. |

## CEDS64ChanYRange

You can store and retrieve two floating point values for each channel that indicate suggested low and high values for displaying the channel, or that can be used for any other purpose you choose. These might be useful with very large files, where scanning all the data for the maximum and minimum value could take a very long time. Spike2 does not make use of these values, and they are initialised to 0 when a channel is created.

```
[ iOk{, dLo, dHi} ] = CEDS64ChanYRange( fhand, ichan{, dNewLo, dNewHi} )
```

| | |
|---|---|
| fhand | An integer file handle to identify the file. |

| | |
|---|---|
| iChan | The channel number (starting at 1). |
| dNewLo | If present, the new channel low value. If omitted, no change is made. |
| dNewHi | If present, the new channel high value. If omitted, no change is made. |
| iOk | 0 if the operation completed without error, otherwise a negative error code. |
| dLo | If present, returned as the original low value before any change made by dNewLo. |
| dHi | If present, returned as the original high value before any change made by dNewHi. |

## CEDS64Close

Close the file with a nominated file handle that was returned from CEDS64Create() or CEDS64Open().

```
[ iOk ] = CEDS64Close( fhand )
```

| | |
|---|---|
| fhand | An integer file handle to identify the file. |
| iOk | 0 if the file was closed, otherwise a negative error code. |

## CEDS64CloseAll

Close all open SON files.

```
[ iOk ] = CEDS64CloseAll( )
```

| | |
|---|---|
| iOk | 0 if all open files were closed, otherwise a negative error code. |

## CEDS64Create

Create a new, empty file on disk. The new file is created with no file creator, no time and date, a tick period (time resolution) of 1 us, no file comments.

When you create a new file it is assumed that you will be writing to it. To this end, any channels are created with additional buffering that allows you to control the flow of data to disk. However, these buffering features are usually only of use in real-time data acquisition and they are not currently supported by the MATLAB® interface.

```
[ fhand ] = CEDS64Create( sPath{, iChans{, iType}} )
```

| | |
|---|---|
| sPath | A string containing the path and file name for the new file. This follows the usual rules for naming files. We do not add any file extension. The iType argument usually sets the file type. |
| iChans | The maximum number of channels the file can have, note this just assigns room in the file structure for the channels, the file is created with no channels. If omitted, 32 channels are set. |
| iType | The type of the file. 0 = 'small' 32-bit .smr, 1 = 'large' 32-bit .smr, 2 = 64-bit .smrx. You can also set -1, which sets type 0 if sFile ends with .smr, otherwise it sets type 2. If you omit this argument, it is taken as being -1. |
| fhand | An integer handle for the created file, 0 if no more handles or a negative error code. |

## CEDS64EditMarker

This allows you to replace the marker codes attached to a Marker item or an extended Marker (TextMark, RealMark, WaveMark) item in a channel of an open SON file.

```
[ iOk ] = CEDS64EditMarker( fhand, iChan, i64Time, cMarker )
```

| | |
|---|---|
| fhand | An integer file handle to identify the file. |
| iChan | The channel number (starting at 1). |

i64Time The time of an existing marker item in the channel.

cMarker A CEDMarker object from which we will copy the marker codes and apply them to the existing item in the file.

iOk 0 if the item was located at i64Time and was updated, otherwise a negative error code.

## CEDS64EmptyFile

This deletes all the channels and data in a file, but preserves the header information (file comments, date and time, creator, time base). If data is already written, the list of disk blocks reserved for each channel is preserved and any new data written to a previously used channel will reuse the previously allocated disk blocks before allocating more space in the file.

This is normally used when sampling data to restart sampling to the same file, but might be used when processing data to a new file to restart an analysis.

```
[ iOk ] = CEDS64EmptyFile( fhand )
```

fhand An integer file handle to identify the file.

iOk 0 if the operation completed sucessfully, otherwise a negative error code.

## CEDS64ErrorMessage

Converts a negative integer error code into a MATLAB® warning message describing the error.

```
[ ] = CEDS64ErrorMessage( iError)
```

iError A negative error code returned by one of the other matlib functions.

## CEDS64FileCount

Gets the number of SON data files currently opened via the library.

```
[ iFiles ] = CEDS64FileCount(  )
```

iFiles Returns the number of files currently open or a negative error code.

## CEDS64FileSize

Returns the (possibly approximate) current size of a SON file on disk.If you are writing a file, this will attempt to allow for data that is buffered in memory, but not yet written, hence approximate size. If a file is open for reading only, the size should be accurate.

```
[ i64Size ] = CEDS64FileSize( fhand )
```

i64Size Returns a 64-bit integer number that is the size of the file, or a negative error code.

## CEDS64FileComment

This gets and/or sets a comment associated with a file. The 32-bit SON library allowed 5 comments, each with up to 79 ASCII characters. The 64-bit format allows 8 comments and does not impose a length limit. However, we suggest that you restrict yourself to 5 comments of limited length as Spike2 will only display the first 5. We may assign the extra comments specific purposes, such as holding XML encoded meta-data.

```
[ iOk {, sCom} ] = CEDS64FileComment( fhand, iIndex{, sNew} )
```

fhand An integer file handle to identify the file.

iIndex The comment index in the range 1-5 (6-8 also exist in 64-bit files).

sNew — If present, a string to set a new comment. If omitted, no change is made.

iOk — 0 if the operation completed without error, otherwise a negative error code.

sCom — Returned as the original comment (before any change if sNew is provided).

## CEDS64GetExtMarkInfo

Returns the number of rows and columns of an extended marker channel. This is used in conjunction with CEDS64ReadExtMarks() and CEDS64WriteExtMarks() to make sure that the buffers created to hold the extended marker information are big enough.

```
[ iPre {, iRows {, iCols} } ] = CEDS64GetExtMarkInfo( fhand, iChan )
```

fhand — An integer file handle to identify the file.

iChan — The channel number (starting at 1) of an extended marker channel.

iPre — The number of pre-alignment points for WaveMark channels, otherwise 0 or a negative error code.

iRows — The number of rows of extended marker data. For RealMark data from Spike2 this will be the number of values. For TextMark data, this is the maximum number of 8-bit characters that could be stored (with space for a zero terminator). For WaveMark data, this is the number of waveform points per trace.

iCols — The number of columns of extended marker data (usually 1 unless this is a WaveMark channel with multiple traces when this is the number of traces).

## CEDS64GetFreeChan

This returns the number of the first unused channel in the file, if all channels are used you will get a negative error code (NO_CHANNEL).

```
[ iChan ] = CEDS64GetFreeChan( fhand )
```

fhand — An integer file handle to identify the file.

iChan — Returned as the channel number (starting at 1) of an unused channel or a negative error code.

## CEDS64IdealRate

Gets and/or sets the ideal rate, in Hz, for a channel. For a Waveform-based channel, this is the rate that the user who created the channel would have liked to sample at (the actual rate available may have been constrained by the data acquisition hardware), and is for information only. For an Event-based channel, this is usually set to the expected maximum event rate averaged over a period of a few seconds and can be useful when creating data displays. The SON library uses this value for event-based channels to allocate buffering space when writing to a new channel to optimise real-time sampling performance.

```
[ dRate ] = CEDS64IdealRate( fhand, iChan{, dNew} )
```

fhand — An integer file handle to identify the file.

iChan — The channel number to be deleted (starting at 1).

dNew — An optional argument. if present, it sets a new channel ideal rate value.

dRate — Returns the ideal rate for the channel before any change made by dNew, otherwise a negative error code.

# CEDS64IsOpen

Test if a file is open for a particular file handle.

```
[ iOpen ] = CEDS64IsOpen( fhand )
```

fhand An integer file handle to identify the file.

iOpen Returns 1 if there is an open file with this handle, 0 if not open, otherwise a negative error code.

# CEDS64LoadLib

Loads the ceds64int library allowing the rest of the CEDS64AbcXyz() functions to be used. This function must be called first. It must also be preceded by an addpath command, otherwise MATLAB® will not be able to find the CEDS64LoadLib.m file holding this command.

```
[ iOk ] = CEDS64LoadLib( sPath )
```

sPath A path to the folder in which the CED SON library MATLAB® support was included.

iOk 0 if the operation completed without error, otherwise a negative error code. However, this will only detect trivial errors such as the wrong number of arguments. If there is a problem loading the DLLs this library uses, there will be a MATLAB® error and the command will stop.

When you are done with these function you can use:

```
unloadlibrary ceds64int
```

to remove the library from memory. See the example here for a typical library startup sequence.

# CEDS64MaskCodes

Marker masks are opaque objects stored by the library that you can manipulate through function calls and apply to data reading functions to filter the returned values. This function transfers mask information between a marker mask held by the library and a 256 x 4 (or larger) numeric matrix. There are two variants of the call:

```
[ iOk, matCdRd ] = CEDS64MaskCodes( maskh)
        [ iOk ] = CEDS64MaskCodes( maskh, matCdWr )
```

maskh An integer handle to a mask.

matCdWr A matrix of 256 x 4 of a numeric type. Zero elements clear the mask, non-zero elements set it. For example: passing zeros(256, 4, 'uint8') would clear all elements of the mask.

iOk Returned as 0 if the operation completed without error, otherwise a negative error code.

matCdRd Returned as a 256 x 4 matrix of uint8. Each element is 0 or 1, depending on the state of the corresponding element of the mask.

### Examples

See the marker mask section for examples of this function.

# CEDS64MaskCol

Marker masks are opaque objects stored by the library that you can manipulate through function calls and apply to data reading functions to filter the returned values. They are also used to select which column of data to return when an extended marker item holds multiple columns.

For example, the WaveMark data type can hold multiple, interleaved data traces (typically it holds 1, 2 or 4 traces). When you read this data as a waveform (which has no way to return multiple traces) it will normally return the first trace. You can use this function to choose which trace is returned.

```
[ iOk {, iPrevCol} ] = CEDS64MaskCol( maskh {, iCol} )
```

maskh An integer handle to a mask.

iCol If present, 0 to n-1 to select the marker column. In the case of a WaveMark channel, n is the number of interleaved traces. If omitted, the marker column (trace number for WaveMark channels) is not changed.

iOk Returned as 0 if the operation completed without error, otherwise a negative error code.

iPrevCol The mask column select value for this filter before any change made by this call.

**Remarks**
At the time of writing, this function is only used with WaveMark data to select the trace returned by the CEDS64ReadWaveF() and CEDS64ReadWaveS() routines.

## CEDS64MaskMode

This function gets and/or sets the mode of a marker mask.

Each marker mask has a mode, currently either AND (0) or OR (1). In AND mode, for a particular Marker item to be passed by the filter, all 4 marker codes must be present in each of the corresponding 4 layers of the filter. In OR mode, only the first layer of the filter is used and a Marker item is passed by the filter if any of the 4 marker codes are in the filter.

```
[ iMode ] = CEDS64SetMaskMode( maskh{, iNew} )
```

maskh An integer handle to a mask.

iNew If present, sets the new mode as 0 for AND, 1 for OR.

iMode Returned as the mode before any change made by iNew.

**Examples**
See the marker mask section for examples of this function.

## CEDS64MaskReset

Resets a nominated marker mask. Removes all marker codes from all layers and sets mode 0 (AND). This sets a mask that passes nothing.

```
[ iOk ] = CEDS64MaskReset( {maskh} )
```

maskh The integer handle of the marker mask to reset or omitted to reset all masks.

iOk 0 if the operation completed without error, otherwise a negative error code.

## CEDS64MaxChan

Return the number of channels that the current file supports. This number was set when the file was created.

```
[ iChans ] = CEDS64MaxChan( fhand )
```

fhand An integer file handle to identify the file.

iChans Returns the maximum number of channels this file can hold or a negative error code.

# CEDS64MaxTime

Get the maximum time of any item in the file. The maximum time written to disk is saved in the file header, so if you open a file for reading this should be a very quick operation. If the time is not set in the file header the maximum time is found by scanning all the channels for the maximum time saved in any channel.

```
[ i64Time ] = CEDS64MaxTime( fhand )
```

fhand — An integer file handle to identify the file.

i64Time — The time of the last item in the file, or -1 if no items exist, otherwise a negative error code.

# CEDS64Open

Opens an existing file (there is a limit on the number of files that can be open at once, currently 8).

```
[ fhand ] = CEDS64Open( sPath{, iMode} )
```

sPath — A string holding the path and file name of the file to open. This follows the usual operating system rules for file names. If the file name has the extension .smr, we will first try to open it as a 32-bit SON file. Otherwise we will open it as a 64-bit SON file.

iMode — If omitted or set to -1, we will attempt to open the file in Read/Write mode, and if this fails, we will open it in ReadOnly mode. Otherwise set 1 to open in ReadOnly mode and 0 for Read/Write mode.

fhand — Returned as an integer file handle to the opened file, 0 if no more handles or a negative error code.

### Read only mode

When a 64-bit file is open in read only mode, we do not stop you making changes to things in memory, but nothing gets written to disk. This is a convenience to allow you to change things like the time base of a file opened in read only mode.

# CEDS64PrevNTime

Get the time of N items before a given time.

```
[ i64Time ] = CEDS64PrevNTime( fhand, iChan, i64From{, i64To{, iN {,
        maskh{, bAsWave}}} )
```

fhand — An integer file handle to identify the file.

iChan — The channel number (starting at 1).

i64From — The time in ticks to start searching backwards from. This time is not included in the search range.

i64To — Optional, value 0 if omitted. The end time of the backwards search, in ticks. This time is included in the search range.

iN — Optional, value 1 if omitted. A 32-bit integer holding the number of points backwards to search. If you need more than 32-bits worth you will have to loop this... and ask yourself why!

maskh — Optional, no filter if omitted. If present and non-zero, this is a handle to a marker mask for use with Marker and extended Marker channel types.

bAsWave — Optional, value 0 if omitted. Used with WaveMark data to force the channel to be treated as a waveform, not as events.

i64Time — Returned as the time of the data item or -1 if no item in the range or a negative error code.

### Event-based channels

For an event-based channel, we just count events (or filtered events) backwards from the i64From time. If N is 1, we get the first event before i64From. Things are a little more complicated for if maskh is set for a Marker-based channel as we have to ignore items that are filtered out. If there are not N in the given time range, the return value is -1. In that case, you can find the first event in the time range by using CEDS64ReadEvents().

**Waveform-based channels**
For a waveform channel, this returns the latest time at which a read of N items would end before the nominated time. This may mean that a waveform read would not get N items. Things are much more complicated with waveforms as there can be gaps. To find a contiguous range of N items you will have to iterate with this command and CEDS64ReadWaveS() or CEDS64ReadWaveF().

## CEDS64ReadEvents

Read event times from any suitable channel.

You can read event times from any event channel, marker channel or extended marker channel. You can use this to read EventBoth channels, which allows you to select rising or falling events by setting a filter.

```
[ iRead, vi64T ] = CEDS64ReadEvents( fhand, iChan, iN, i64From{, i64UpTo
        {, maskh}} )
```

| | |
|---|---|
| fhand | An integer file handle to identify the file. |
| iChan | The channel number (starting at 1). |
| iN | The maximum number of events to return. |
| i64From | The time in ticks to start searching from. This time is included in the range. |
| i64UpTo | Optional, taken as -1 if omitted. The end time of the search, in ticks. If this is set to -1, the search extends to the end of the file. This time is not included in the range. |
| maskh | Optional, no filter if omitted. If present and non-zero, this is a handle to a marker mask for use with Marker and extended Marker channel types. |
| iRead | Returned as the number of event times read or as a negative error code. |
| vi64T | Returned as a vector of 64-bit integers holding event times in ticks. This can be of zero length if no data is returned. |

## CEDS64ReadExtMarks

Read extended marker data from an extended marker channel.

```
[ iRead, vMObj ] = CEDS64ReadExtMarks( fhand, iChan, iN, i64From{, i64To{,
        maskh}} )
```

| | |
|---|---|
| fhand | An integer file handle to identify the file. |
| iChan | The channel number (starting at 1). This must be an extended marker channel. |
| iN | The maximum number of events to return. |
| i64From | The time in ticks to start searching from. This time is included in the range. |
| i64UpTo | Optional, taken as -1 if omitted. The end time of the read, in ticks. If this is set to -1, the read extends to the end of the file. This time is not included in the range. |
| maskh | Optional, no filter if omitted. If present and non-zero, this is a handle to a marker mask for use with Marker and extended Marker channel types. |
| iRead | Returns the number of event times read or a negative error code. |
| vMObj | Returns a vector of CEDTextMark, CEDRealMark or CEDWaveMark objects, depending on the type of the channel that is being read. This can be of zero length if no data is returned. |

## CEDS64ReadLevels

Read alternating levels from an EventBoth channel.

This is provided as a convenience, avoiding the need to read back the data as Markers (as an EventBoth channel stores data as Markers using codes 0 and 1 for low and high). It assumes that the channel holds alternating high and low levels.

```
[ iRead, vi64T, iLevel ] = CEDS64ReadLevels( fhand, iChan, iN, i64From{,
        i64UpTo} )
```

| | |
|---|---|
| fhand | An integer file handle to identify the file. |
| iChan | The channel number (starting at 1). |
| iN | The maximum number of event times to return. |
| i64From | The time in ticks to start searching from. This time is included in the range. |
| i64UpTo | Optional, taken as -1 if omitted. The end time of the search, in ticks. If this is set to -1, the search extends to the end of the file. This time is not included in the range. |
| iRead | Returns the number of event times read or a negative error code. |
| vi64T | Returns a vector of 64-bit integers holding event times in ticks. |
| iLevel | Returns the state of the first returned level time as 0 for low or 1 for high. This can be of zero length if no data is returned. |

## CEDS64ReadMarkers

Read marker data from a Marker or an extended marker channel.

```
[ iRead , vMObj ] = CEDS64ReadMarkers( fhand, iChan, iN,  i64From{,
        i64UpTo{, maskh}} )
```

| | |
|---|---|
| fhand | An integer file handle to identify the file. |
| iChan | The channel number (starting at 1). This must be an extended marker channel. |
| iN | The maximum number of events to return. |
| i64From | The time in ticks to start searching from. This time is included in the range. |
| i64UpTo | Optional, taken as -1 if omitted. The end time of the read, in ticks. If this is set to -1, the read extends to the end of the file. This time is not included in the range. |
| maskh | Optional, no filter if omitted. If present and non-zero, this is a handle to a marker mask for use with Marker and extended Marker channel types. |
| iRead | Returns the number of CEDMarker objects read or a negative error code. |
| vMObj | Returns a vector of CEDMarker objects. This can be of zero length if no data is returned. |

## CEDS64ReadWaveF

Read waveform data from waveform, realwave, wavemarker channels as 32-bit floating point values and returns them as a vector. If you read an Adc or WaveMark channel, the values are converted to floats using the scale and offset set for the channel.

The read operation stops at a gap in the data. A gap is defined as an interval between data points that is not the CEDS64ChanDiv() value defined for the channel.

```
[ iRead, vfWave, i64Time ] = CEDS64ReadWaveF( fhand, iChan, iN, i64From{,
        i64UpTo {, maskh}} )
```

| | |
|---|---|
| fhand | An integer file handle to identify the file. |

| | |
|---|---|
| iChan | The channel number (starting at 1). This must be a suitable channel. |
| iN | The maximum number of events to return. |
| i64From | The time in ticks to start searching from. This time is included in the range. |
| i64UpTo | Optional, taken as -1 if omitted. The end time of the read, in ticks. If this is set to -1, the read extends to the end of the file. This time is not included in the range. |
| maskh | Optional, no filter if omitted. If present and non-zero, this is a handle to a marker mask for use with WaveMark channels. |
| iRead | Returns the number of waveform points read or a negative error code. |
| vfWave | Returns a vector of 32-bit floating point, being the waveform data. This can be of zero length if no data is returned. |
| i64Time | If any data is read, this returns the time in ticks of the first item read into vfWave. If nothing is read, the value should be ignored. |

**Remarks**
When reading from WaveMark data with multiple traces, you can use the CEDS64MaskCol() command to select the trace to return in the marker mask.

# CEDS64ReadWaveS

Read waveform data from waveform, realwave, wavemarker channels as 16-bit integers (shorts) and return them as a vector of shorts (note you will need the offset and scale to interpret this data correctly).

If you read a RealWave channel, the values are converted to integers using the scale and offset set for the channel. If the value would exceed the 16-bit range, the value is limited to 16-bit range.

The read operation stops at a gap in the data. A gap is defined as an interval between data points that is not the lDvd value defined for the channel.

```
[ iRead, vi16Wave, i64Time ] = CEDS64ReadWaveS( fhand, iChan, iN,
        i64From{, i64UpTo {, maskh}} )
```

| | |
|---|---|
| fhand | An integer file handle to identify the file. |
| iChan | The channel number (starting at 1). This must be a suitable channel. |
| iN | The maximum number of events to return. |
| i64From | The time in ticks to start searching from. This time is included in the range. |
| i64UpTo | Optional, taken as -1 if omitted. The end time of the read, in ticks. If this is set to -1, the read extends to the end of the file. This time is not included in the range. |
| maskh | Optional, no filter if omitted. If present and non-zero, this is a handle to a marker mask for use with WaveMark channels. |
| iRead | Returns the number of waveform points read or a negative error code. |
| vi16Wave | Returns a vector of 16-bit integers, being the waveform data. This can be of zero length if no data is returned. |
| i64Time | If any data is read, this returns the time in ticks of the first item read into vi16Wave. If nothing is read, the value should be ignored. |

**Remarks**
When reading from WaveMark data with multiple traces, you can use the CEDS64MaskCol() command to select the trace to return in the marker mask.

## CEDS64SecsToTicks

This converts seconds into 64-bit ticks. You can convert a single value, or a vector of values. See CEDS64TicksToSecs() to convert ticks to seconds.

```
[ i64Tick ] = CEDS64SecsToTicks( fhand, dSecs )
        [ vi64Tick ] = CEDS64SecsToTicks( fhand, vdSecs )
```

fhand An integer file handle to identify the file.

dSecs A time, in seconds as a floating point value, to be converted to ticks.

vdSecs A vector of doubles holding times, in seconds, to be converted to ticks.

i164Tick Returned as a 64-bit integer holding the equivalent time, in ticks.

vi64Tick A vector of 64-bit integers holding the converted times in ticks.

There is a huge number of ticks that the SON library uses for the maximum time allowed. If the result of the conversion exceeds this value, the returned value is limited to the maximum value.

## CEDS64SetEventChan

This creates an EventRise, EventFall or EventBoth channel in a SON file. The file must not be opened in read only mode and the selected channel must not be in use. If a channel is in use, you must delete it first. You can also use CEDS64SetLevelChan() to create an EventBoth channel (which is actually a Marker channel in the 64-bit SON library).

```
[ iOk ] = CEDS64SetEventChan( fhand, iChan, dRate{, iType} )
```

fhand An integer file handle to identify the file.

iChan The channel number (starting at 1) to be created. The channel must be of type 0 (unused); it is OK to reuse a deleted channel.

dRate The expected sustained event rate in Hz. This is used when allocating buffer space for writing. This is more important when data files are used for real-time data acquisition, but it is worth putting in a reasonable estimate here, even when working off line.

iType The channel type as 2 for EventFall, 3 for EventRise and 4 for EventBoth. If you omit this argument, EventFall is used.

iOk 0 if the channel was created correctly, otherwise a negative error code.

## CEDS64SetExtMarkChan

This creates an extended Marker channel in a SON file. The file must not be opened in read only mode and the selected channel must not be in use. If a channel is in use, you must delete it first. You suggest that you use CEDS64SetTextMarkChan() to create a TextMark channel.

Each extended Marker data item holds a Marker plus a matrix of attached data items. The matrix usually has a single column. If the data is to be read by Spike2, the only data type that supports multiple columns is WaveMark data, which supports 1, 2 or 4 columns (corresponding to 1, 2 or 4 traces). As every item in the file has data space for the full matrix, it is very wasteful of disk space to use more matrix space than you need. Further, data is written in physical buffers of 32 kB in the 32-bit SON library and 64 kB in the 64-bit SON library. The buffer size limits the maximum size of an extended marker. We suggest that you limit TextMark channels to no more than 100 rows (character), RealMark channels to maybe 30 rows and WaveMark channels to 100 rows (points per trace) on up to 4 columns (traces).

```
[ iOk ] = CEDS64SetExtMarkChan( fhand, iChan, dRate, iType, iRows{,
        iCols{, i64Div}} )
```

fhand An integer file handle to identify the file.

iChan The channel number (starting at 1) to be created. The channel must be of type 0 (unused); it is OK to reuse a deleted channel.

dRate The expected sustained event rate in Hz. This is used when allocating buffer space for writing. This is more important when data files are used for real-time data acquisition, but it is worth putting in a reasonable estimate here, even when working off line.

iType The channel type as 6 for WaveMark, 7 for RealMark and 8 for TextMark.

iRows The number of rows in the matrix of attached data.

iCols The number of columns. If omitted, 1 column is set. Spike2 only supports more than 1 column for WaveMark data. The 32-bit SON library ignores requests for other than 1 column except for WaveMark data.

i64Div This is only used for WaveMark channels, when it must be provided. It sets the interval, in ticks, between row data points for WaveMark channels.

iOk 0 if the channel was created correctly, otherwise a negative error code.

## CEDS64SetInitLevel

This sets the initial level for an EventBoth channel, that is the resting level of the input before the first change (event) written to the file. Subsequent calls to CEDS64WriteLevels() are expected to write times at which the level changes. This call is only valid when no data has been written to the channel. Once data has been written, you can not change the initial state.

```
[ iOk ] = CEDS64SetInitLevel( fhand, iChan, iLevel )
```

fhand An integer file handle to identify the file.

iChan The channel number of an EventBoth channel.

iLevel The initial level of the channel, before the first change. 0 for low and 1 for high.

iOk 0 if the operation completed correctly, otherwise a negative error code.

## CEDS64SetLevelChan

This creates an EventBoth channel in a SON file. The file must not be opened in read only mode and the selected channel must not be in use. If a channel is in use, you must delete it first. An EventBoth channel is a variant of a Marker channel in the 64-bit SON library. The initial level of the channel is set to 0 (low) and can be modified using CEDS64SetInitLevel(). You can treat this channel as a Marker channel, in which case it is your responsibility to write only events with alternating codes 0 and 1 (or any non-zero code) for the first marker code and the remainder set to 0. However, you can also use this channel with CEDS64WriteLevels() and CEDS64ReadLevels() using only a list of times that are assumed to be alternating levels.

```
[ iOk ] = CEDS64SetLevelChan( fhand, iChan, dRate )
```

fhand An integer file handle to identify the file.

iChan The channel number (starting at 1) to be created. The channel must be of type 0 (unused); it is OK to reuse a deleted channel.

dRate The expected sustained event rate in Hz. This is used when allocating buffer space for writing. This is more important when data files are used for real-time data acquisition, but it is worth putting in a reasonable estimate here, even when working off line.

iOk 0 if the channel was created correctly, otherwise a negative error code.

## CEDS64SetMarkerChan

This creates a Marker or EventBoth channel in a SON file. The file must not be opened in read only mode and the selected channel must not be in use. If a channel is in use, you must delete it first. Markers are 64-bit integer) time stamps plus marker codes. In the 64-bit son library EventBoth data is stored as a Marker, not as an event,

using code 0 in the first code to mean low and any other code to mean high. Markers can be filtered using the codes.

You can also use CEDS64SetLevelChan() or CEDS64SetEventChan() to create an EventBoth channel (which is actually a Marker channel in the 64-bit SON library).

```
[ iOk ] = CEDS64SetMarkerChan( fhand, iChan, dRate{, iType} )
```

fhand — An integer file handle to identify the file.

iChan — The channel number (starting at 1) to be created. The channel must be of type 0 (unused); it is OK to reuse a deleted channel.

dRate — The expected sustained marker rate in Hz. This is used when allocating buffer space for writing. This is more important when data files are used for real-time data acquisition, but it is worth putting in a reasonable estimate here, even when working off line.

iType — The channel type as 5 for Marker or 4 for EventBoth. If you omit this argument, Marker is used.

iOk — 0 if the channel was created correctly, otherwise a negative error code.

## CEDS64SetTextMarkChan

This creates a TextMark channel in a SON file. The file must not be opened in read only mode and the selected channel must not be in use. If a channel is in use, you must delete it first.

A TextMark is a Marker plus a zero terminated 8-bit character string. You can store UTF-8 text in the marker (or anything else you want as the library does not interpret the string. The 64-bit SON library rounds up the size of a TextMark to the next multiple of 8 bytes. The 32-bit SON library rounded up the size to a multiple of 4 bytes. Data is written and read using CEDS64WriteExtMarks() and CEDS64ReadExtMarks(). TextMark items can be filtered on read.

```
[ iOk ] = CEDS64SetMarkerChan( fhand, iChan, dRate, iMax )
```

fhand — An integer file handle to identify the file.

iChan — The channel number (starting at 1) to be created. The channel must be of type 0 (unused); it is OK to reuse a deleted channel.

dRate — The expected sustained marker rate in Hz. This is used when allocating buffer space for writing. This is more important when data files are used for real-time data acquisition, but it is worth putting in a reasonable estimate here, even when working off line.

iMax — The number of 8-bit bytes (characters) attached to each TextMark data item, including the zero byte to terminate the string. This is usually set to a multiple of 8 (but it need not be).

iOk — 0 if the channel was created correctly, otherwise a negative error code.

## CEDS64SetWaveChan

Create a waveform channel using short (Adc) or float (RealWave) data items.

The library embodies the concept of waveform channels. That is channels holding data that is equally spaced in time (but allowed to have gaps with missing data). Although we could allow data of any underlying type, the library is currently set for two types: short and float. These cover most of the target applications.

The use of short data (16-bit signed integers) is convenient and compact (waveform data usually accounts for the bulk of the disk space used for data files. It also matches a common ADC specification. However, user data is more conveniently used in user units, so waveform channels have an associated scale and offset value that is used to convert between integer units and real units.

```
[ iOk ] = CEDS64SetWaveChan( fhand, iChan, i64Div, iType{, dRate} )
```

fhand — An integer file handle to identify the file.

iChan — The channel number (starting at 1) to be created. The channel must be of type 0 (unused); it is OK to reuse a deleted channel.

`i64Div` The channel sample interval as an integral divide down from the file time base (with a minimum value of 1).

`iType` The type of waveform channel to create as 1 for a 16-bit integers or 9 for a RealWave channel.

`dRate` Optional. The desired sample rate in Hz. It can happen that due to the choice of the file time base, the `i64Div` value is only an approximation to the desired rate. This stores the desired rate for information purposes, see `CEDS64IdealRate()`. If set 0 or negative, `dRate` is calculated as: `1.0/(i64Div*CEDS64TimeBase())`.

## CEDS64TicksToSecs

This converts seconds into 64-bit ticks. You can convert a single value, or a vector of values. See `CEDS64SecsToTicks()` to convert seconds to ticks.

```
[ dSecs ] = CEDS64TicksToSecs( fhand, i64Tick )
        [ vdSecs ] = CEDS64TicksToSecs( fhand, vi64Tick )
```

`fhand` An integer file handle to identify the file.

`i64Tick` A time, in ticks to be converted to seconds.

`vi64Tick` A vector of 64-bit integer times in ticks to be converted to a vector of seconds.

`dSecs` Returned as a double holding the equivalent time, in seconds.

`vdSecs` Returned as a vector of doubles holding the converted times in seconds.

### Resolution of times in seconds

A SON library has a constant time resolution of 1 tick, regardless of the magnitude of the time. Times are stored in the file using 64-bit integers; if we restrict ourselves to positive times, we have 63 bits of precision. This means that calculations of time intervals will be just as accurate at the end of a file as at the start.

This is not the case for times in seconds, which are stored in double precision floating point values. These have approximately 53 bits of precision. The result of this is that if you need to form differences of times that are large in tick terms, you will get precise results when working with ticks, but less precise ones when working with times in seconds. However, this is unlikely to be a problem in most circumstances. For example, if you are working with a time base of 1 microsecond per tick, times as seconds will start to have uncertainties at the microseconds level at a file duration around 100 years. With a time base of 1 nanosecond, seconds would become inaccurate at the nanosecond level around a duration of 1 month.

## CEDS64TimeBase

Gets and/or sets the file time base (the length of each tick) for the data file.

```
[ dSecs ] = CEDS64TimeBase( fhand{, dNew} )
```

`fhand` An integer file handle to identify the file.

`dNew` An optional argument. if present, it sets a new time base value, in seconds per tick.

`dSecs` If present, returned holding the time base in seconds before any change made by providing `dNew`. If there is an error, this is returned as a negative error code.

### Changing the time base of an existing file

By changing the time base you can compress or expand time in the file. This has been used to recover data from long-term analogue recorders that can play back at many times real time so that Spike2 (for example) can record 1 day of data in a few minutes by sampling very fast with a fast time base, then the data can be slowed down back to real time by changing the time base. However, if the original file had a time base of 1 microsecond and you slowed it down by a factor of 100, the time resolution in the file become 100 microseconds. It is possible then to write this data to a new file with a 1 mircosecond (or any other) time base; see the Spike2 `ChanSave()` script command for details.

## CEDS64TimeDate

The data file reserves space to hold the time and date at which sampling starts. There is no guarantee that any file will have this value set, or that if it is set, it is accurate. Spike2 always sets this value as accurately as it can. If it is not set, the entire space is set to zero. This is not the same as the date and time that the operating system has associated with the file.

The time and date is encoded in an array of integers with values within the limits described below. You can set any values you like, but if you set values beyond the limits, Spike2 will ignore the time and date. Other applications may behave differently with invalid times and dates.

| Index | Low | High | Meaning |
|---|---|---|---|
| 1 | 0 | 99 | Hundredths of a second |
| 2 | 0 | 59 | Seconds |
| 3 | 0 | 59 | Minutes |
| 4 | 0 | 23 | Hours |
| 5 | 1 | 31 | Day number in the month |
| 6 | 1 | 12 | Month number in the year |
| 7 | 1980 | 2200 | Year |

```
[ iOk, TimeDate ] = CEDS64TimeDate( fhand {, newTD } )
```

fhand — An integer file handle to identify the file.

newTD — Optional. If present the application time and date field will be changed. This should be an array of at least 7 integers holding the new date in the format given above. Setting all values to 0 is also acceptable, meaning clear the time and date field.

iOk — Returned as 0 if the operation completed without error, otherwise a negative error code.

TimeDate — Returned as and array of 7 integers holding the contents of the time and date field in the format given above. If you are setting a new value, the returned value is the original time and date.

## CEDS64Version

This returns the version of the data file, which also tells you if it is a 32-bit file or a 64-bit file.

```
[ iVersion ] = CEDS64Version( fhand )
```

fhand — An integer file handle to identify the file.

iVersion — Returned as the file version. Versions 1 to 8 are 32-bit files with a maximum size of 2 GB. Version 9 is a 32-bit file with a maximum size of 1 TB. versions 256 and later are 64-bit files.

## CEDS64WriteEvents

Write EventFall or EventRise data to an event channel. Use CEDS64WriteLevels() for an EventBoth channel. The data must be after all data written previously to the channel. The data must be in ascending time order (or the file will be corrupted and reads from the channel will be confused).

```
[ iOk ] = CEDS64WriteEvents( fhand, iChan, vi64Event )
```

fhand — An integer file handle to identify the file.

iChan — The channel number to write to. This must be an EventFall or EventRise channel.

vi64Time — A vector of 64-bit times in ticks to write to the channel. Use CEDS64SecsToTicks() to convert times in seconds to ticks.

iOk — Returns 0 if the operation completed with no problems or a negative error code.

**Example**
This creates a vector of 1000 times in ticks, then sets them to the tick equivalent of m*m milliseconds, where m is the index of each item. It then writes the data to a channel that must have been created with CEDS64SetEventChan() and that does not hold any data at or after 1 millisecond.

```
vTime = zeros(1000, 1, 'int64'); % create a vector of 1000 64-bit times
for m=1:1000
    % set time of each item as 0.001*(m*m) seconds in ticks
    vTime(m) = CEDS64SecsToTicks( fhand, 0.001*(m*m) );
end
% The type of channel evchan must be EventRise or EventFall
fillret = CEDS64WriteEvents( fhand, evchan, vTime );
if fillret < 0, CEDS64ErrorMessage(fillret); end;
```

## CEDS64WriteExtMarks

Write a vector of extended marker objects to an extended marker channel. The data must be after all data written previously to the channel. The data must be in ascending time order (or the file will be corrupted and reads from the channel will be confused).

```
[ iOk ] = CEDS64WriteExtMarks( fhand, iChan, vExtMks )
```

fhand — An integer file handle to identify the file.

iChan — The channel number to write to. This must be a TextMark, RealMark or WaveMark channel.

vExtMks — A vector of extended markers. The type of the vector must match the channel type, that is a TextMark channel expects a vector of CEDTextMark objects, a RealMark channel expects a vector of CEDRealMark objects and a WaveMark channel expects a vector of CEDWaveMark objects.

iOk — Returns 0 if the operation completed with no problem or a negative error code.

**Example**
This creates a vector of 1000 CEDTextMark objects with time and codes set to 0 and holding an empty string. It then sets their times to the marker index in seconds and sets the first two marker codes to arbitrary values and the string to a message. The data is written to a channel that must have been created with CEDS64SetTextMarkChan() and that holds no data at or after 1 second.

```
vTMk(1000, 1) = CEDTextMark(); % create a vector of 1000 empty text markers

for m = 1:1000
    vTMk(m).SetTime( CEDS64SecsToTicks( fhand, m ) ); % set time to m secs in ticks
    vTMk(m).SetCode( 1, uint8(mod(m+60, 256)) );      % set code 1 to m+60 mod 256
    vTMk(m).SetCode( 2, uint8(mod(m+61, 256)) );      % set code 2 to m+61 mod 256
    vTMk(m).SetData( horzcat(int2str(m), ' squared is equal to ', int2str(m*m)) );
end

% tmarkchan must be a TextMark channel as we write a string
fillret = CEDS64WriteExtMarks(fhand, tmarkchan, vTMk);
if fillret < 0, CEDS64ErrorMessage(fillret); end;
```

## CEDS64WriteLevels

This command expands a vector of 64-bit times into Marker data and writes it to a EventBoth channel. The written data must lie after all preceding data and the first time is assumed to have the opposite level to the last written level. If no data has been written, the first level is assumed to toggle the level set by CEDS64SetInitLevel(). This command is here both for convenience and for compatibility with the 32-bit SON library where EventBoth data was stored as event times, not as Markers.

```
[ iOk ] = CEDS64WriteLevels( fhand, iChan, vi64Time )
```

fhand — An integer file handle to identify the file.

iChan — The channel number to write to. This must be an EventBoth channel.

`vi64Time` A vector of 64-bit times in ticks to write to the channel. Use `CEDS64SecsToTicks()` to convert times in seconds to ticks.

`iOk` Returns 0 if the operation completed with no problems or a negative error code.

## CEDS64WriteMarkers

Write a vector of `CEDMarker` data to a Marker or an EventBoth channel. If you use this with an EventBoth channel, you take responsibility for getting the order of the marker codes correct. The data must be after all data written previously to the channel. The data must be in ascending time order (or the file will be corrupted and reads from the channel will be confused).

```
[ iOk ] = CEDS64WriteMarkers( fhand, iChan, vMarker )
```

`fhand` An integer file handle to identify the file.

`iChan` The channel number to write to. This must be a Marker or EventBoth channel.

`vMarker` A vector of `CEDMarker` data.

`iOk` Returns 0 if the operation completed with no problem or a negative error code.

**Example**

This creates a vector of markers, timed at 1 second intervals, and sets the first marker code, leaving codes 2-4 as 0. We write the data to a channel that must already have been created with CEDS64SetMarkerChan() and that must not hold any data at or after 1 second (the first time we write).

```
vMark(1000, 1) =  CEDMarker(); % create a vector of 1000 empty markers
for m=1:1000
    vMark(m).SetTime( CEDS64SecsToTicks( fhand, m ) ); %set time to m secs in ticks
    vMark(m).SetCode( 1, uint8(mod(m, 256)) );         %set code 1 to m mod 256
end

% markchan must be a Marker channel as we are writing CEDMarker items
fillret = CEDS64WriteMarkers( fhand, markchan, vMark );
if fillret < 0, CEDS64ErrorMessage(fillret); end;
```

## CEDS64WriteWave

Write waveform data as shorts(16-bit integers) or as floats (32-bit floating point) to a waveform channel. Please read the documentation for `vWave`, below, carefully.

Unlike event-based channel types where all data must be written after any data already present in a channel, we allow you to overwrite previously-written waveform data. This is to remain compatible with the 32-bit SON library though it could be argued that the modification of primary data should not be allowed. You cannot fill in gaps in the orignal data in this manner. It is imagined that you would use this to fix glitches or remove transients.

In normal use, all waveform data for a channel will be aligned at time that are the first data time plus an integer value times the `i64Div` value set for the channel with `CEDS64SetWaveChan()`. We do not prevent you writing data where the alignment is not the same after a gap, but programs like Spike2 will have subtle problems if you do this.

```
[ i64TNext ] = CEDS64WriteWave( fhand, iChan, vWave, i64Time )
```

`fhand` An integer file handle to identify the file.

`iChan` The channel number to write to. This must be an Adc or RealWave channel.

`vWave` A vector of floats (32-bit floating point) or shorts (16-bit integer) data. If you write floating point data to an Adc channel or integer data to a RealWave channel, the data will be converted from floating point to integer or integer to floating point using the channel scale and offset values.

`i64TNext` Returns the time of the next write to add contiguous data to this channel. That is, if you write N value at time i64Time, it will return `i64Time + N * CEDS64ChanDivide(fhand, iChan)`. If there was a problem the return value is a negative error code.

### Example

This example creates vectors of random 16-bit integers in the range -32768 to 32767 (the full range of 16-bit signed integers) and of single precision floating point variables in the range 0 to 1. We then write the integer data to an Adc channel starting at 10 seconds and the floating point data to a RealWave channel, starting at 20 seconds. These channels must already exist (you can create them with CEDS64SetWaveChan()). The interval between the data samples is set when the channel is created.

```
% create a vector of 5000 random 16-bit integers between -30000 and 30000
vWaveS = randi([-32768, 32767],5000,1, 'int16');

% create a vector of 5000 random 32-bit floats between 0 and 1
vWaveF = rand(5000, 1, 'single');

sTime = CEDS64SecsToTicks( fhand, 10 ); % offset start by 10 seconds
fillret = CEDS64WriteWave( fhand, ADCwavechan, vWaveS, sTime );
if fillret < 0, CEDS64ErrorMessage(fillret); end;

sTime = CEDS64SecsToTicks( fhand, 20 ); % offset start by 20 seconds
fillret = CEDS64WriteWave( fhand, Rwavechan, vWaveF, sTime );
if fillret < 0, CEDS64ErrorMessage(fillret); end;
```

# 4: Class objects

# Class objects

In addition to the functions, the library can create 4 types of class object to hold Marker and extended marker data. These object are designed to make it easier for a MATLAB® programmer to deal with these data types.

## CEDMarker

A `CEDMarker` object is defined in the `CEDMarker.m` file. You construct an object of this type with:

```
[ marker ] = CEDMarker({Time{, Code1{, Code2{, Code3{, Code4}}}}})
```

`Time` This is the time of the object, in ticks.Only positive integers are accepted, otherwise the time is set to 0.

`Code1` The first marker code. If omitted, the code is set to 0. If this is an integer, negative values are set to 0, otherwise it is truncated to an unsigned 8-bit integer. If this is a floating point number, it is converted to an integer (rounded down) and then treated as an integer. If this is a vector (of numeric), the first element is used and treated as above. If a string is given, the first character is converted to an integer via the ASCII map.

`Code2` The second marker code. This is handled as Code1.

`Code3` The third marker code. This is handled as Code1.

`Code4` The fourth marker code.This is handled as Code1.

Internally, the values you set are stored in the properties:

```
    properties
        m_Time;           % created as a 64-bit integer holding the time
        m_Code1;          % The codes are created as uint8 holding the codes
        m_Code2;
        m_Code3;
        m_Code4;
    end
```

The internal class members have public access for reading, but not for writing. This allows them to be viewed within MATLAB®, but not changed. We also provide the following access functions.

### Access functions

The class (and all the classes derived from it: `CEDRealMark`, `CEDTextMark`, `CEDWaveMark`) implements the following access functions:

```
[ code ] = GetCode( n )
```

`n` The index of the code to return in the range 1-4.

code Returned as the desired code, or a negative error.

```
[ err ] = SetCode(n, new)
```

`n` The index of the code to set in the range 1-4.

`new` The new code value, following the rules for the constructor.

`err` Returned as 0 or negative error code.

```
[ time ] = GetTime()
```

`time` Returned as the time value.

```
[ err ] = SetTime( time )
```

`time` The new time to store in the marker, following the same rules as the constructor.

err Returned as 0 or negative error code.

**[ data ] = GetData()**

data Returned as the data attached to the object. This is for use with the CEDRealMark, CEDTextMark and CEDWaveMark objects. If you use this on a CEDMarker object the value returned is always the BAD_PARAM error code.

**[ err ] = SetData( data )**

data Data of the correct type to attach to the object. You should NOT call this for a CEDMarker object.

err Returned as 0 or negative error code. As the CEDMarker class has no attached data, this always returns BAD_PARAM.

**[ r, c ] = Size()**

r Returned as the number of rows of items attached to this object; always 0 for a CEDMarker object. This will be the number of real values for a CEDRealMark object, the number of characters (including the terminating 0) for a CEDTextMark object and the number of data points per trace for a CEDWaveMark object.

c Returned as the number of columns in the attached data; always 0 for a CEDMarker object. This is usually 1 except for a CEDWaveMark object with multiple traces.

Note that you can replace the GetXxxx() calls with a direct reference to the properties as they are marker for public access when reading but private access for writing. However, using the access functions insulates you from any changes to the internal structure of the CEDMarker object.

### Examples

```
marker = CEDMarker(0, 'This', 'should', 'be', 'OK!');
marker.SetCode(1, 99);
```

Sets the time to 0 and codes to ('T', 's', 'b', 'O') then changes the first code to 99.

```
m = CEDMarker(1000, 1000, 25.123, -1234, 1234.5678);
time = m.GetTime();
```

Sets the time to 1000 ticks and the codes to (255, 25, 0, 255), then reads back the time.

```
mark = CEDMarker();
mark.SetTime(10000);
```

Sets the time and all codes to 0, then sets the time to 10000 ticks.

## CEDTextMark

A CEDTextMark object is defined in the CEDTextMark.m file, derives from a CEDMarker and has all the same access functions. It has an additional property:

```
    properties
        m_Data;           % string (empty if no string passed to constructor)
    end
```

The constructor is:

**[tMark] = CEDTextMark(Time{, Code1{, Code2{, Code3{, Code4{, text}}}}})**

Time As for CEDMarker.

Code1-4 As for CEDMarker.

text Omitted or a string holding the text to set in the object. If the value is unsuitable or omitted, m_Data is set to be an empty string. There is no length limit on the text stored in a CEDTextMark. However, if you write the data to a data file, the CEDS64WriteExtMarks() call will truncate strings that are too long for the space reserved in the target channel.

### Access functions

These are all as described for CEDMarker. SetData() follows the same rules as the constructor. GetData() returns a string.

## CEDRealMark

A CEDRealMark object is defined in the CEDRealMark.m file, derives from a CEDMarker and has all the same access functions. It has an additional property:

```
properties
    m_Data;          % created as a real or a vector of reals
end
```

The constructor is:

```
[rMark] = CEDRealMark(Time{, Code1{, Code2{, Code3{, Code4{, real }}}}})
```

| | |
|---|---|
| Time | As for CEDMarker. |
| Code1-4 | As for CEDMarker. |
| real | Omitted or a real or a vector of reals to store in the object. There is no length limit on the vector stored in a CEDRealMark. However, if you write the data to a data file, the CEDS64WriteExtMarks() call will truncate vectors that are too long for the space reserved in the target channel. |

### Access functions

These are all as described for CEDMarker. SetData() follows the same rules as the constructor.

## CEDWaveMark

A CEDWaveMark object is defined in the CEDWaveMark.m file, derives from a CEDMarker and has all the same access functions. It has an additional property:

```
properties
    m_Data;          % created as a vector or a matrix(traces, pointsPerTrace) of
end
```

The constructor is:

```
[wmark] = CEDWaveMark(Time{, Code1{, Code2{, Code3{, Code4{, wave }}}}})
```

| | |
|---|---|
| Time | As for CEDMarker. |
| Code1-4 | As for CEDMarker. |
| wave | Omitted or a vector or matrix of 16-bit integers to attach to the object. If omitted, the value 0 is attached. There is no limit on the size of the item stored in a CEDWaveMark. However, the maxium number of columns supported by Spike2 (and by the 32-bit SON library) is 4. If you write the data to a data file, the CEDS64WriteExtMarks() call will truncate vectors or matices that are too big for the space reserved in the target channel.<br><br>If you have multiple traces, the waveform data is organised in memory in an interleaved format. That is the first point of all traces, followed by the second point of all traces, and so on. If you get this the wrong way around, you will see each trace displayed in Spike2 as a downsampled version of the first trace followed by a down-sampled version of the second trace, and so on. |

### Access functions

These are all as described for CEDMarker. SetData() follows the same rules as the constructor.

# 5: Use of these libraries from other languages

# Use of these libraries from other languages

The ceds64int DLL is an interface between the son64 DLL and MATLAB® code. The same DLL can also be used from other languages, and we have provided the .lib files to make it easy for you to do this. If you choose to do this you are on your own; we assume that you know what you are doing and can solve your own problems.

The son64 DLL is written with a C++ interface and we do not suggest that you try to use it (especially if you are writing in a language other than C++). Instead, use the simpler ceds64int DLL (written in a C compatible manner) as your interface. The `ceds64int.h` file contains DOxygen style documentation for every routine and structure (and you can extract it using DOygen). You will find that in almost every case, the routine documented for MATLAB® use as:

```
CEDS64XxxXxxx(...)
```

is implemented though a function in ceds64int with the name:

```
S64XxxXxxx(...)
```

These pairs of routines will often have similar (if not identical) arguments, but you will need to check this. In particular, arguments that are described as MATLAB vectors and matrices will have been specially treated as the C compatible interface works with C arrays of short, float and 64-bit integers and also arrays of structures that represent Markers and extended Marker data.

We make no promises regarding the stability of the ceds64int DLL API (though we will not change it needlessly). That is, each time we release a new version of this kit it is possible that the details of the interface will change, requiring you to modify your code. This has not happened yet, but you should be aware of it. The son64 DLL is the same as we use in the version of Spike2 current when we release the kit. However, do not replace this DLL with one from Spike2 as there is no guarantee that it will have an identical interface (we update Spike2 much more often than we update this kit).

If you do use this library to build a tool, please let us know.

# Index

## - F -



## - G -



## - I -



## - L -



## - M -



## - O -



## - R -



## - S -



## - T -



## - U -



## - V -

## - W -



## - Y -