SON
Data storage library

Version 9

January 2009

Version 1	August 1988
Version 2	January 1989
Version 3	June 1989
Version 4	July 1992
Version 5	November 1992
Revised	October 1993
Revised	December 1998
Revised	June 2000
Version 6	July 2001
Version 7	July 2002
Version 8	June 2006
Version 9	January 2009

Published by:

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

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

Table of contents

The SON filing system

Introduction

This document is intended for programmers who are familiar with C/C++ who wish to make direct use of SON data files in their own programs. There are few concessions made to a casual reader; this is not for beginners. This document describes version 9 of the SON filing system as used in Spike2 version 6.11 onwards.

The SON filing system was originally designed to store efficiently channels of 16-bit integer waveforms and discrete 32-bit integer time stamps from a continuous stream generated by hardware. The data acquisition hardware was modelled as having one Analogue to Digital Converter shared between the waveform channels and running at a rate that was a multiple of a clock used for timing events.

The system has been extended several times to include new channel types and remove limitations in the original design. The filing system is now an established protocol for storing data on disk, plus a set of functions designed to store event times, waveform and marker data so that any desired event, marker or section of waveform data can be retrieved using the time of occurrence of the data as the key.

The SON filing system was originally written in Pascal for use on DOS machines and some of this heritage is evident in the underlying disk data structures. The Pascal version is no longer being developed though it will still read older files produced by the C version. We recommend that all users use the C code and Windows users of non-C languages use the DLL version of the C library with suitable headers for their language. We have maintained binary backwards compatibility of the file structure for all versions of the C library (i.e. the current version will read all previous revisions).

The filing system is treated at several levels:

- The rest of this chapter gives a description of the main features of the system and provides background information.
- An overview of the functions and sequence of operations required to read an existing file, and to create a new file, is presented for C programmers.
- A C/C++ reference section describes each of the error codes, functions and data structures visible to a programmer. A separate section describes internal structures and functions that are only visible to programmers working inside the SON library.
- The `SON32.DLL` contains a type library arranged to make it possible for a Visual Basic user to run the library. You can use this sections as a model for using the library from Visual Basic and as a starting point for other languages.

Clock ticks and base time units

The most basic concept behind the system is that of the indivisible unit of time, the *clock tick*. The clock tick period can be set to an integer multiple in the range 1 to 32767 of a base timing unit (usually microseconds). Normal settings are from 2 to 1000 base units with the base units in microseconds. Before version 6, the base units were always microseconds. From version 6 onwards you can set the base unit to any value but most files still use microseconds to be compatible with old applications.

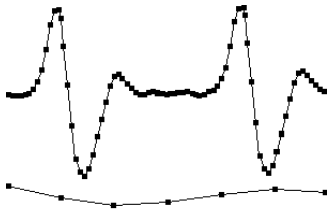
Time is measured in multiples of this clock tick, using a 32-bit integer representation. 32-bit signed numbers run from –2,147,483,648 to 2,147,483,647 and data files are only allowed to hold positive times, so the length of a data file is limited to the clock tick period times 2,147,483,647. Using a 2 microsecond clock tick period limits the file length to 71.5 minutes; with a 10 microsecond clock tick, files may last for 7 hours, which is usually sufficient.

Data channel types

A SON file contains from 32 to 451 channels of data (not all channels need be used). The number of channels cannot be changed once a file has been created. The standard and backwards compatible number of channels is 32. The first channel is number 0. There are currently nine different types of data channel:

- 16-bit integer waveform data (called `Adc` throughout the SON library)
- Event data, times taken on the low going edge of a pulse (`EventFall`)
- Event data, times taken on a high going edge of a pulse (`EventRise`)
- Event data, times taken on both edges of a pulse (`EventBoth`)
- Markers, an event time plus four identifying bytes (`Marker`)
- 16-bit integer waveform transient shapes (we call this `AdcMark` data), an array of waveform data attached to a marker. In version 6 the waveform data may be multiple, interleaved channels.
- Real markers, an array of real numbers attached to a marker (`RealMark`)
- Text markers, a string of text attached to a marker (`TextMark`)
- 32-bit floating point waveforms (`RealWave`). These are new at version 6.

Waveform data



The waveforms you record are continuously changing voltages. The SON file format stores waveforms as a list of 16-bit signed integers (`Adc`) or 32-bit floating point numbers (`RealWave`) that represent the waveform amplitude at equally spaced time intervals. We also store a scale factor and offset for each channel to convert between integers and user defined units, the value of the data in user units is given by:

```
real value = integer * scale / 6553.6  +  offset
```

This scale factor was so that on systems where the 16-bit integer range corresponded to ±5 Volts, a scale factor of 1.0 and an offset of 0.0 produced a real value in Volts. The `scale` and `offset` allow us to read `RealWave` data as `Adc` and `Adc` data as `RealWave`.

The process of converting a waveform into a number at a particular time is called sampling. The time between two samples is the sample interval and the reciprocal of this is the sample rate, which is the number of samples per second. The dots in the diagram represent samples, the lines show the original waveform.

The sample rate for a waveform must be high enough to correctly represent the data. It can be demonstrated mathematically that you must sample at a rate at least double the highest frequency contained in the data. On the other hand, you want to sample at the lowest frequency possible, otherwise your disk system will very soon be filled. Unlike many data storage systems, the SON library allows you to save different waveform channels at different rates.

In the SON data model, waveform data is sampled at an integer multiple of the file clock tick. Each waveform channel can be sampled at a different multiple of this clock tick, thus all the waveform channels can be sampled at different rates.

Before version 6, all waveform channels were presumed to be sampled at a multiple of a time interval that was itself a multiple of the file clock tick period. The time interval was given by `usPerTime * timePerADC` *base time units. This modelled the situation where data came from an ADC (Analogue to Digital Converter) that ticked every* `timePerADC` *clock ticks. We define* `timePerADC` *later.*

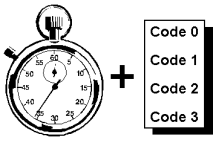
Each block of waveform data holds a start time, so waveform data need not be continuous. Two waveform blocks on the same waveform channel hold continuous data if the time interval between last sample in the first block and the first sample of the second block is equal to the sample interval.

Event data

There are three types of Events that are stored in the same way: `EventFall`, `EventRise` and `EventBoth`. We refer to all three types as `Event` data. Events are 32-bit time stamps. If the value of the time stamp is `n`, this means a time of `n` clock ticks which is `n * usPerTime` base time units. Events can be either discrete points having a time of occurrence but no duration or they can mark the change of state of a signal between two levels. Generally the `EventFall` and `EventRise` forms of events are the more useful.

Event channel types

Marker data

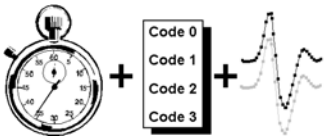


Marker events (`Marker`) are stored as 32-bit times, like events, but they have 4 additional bytes of information associated with each time. These additional bytes are used to hold information about the type of each marker. Markers typically hold user key press information, or the state of variables. For example, the CED VS software uses markers to hold the start time of stimulus presentations, and to hold the state of the independent variables that define the presentation. The Spike2 software uses the first byte as the ASCII code of a key press or data from the digital inputs.

Markers can be filtered by a user-supplied mask so that only those markers which meet a given set of criteria will be considered.

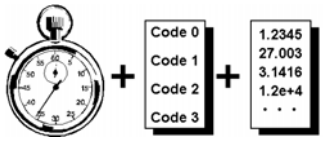
The SON library functions can read a `Marker` channel as an `Event` channel or as a `Marker` channel.

AdcMark data



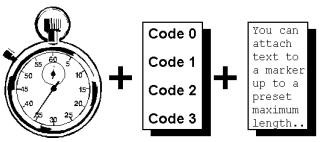
This type is a combination of waveform and marker data. It is stored as a 32-bit time and 4 bytes of marker information, followed by waveform points. From version 6 onwards, the waveform data may be from 1 up to 8 channels, stored as interleaved data. The waveform(s) typically holds a transient shape, and the marker bytes hold any classification codes required for the transient. The first point in the waveform data is sampled at the marker time. The SON library functions can use an `AdcMark` channel as if it were a waveform (only if there is a single channel of data), `Event`, `Marker` or `AdcMark` channel. To avoid alignment problems, we recommend that the number of points times the number of interleaved channels is an even number.

RealMark data



This is effectively `Marker` data to which is attached an array of real numbers. It is stored as a 32-bit time and 4 bytes of marker information, followed by an array of 4 byte real numbers. The SON library functions can use a `RealMark` channel as if it were an `Event`, `Marker` or `RealMark` channel. A `RealMark` channel with a single real value can be treated as a non equal-spaced waveform channel.

TextMark data



This is `Marker` data to which is attached a character string. It is stored as a 32-bit time and 4 bytes of marker information, followed by an array of characters. The character array is of fixed size, the strings stored in the arrays are terminated by a zero byte and can be of any length up to the array size-1. We recommend that you use array sizes that are multiples of 4 to avoid alignment problems. The SON library functions can use a `TextMark` channel as if it were an `Event`, `Marker` or `TextMark` channel.

Reading data

Data is read back from a SON file by specifying a channel and a time range. When waveform (`Adc` or `RealWave`) data is read, the time of the first waveform data point is returned, as well as the data itself. If there is a gap in the waveform data within the requested range, data is only returned up to the gap. The following data must be accessed by a new read request, timed to start after the end of the data from the previous read.

For all channel types, if the buffer is returned full, further reads should be done with the start time set to be the time of the last data item in the buffer plus 1, as the only way to be sure you have reached the end of the data is if you are returned a partially full buffer.

Writing data

Data is written in time sequence for a particular channel and the channels can be written in any order. Each channel forms a doubly linked list of data blocks in the file. You can read from a file while it is open for writing.

Data channels can be deleted. If this is done, the chain of blocks that was in use by a channel remain and will be re-used if the channel is re-written. The only way to remove the space used by a deleted channel is to write a program to copy the file to a new file, omitting the deleted channels.

File selection and number of files

When a SON file is opened or created, the library returns an integer value called a handle, to the calling software. All subsequent function calls that operate on this file must supply the handle. When the file is closed the file handle becomes invalid. The SON library allows up to 2048 files open at any one time, but available memory may reduce this. File description tables occupy memory space when the files are in use, so it is good practice to close files when they are no longer needed.

Physical disk storage

The data for each channel is written to disk in blocks. For a particular channel, the block is of a fixed size, although the block need not be full. All blocks are a multiple of 512 bytes long. Each block has a header that is 20 bytes long. This header holds the pointers to the previous and the next block in the file, the times of the first and last data items in the block, the channel number and the number of data items in the block.

In the file header is an array of descriptors, one for each data channel. These descriptors have pointers to the first and last data blocks for each channel. We use a pointer of value -1 to mean the end of a chain of blocks.

Data is found by skipping along the linked lists until the required block is found. The library builds an index table of blocks in order to speed up the search for data. You will not normally need to use any of this physical storage information. More details are available in the chapter on internal structures and functions

SON file versions

This manual describes version 9 of the SON filing system. The library functions will read files produced by all previous versions, files created or updated by the version 9 library have version numbers as low as possible depending upon the library features used. Previous versions of the library will not read files with a higher library version. The major changes between revisions are:

Version 1 The original SON filing system, in Pascal, supported waveform and `Event` data only.

Version 2 New `Marker` data type. Extra space was added to the file header for future expansion. The version 2 library was much faster dealing with large data files as it remembered the last access point for each file during reading.

Version 3 The `FilterMarker` function was added with this version and the meaning of the divide element of the channel record for waveform data was changed to prevent an apparent change of sampling rate if a waveform channel was added or deleted.

Version 4 The `AdcMark` data type was added, the use of `FilterMarker` was changed and the library now remembers the data block currently in the data buffer to avoid reading the disk when the required block is already in memory.

Version 5 The `TextMark` and `RealMark` data types and the `MaxTime` and `ChanMaxTime` functions were added. Versions were also written in C for both the IBM PC in DOS and under Windows and for the Macintosh. The version 5 libraries write version 3 and 4 files as appropriate to prevent unnecessary conflict between library versions.

In 1998, the C version of the library was extended to cope with read-only files by adding `iMode` to `SONOpenOldFile` and the `SONCanWrite` function was added. The return type and `max` argument of the `SONWriteXXXXData` routines were changed from `WORD` to `long`. Added lookup tables into the file to speed up data access in long files and to provide write buffering. Write buffering removes the need for `FastWrite` mode, allows peri-triggered sampling, optimises use of disk space and speeds up writing data to disk. The file commit mechanisms were improved.

Version 6 This version of the library is documented for C/C++ use only, with information on how it can be used as a DLL by other languages. The `RealWave` channel type is added, `AdcMark` channels are allowed to have multiple channels, waveform sample rates are set by `lChanDvd` and the previous `v.adc.divide` now sets the interleave for `AdcMark` data. We have added support for a file time and date stamp, a file basic time unit and support for an application identifier. Files will still be written to be compatible with versions 3, 4 or 5 if they do not use version 6 features. The API was maintained, but some argument types changed: `TDataKind` is an enumerated type and buffer sizes are `long` not `WORD`.

Version 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. See the new `SONExtMarkAlign()` function.

Version 8 Altered storage of channel numbers so that we avoid the bit used to hold the initial state for a level channel, plus extended the maximum number of channels possible to 451. Extended the lookup table system so that the table for a channel starts small and grows up to the limit (larger than the fixed size used previously).

Version 9 Added support for big files (up to 512 GB), rewrote the internal lookup table system to improve speed and now saves the table as part of the file in Big file mode. Support for the Macintosh was removed. Linux support is added. In a version 9 file, all pointers to disk space that were previously byte pointers (but multiples of 512) are now used as block pointers. That is, a pointer value 10 means byte 5120 in the file. This allows us to increase 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.

Reading an existing file

Objectives

In this chapter you will learn how to include the SON library in your C/C++ programs, and how to use it to extract simple information from a file. This chapter does not include a detailed description of the functions used to do this. The functions are described in detail in the C reference section of this document. The examples are symbolic; they will not compile in C as variables are declared where they are used, not at the start of a scope.

Including the library

A program that uses the SON library must include `SON.H` to make the declarations of all the functions and data structures available to the program:

```
#include "son.h"
```

Software that requires low-level access to SON should also include `SONINTL.H`. Generally this should not be required.

Initialising and closing down the library

The first function in the SON library to be used must be `SONInitFiles`. This is not necessary for Windows programs; the DLL calls `SONInitFiles` when the DLL loads:

```
SONInitFiles();
```

`SONInitFiles` marks all files as unused and allocates some work space. Do not call `SONInitFiles` again in your program or you may leak memory and system resources. After the SON library has been used, you must call `SONCleanUp` to free memory and tidy up (this is not required for 32-bit Windows DLL use as the DLL calls this for you when your program detaches from the DLL):

```
SONCleanUp();
```

Opening an existing file

We now want to open an existing data file. Use `SONOpenOldFile` function to open an existing file. The second argument sets the read/write mode (the example allows both). No change is made to the file if no data writing functions are used.

```
short sFh = SONOpenOldFile("testfile.smr" ,0);       /* try to open file */
if (sFh < 0) Error();                                /* error if we fail */
```

`SONOpenOldFile` returns a SON file handle, a positive number, if all was well, otherwise it returns a negative error code. Once a file has been opened successfully, the handle value must be stored and used for all further file accesses.

Reading file information

We will assume that you do not know what each channel of the file holds. The simplest way to get this information is to use the `SONChanKind` function. This returns a variable of type `TDataKind` as defined in `SON.H`:

```
for (i=0;i<SONMaxChans(sFh); ++i)              /* for each data channel */
{
    switch (SONChanKind(sFh, i))
    {
    case ChanOff:   break;               /* do nothing if channel unused */
    case Adc:        printf("%2d is Adc data\n",i); break;
    case EventFall: printf("%2d Event on falling edge\n",i); break;
    case EventRise: printf("%2d Event on rising edge\n",i); break;
    case EventBoth: printf("%2d Event on both edges\n",i); break;
    case Marker:     printf("%2d Marker data\n",i); break;
    case AdcMark:    printf("%2d AdcMark data\n",i); break;
    case RealMark:   printf("%2d RealMark data\n",i); break;
    case TextMark:   printf("%2d TextMark data\n",i); break;
    case RealWave:   printf("%2d RealWave data\n", i); break;
    default:          printf("Error: %2d is unknown type\n");
    }
}
```

This code prints out a simple message for each used channel. The function SONMaxChans() was added at version 6 to report the number of channels of data in the file. The constant SONMAXCHANS is also defined. This is set to 32 and is the number of channels in all files before version 6. All version 6 files have at least 32 channels.

You can get all the available information about a channel from the various channel information functions. These include functions to return the channel type, the channel comment and title and the maximum time in a channel, as well as functions that return the channel information specific to the various types of channel. For example, to get the user comment on a channel and the channel title:

```
char szComment[SON_CHANCOMSZ+1];                  /* SON_CHANCOMSZ is 71 */
SONGetChanComment(sFh, i, szComment, SON_CHANCOMSZ);
printf("%s\n", szComment);                        /* the channel comment */
char szTitle[SON_TITLESZ+1];                         /* SON_TITLESZ is 9 */
SONGetChanTitle(sFh, i, title);
printf("%s\n", title);                             /* the channel title */
```

The SON_ constants are defined in son.h. Notice that you must allocate space for 1 extra character to allow for the zero terminating character. There are also functions that return information about the file as a whole. These include the basic time unit, the time units per clock tick, and the file comment. These can all be obtained by code of the form:

```
WORD usPerTime = SONGetusPerTime(sFh);           /* Get time information */
double dTickLen = SONTimeBase(sFh, 0.0);      /* base tick units, seconds */
printf("%g\n", usPerTime*dTickLen);          /* print the clock tick time */
for (i=0;i<5;i++)                             /* write 5 lines of comment */
{
    char szFComment[SON_COMMENTSZ+1];             /* SON_COMMENTSZ is 79 */
    SONGetFileComment(sFh, i, szComment, SON_CHANCOMSZ); /* Read comment */
    printf("%s\n", szFomment);
}
```

The sampling interval, in clock ticks, for any Adc channel can be obtained by using the SONChanDivide function:

```
long lDivide = SONChanDivide(sFh, 3);      /* get interval for channel 3 */
double dFreq = 1.0/(lDivide*(usPerTime*dTickLen));    /* frequency in Hz */
```

where lDivide is of type long.

Reading an event channel

The SONGetEventData function is used to collect Event, Marker, AdcMark, RealMark and TextMark times from the file and place them in an array of longs (if you want to read marker times and marker data you must use the SONGetMarkData function and you must use SONGetExtMarkData to get at the AdcMark waveforms, the RealMark real data or the TextMark characters). Let us assume that channel 0 holds events. We want to get up to 100 event times from the start of the file (time 0) to 100,000 clock ticks. Use a call of the form:

```
double dTime = SONTimeBase(sFh, 0.0) * SONGetusPerTime(sFh);
long alTimes[100];
long nEv = SONGetEventData(sFh, 0, alTimes, 100, 0, 100000, NULL, NULL);
for (i=0;i<nEv;i++)
    printf("%f\n", alTimes[i]*dTime);    /* display the times in seconds */
```

The function returns the number of events copied to the buffer.

Reading a waveform channel

The function SONGetADCData is used to read contiguous 16-bit integer waveform data between two times from an Adc, AdcMark or RealWave channel. If you read from a RealWave channel, the library uses the scale and offsets set for the channel to convert the float data into 16-bit integers. The function SONGetRealData reads contiguous Adc, AdcMark or RealWave data as float data. Both functions return the number of contiguous points found from the starting time and they also return the time of the first waveform point in a variable. If your waveform data is fragmented (i.e. there are gaps in the recording) you will need to call the function at least once for each fragment. The first point returned is timed at or after the start time of the search.

You could collect waveform data from time 0 to 100,000 clocks ticks into data arrays of 100 elements from channel 1 (Adc or AdcMark data) using:

```
short asData[100];                          /* example short data array */
long beginTime;                 /* for returned time of first data point */
long nP = SONGetADCData(sFh, 1, asData, 100, 0, 100000, &beginTime, NULL);
SONGetADCInfo(sFh, 1, &scale, &offset, &units, &points, &preTrig);
for (i=0; i<nP; ++i)                         /* show data in user units */
    printf("%g\n", asData[i]/6553.6*scale + offset);

float afData[100];                        /* example floating point data */
nP = SONGetRealData(sFh, 1, afData, 100, 0, 100000, &beginTime, NULL);
for (i=0; i<nP; ++i)                         /* show data in user units */
    printf("%g\n", afData[i]);
```

If you select an AdcMark channel and supply a pointer to a marker filter function, only AdcMark events that are accepted by the filter contribute to the result. SONGetADCData and SONGetRealData on an AdcMark channel can return waveforms from multiple events as long as there are no gaps in the waveforms.

Reading Marker data

The SONGetMarkData function is used to read both the marker times and the marker bytes attached to each marker from a Marker, AdcMark, RealMark or TextMark channel. If you only need the marker times you can use the SONGetEventData function described above. If we assume that channel 2 holds Marker (or AdcMark, RealMark or TextMark) data then we can read up to 100 markers from the start of the file up to 100,000 clock ticks with:

```
TMarker amData[100];
long nEv = SONGetMarkData(sFh, 2, amData, 100, 0, 100000, NULL);
for (i=0; i<nEv; ++i)
    printf("%d %d %d %d %d\n", amData[i].mark, amData[i].mvals[0],
           amData[i].mvals[1], amData[i].mvals[2], amData[i].mvals[3]);
```

Reading AdcMark data

Reading AdcMark data is slightly harder than reading any of the above data types because the number of waveform data points (although fixed for a particular channel) is variable. This means that the structure used to hold AdcMark data in memory has to be allocated dynamically, and the programmer has to be responsible for moving pointers forwards and backwards. You can find out the number of points of waveform information attached to the AdcMark using the SONGetExtMarkInfo function. The size, in bytes, of the complete AdcMark is returned by the SONItemSize function. The following code is the sort of thing you will need in your program if you are to read AdcMark data:

```
WORD  nPts, nByt;        /* number of data points and bytes per AdcMark */
int nA, i;                        /* number of AdcMarks read, a counter */
TpAdcMark  amP, wP;          /* pointers to our data, and a work pointer */

SONGetExtMarkInfo(sFh, chan, &units, &nPts, &i);     /* Get # ADC values */
```

The number of bytes to hold each AdcMark structure is given by:

```
nByt = SONItemSize(sFh, chan);                    /* Bytes per data item */
```

To allocate a buffer for 100 AdcMark events and read them you would need code like:

```
amP = (TpAdcMark)malloc(nByt * 100);               /* assume we succeed */
nA = SONGetExtMarkData(sFh, chan, amP, 100, 0, 100000, NULL);
```

To write out the time, the first marker byte and the first waveform point of each AdcMark needs code of the following type:

```
wP = amP;                                        /* pointer to first marker */
for (i=0;i<nA;i++)
{
    printf("Time %d code %d data %d\n", wP->m.mark, wP->m.mvals[0],
             wP->a[0]);                        /* Write out the data value */
    wP = (TpAdcMark)((char*)wP + nByt);          /* move on by nByt bytes */
}
```

Reading RealMark data

Reading RealMark data is very similar to reading AdcMark data in that the number of real numbers associated with the data type (although fixed for a particular channel) is variable. This means that the structure used to hold RealMark data in memory must also be allocated dynamically, and the programmer has to be responsible for moving pointers forwards and backwards. You can find out the number of real numbers attached to the RealMark using the SONGetExtMarkInfo function. The size, in bytes, of the complete RealMark is returned by the SONItemSize function. If you are to read RealMark data you will need code like:

```
WORD  nPts, nByt;      /* number of real numbers and bytes per RealMark */
int nR, i;                        /* number of RealMarks read, a counter */
TpRealMark amP, wP;          /* pointers to our data, and a work pointer */

SONGetExtMarkInfo(sFh, chan, &units, &nPts, &i);  /* Get # real values */
```

The number of bytes to hold each RealMark structure is given by:

```
nByt = SONItemSize(sFh,chan);                     /* Bytes per data item */
```

So, to allocate a buffer to hold 100 RealMark events and read them you would need code like:

```
amP = malloc(nByt * 100);           /* Allocate space, assume we succeed */
nR = SONGetExtMarkData(sFh, chan, amP, 100, 0, 100000, NULL);
```

To write out the time, the first marker byte and the first real number of each RealMark needs code of the following type:

```
wP = amP;                                        /* pointer to first marker */
for (i=0;i<nR;i++)
{
    printf("Time %d code %d data %g\n", wP->m.mark, wP->m.mvals[0],
                wP->r[0]);                    /* Write out the data value */
    wP = (TpRealMark)((char*)wP + nByt);         /* move on by nByt bytes */
}
```

Reading TextMark data

Reading TextMark data is very similar to reading both AdcMark and RealMark data. The only real difference being in the extraction of the text. If you are to read TextMark data you will need code of the form:

```
WORD  nCh, nByt;       /* length of character array, bytes per TextMark */
int nT, i, cnt;                     /* number of TextMarks read, counters */
```

```
TpTextMark amP, wP;          /* pointers to our data, and a work pointer */

SONGetExtMarkInfo(sFh, chan, &units, &nCh, &i);           /* Get # chars */
```

The number of bytes to hold each `TextMark` structure is given by:

```
nByt = SONItemSize(sFh, chan);                    /* Bytes per data item */
```

To allocate a buffer for 100 `TextMark` events and read them you would need code like:

```
amP = malloc(nByt * 100);           /* Allocate space, assume we succeed */
nT = SONGetExtMarkData(sFh, chan, amP, 100, 0, 100000, NULL);
```

To write out the time, the first marker byte and the text of each `TextMark` needs code of the following type:

```
wP = amP;                                       /* pointer to first marker */
for (i=0; i<nT; ++i)
{
    printf("Time %d code %d text ", wP->m.mark, wP->m.mvals[0]);
    for(cnt=0;(cnt<nCh) && (wP->t[cnt]!=0);cnt++)
        printf("%c", wP->t[cnt]);                     /* Print each char */
    printf("\n");
    wP = (TpTextMark)((char*)wP + nByt);        /* move on by nByt bytes */
}
```

Closing the file

When you have finished extracting data from a file you should close it using the function `SONCloseFile`:

```
flag = SONCloseFile(sFh);                 /* Close file, check for error */
```

You can have multiple files open at a time by using the SON library. If open files are not closed using `SONCloseFile` any changes made may not be written to disk and will be lost.

After all files are closed, or when the application is exiting, you must call the function `SONCleanUp` to allow the library to tidy up memory allocation. This is not necessary for WIN32 applications; the `SON32 DLL` does this clean up as part of shutting down.

Writing a new file

Objectives

Creating files is a more complicated process than reading them. In this chapter you will learn the sequence of operations, and the required functions, to create a file.

Opening a new file

The first action in a program that creates a new file is the same as for reading an existing file. This is not required when the SON DLL is used in Windows:

```
SONInitFiles();              /* do once per program, DLLs do this for you */
```

The next step is to create a file on disk and set the number of channels and how much extra space to reserve for storing application specific data. You cannot change the number of channels or the size of the extra space. The minimum (and standard) number of channels is 32; if you use more than 32 channels, programs that use old versions of the SON filing system cannot open the file. This code example is for Windows, the Macintosh version has more arguments (see the reference section):

```
i = SONCreateFile("Gregs.smr", 0, 0);      /* create a standard new file */
if (i < 0)
    Error();                                      /* You must trap errors */
else
    sFh = i;                                  /* Save handle for future use */
```

The first zero parameter in the `SONCreateFile` function sets the number of channels; a zero here means set the standard number, which is 32. The second zero parameter tells the system that we do not require any "extra" space allocated to us in the file header. The function `SONGetExtraData` read and write this space. The format of this area is undefined and is used by programs such as the CED VS system to hold private information. CED suggests that anyone using this area leaves the first 20 bytes undefined (and set to zeros) to allow us to define a convention for use of this area in the future.

Creating a big file

The previous code opens up a file using a disk format the limits the file size to 2 GB on disk. If you need to create large files, you can use:

```
i = SONCreateFileEx("Gregs.smr, 0, 0, 1);           /* create a big file */
```

This has two advantages and one disadvantage. The advantages are that you can create files of up to 512 GB in size and that the lookup table used by the library to locate data in a channel is saved in the file, so you can locate quickly when you open a big file. The disadvantage is that the file cannot be opened by software that used version 8 or earlier of the library.

Set time base information

The next step is to set the file timing information. To set the file clock tick in units other than microseconds, use `SONTimeBase`. Most users do not call this function and use microsecond base clock units.

```
SONTimeBase(sFh, 1.0e-9);          /* set base time units to nanoseconds */
```

The `SONSetFileClock(sFH, usPerTime, timePerAdc)` function sets the number of base time units per clock tick and the number of clock ticks per ADC convert. The number of base clock units per clock tick is vitally important as it determines the time resolution of your file. It is tempting to set this to 1 and use the time base to set your file scaling. However, to write a file that is 100% compatible with previous versions of the SON system you should leave the base time units as microseconds and control the resolution with `usPerTime`.

If you never need to read your file with previous version SON libraries, you can set `timePerAdc` to 1 and forget about it. If you want to write a file that can be read by pre

version 6 libraries, then you must consider the value more carefully. In these versions of the library, the interval between samples of a waveform channel was given by:

`n` * `usPerTime` * `timePerAdc` *1.0e-6 seconds

where n is an integer between 1 and 65535. From version 6 the interval is:

`lChanDvd` * `usPerTime` * `timeBase` seconds

where `lChanDvd` lies between 1 and 2147483648. If the `timeBase` is set to the standard value of 1.0e-6 and `lChanDvd` is equal to `m`*`timePerAdc`, and m is less than 65536, the channel can be represented exactly by the old method. When a SON file closes, the library marks it with the oldest revision that is compatible with the data, so if all channels meet this requirement, older versions of the library can read the data.

The following example sets 10 base time units per tick and sets `timePerAdc` to 100 ticks (1000 base time units).

```
SONSetFileClock(sFh, 10, 100);                    /* set basic file times */
```

File comments

You can set up to five comment strings. These comments can be set at any time when the file is open for writing. The maximum number of characters in a comment is 71 (not including the terminating 0 character).

```
SONSetFileComment(sFh, 0, "The first line of the file comment");
SONSetFileComment(sFh, 1, "Another line of file comment");
   .......
SONSetFileComment(sFh, 4, "The last line of file comment");
```

Define the channels

You must now define each of the channels you want to use. The channel set-up is simple, except for setting the data block size. Data is written to disk as a block containing data for a single channel. All data blocks for a given channel are the same size. Each block has a 20 byte header followed by the appropriate data, some blocks may contain less than the maximum amount of data possible, in which case the rest of the data space is unused. The total size of the block should be a multiple of `DiskBlock` bytes (the number of bytes in a standard disc block, currently 512) to make reading and writing as efficient as possible.

The maximum number of data items in a block is given by the size of the block, in bytes, less 20 bytes for the header, divided by the size of a data item; 2 for waveform data, 4 for `Event` times, 8 for `Marker` and by (8+extraBytes) for `AdcMark`, `RealMark` and `TextMark` data.

The smaller the block you set, the longer it will take to search through your file to find a specific data item, and a larger percentage of your file will taken up by 20 byte block headers. However, the larger your block, the more space will be wasted when a data write is done which does not entirely fill a block (this will rarely happen in the new library). We suggest a size of at least 4 kB. If you are writing fast waveform data you may want to use a size up to 32 kB. We have not tested the library with a block size of greater than 32 kB and many applications cannot handle blocks of larger size.

We will set data channel 0 as an `Event` channel, active on the falling edge of our input signal, using physical channel 0 with a 2048 byte block size, expected event rate 100 Hz, and a channel title and comment:

```
i = SONSetEventChan(sFh, 0, 0, 2048, "Chan comment",
                    "Chan label", 100.0, EventFall);
if (i != 0) Error();                              /* you must trap errors */
```

We could have set this channel to be a `Marker` channel simply by changing `EventFall` to `Marker` in the `SONSetEventChan` function. `SONSetEventChan` is used to define all forms of `Event` channel plus simple `Marker` channels.

Now set data channel 1 as a waveform channel with input on physical channel 5. The channel is to sample at 125 Hz, there are 100000 clock ticks per second, the channel divide is 100000/125, which is 8000. We will set a buffer of size of 1024 bytes, approximately one buffer every 4 seconds. We will set real units of Volts, so we set the channel scale to 1.0 and the offset to 0.0:

```
i = SONSetWaveChan(sFh, 1, 5, 8000, 1024, "Chan comment", "Adc chan 5",
                   1.0f, 0.0f, 'Volts');
if (i != 0) Error();                                  /* trap any problem */
```

Allocate the data transfer buffer

Once you have described all the channels you can tell the library how much memory should be used to buffer data before it is written to disk. Data that is held in buffers can be accessed more quickly than data that is held on disk. This can be important if you a siphoning data to disk in a real-time application that also needs to read back data for display and calculation purposes.

By default, the library allocates one write buffer per channel. You can allocate more buffers per channel with `SONSetBuffering`.

```
flag = SONSetBuffering(sFh, -1, 1000000);            /* 1M write buffers */
if (flag != 0) Error();                             /* check it worked OK */
```

The function doesn't allocate memory; it just calculates how many data buffers to use when writing data. The second parameter is the channel number to which this applies; use -1 to specify all channels, in which case the memory will be shared between channels in proportion to the expected data rates.

The third parameter is the number of memory bytes for write buffering. The minimum memory used is the sum of the buffer sizes you set for all the active channels. The actual memory used will not be more than the suggested size unless the suggested size is less than the minimum. If you do not call `SONSetBuffering`, all channels have 1 write buffer.

The actual allocation of memory is done by `SONSetBuffSpace`. This must be called after the channels have been defined and before you write any data. If you subsequently add a new channel, you must call `SONSetBuffSpace` again before you write data to the new channel.

```
flag = SONSetBuffSpace(sFh);              /* allocate various data buffers */
if (flag != 0) Error();                   /* check there was enough memory */
```

This function finds the biggest block size defined for any channel and allocates this space on the heap. The space is subsequently used as a data transfer buffer for this file. It also allocates the write data buffers and any other memory required. It does not change the allocations for channels that have already had memory allocated for them.

Write data to the file

You are now in a position to write data to the file. You can write to the channels you have defined in the file in any order, however, the data for a particular channel must always be written in ascending time order.

If you have continuous streaming input data, but you only want to save certain sections of it, you have two choices:

1. You write only the sections of data that you want to appear in the file.

2. You write all available to the file then use the SONSave, SONSaveRange and SONKillRange functions to tell the SON library which data should and should not be saved and included in the final disk file. Use of the library in this way allows SON to optimise internal operations and to ensure that disk space is used efficiently. It also increases application flexibility by allowing the save/discard decision to be made retroactively and by making data that is eventually discarded temporarily appear to be part of the file for display purposes.

To write events, fill a long array with the event times as long values. To write the events to channel 1 from a buffer called eBuf holding nEvts events:

```
i = SONWriteEventBlock(sFh, 1, &eBuf, nEvts);
if (i != 0) Error();                                /* check for errors */
```

You can write any number of events or markers; the internal buffering will ensure that data is written to disk efficiently.

Waveform data is written in a similar manner, but you must tell the function the time of the first waveform point in the buffer. The function returns the time of the next waveform point after the buffer, assuming the data is contiguous. You can write non-contiguous data (i.e. with gaps between blocks) but you must NOT write data with blocks that overlap. To write data to channel 1 from the buffer aData, starting at the time in sTime:

```
sTime = SONWriteADCBlock(sFh, 1, &aData, points, sTime);
if (sTime < 0) Error();                        /* failed if negative time */
```

Saving and discarding data

Initially, all data written to the new file is saved on disk and the library can be used in a straightforward manner. The SONSave, SONSaveRange and SONKillRange functions can be used to control which data is written to disk. To flag that data is to be discarded starting at time sTime:

```
i = SONSave(sFh, -1, sTime, FALSE)
if (i != 0) ...
```

This marks all channels to discard data starting at time sTime. The same function can be used to re-enable writing of data to disk at a later time (we assume that eTime contains a later time value). Note that both of these time values might be in the future (i.e. later than any data currently available), the SON library will store them until appropriate:

```
i = SONSave(sFh, -1, eTime, TRUE)
if (i != 0) ...
```

This will produce a data file with a gap running from sTime to eTime. Because of the way the SON filing system stores data, some data will usually be saved for times after sTime and before eTime, the library only guarantees that all data before sTime and after eTime will be saved. In a similar manner, SONSaveRange and SONKillRange can be used to save or discard sections of data, these functions are slightly more efficient than pairs of calls to SONSave, which is intended for use where the current ‘save state’ is changing for an indefinite period.

Closing down the file

While the SON file is being written, you can at any time 'commit' the file to disk by calling SONCommitFile. The effect of this function is to ensure that the file headers on disk are completely up to date and that the current size and location of the file are correctly entered in the disk directory. This ensures that all of the data written to the file up to the last SONCommitFile call can be successfully retrieved, even if your program subsequently crashes completely. Calling SONCommitFile can take a significant period

of time and should be avoided when maintaining a high throughput of data to disk is important.

Once all the data has been written to disk you must call `SONCloseFile` to update the file header on disk, shut down the file in an orderly fashion and release the handle for future use:

```
i = SONCloseFile(sFh);                          /* shut down the file */
if (i != 0) Error();                                  /* trap errors */
```

If you do not call this function the file will most likely not be recognised by the `SONOpenOldFile` function and the handle will not be available for future use.

After all files are closed, or when the application is exiting, you must call the function `SONCleanUp` (this is not required for 32-bit Windows use) to allow the library to tidy up memory allocation.

Defined constants

SON.H defines a number of constants for the SON library. Where constants give string sizes (for titles, comments and units), the value is the maximum number of characters. When reading back such a string, you must allow for an extra character to hold a terminating zero on the end of the string. Characters use the 8-bit ASCII set.

SONABSMAXCHANS	451	The maximum number of channels in a SON file. Channel numbers run from 0 to SONABSMAXCHANS-1.
SONMAXCHANS	32	The minimum number of channels in a SON file. This is the number of channels in all files before version 6.
SON_NUMFILECOMMENTS	5	The maximum number of file comments.
SON_COMMENTSZ	79	The maximum length of a file comment.
SON_CHANCOMSZ	71	The maximum length of a channel comment.
SON_UNITSZ	5	The maximum length of a channel units string.
SON_TITLESZ	9	The maximum length of a channel title string.

A number of other constants are defined to indicate the kind of SON data stored in a particular channel and to aid marker filter setup. These are mentioned with the definition of TDataKind and with the marker filter documentation below.

Error codes

These constants are defined in SON.H as error returns from the SON library. The error values are all negative. As we may change the values of the errors in future releases of the software we urge you to use the names below, not the values.

SON_NO_FILE	-1	The file handle does not refer to an open SON file or an attempt to open a file failed.
SON_NO_HANDLES	-4	There are too many files open in the system (a DOS problem).
SON_NO_ACCESS	-5	Access was denied, for example insufficient privilege, attempt to search for a block in a FastWrite file, attempt to change a file size failed.
SON_BAD_HANDLE	-6	The file you have referenced is not open in the library.
SON_OUT_OF_MEMORY	-8	The system could not allocate enough memory.
SON_NO_CHANNEL	-9	Data channel out of range 0 to SONMaxChans(fh)-1 or no free channel available.
SON_CHANNEL_USED	-10	The channel number supplied is already in use.
SON_CHANNEL_UNUSED	-11	The channel number supplied is not in use.
SON_WRONG_FILE	-13	File header doesn't match a SON file, unknown version, wrong revision, wrong byte order.
SON_NO_EXTRA	-14	Read/write past end of extra data, or no extra data.
SON_OUT_OF_HANDLES	-16	The SON library has run out of file handles, are you closing files after using them?
SON_BAD_READ	-17	A read from the disk resulted in an error.
SON_BAD_WRITE	-18	A write to the file resulted in an error.
SON_CORRUPT_FILE	-19	Internal file information is inconsistent, try SonFix.
SON_READ_ONLY	-21	Attempt to write to a file opened in read only mode.
SON_BAD_PARAM	-22	A function parameter value is illegal or inconsistent.

Some of these codes map to MS-DOS errors (for historical reasons). The Macintosh version of the library may return OS error codes.

Types and structures

The following data types are defined for C/C++ users of the SON library. Some are defined in the `MACHINE.H` include file that attempts to compensate for machine and environment difference, most are defined in the `SON.H` definition of the SON library:

Simple types

```
typedef unsigned char BOOLEAN;
typedef unsigned short WORD;
typedef unsigned long DWORD;
typedef short TAdc;
typedef TAdc FAR * TpAdc;
typedef long TSTime;
typedef TSTime FAR *TpSTime;
typedef char FAR * TpStr;
typedef const char FAR * TpCStr;
typedef WORD FAR * TpWORD;
typedef BOOLEAN FAR * TpBOOL;
typedef float FAR * TpFloat;
typedef void FAR * TpVoid;
typedef short FAR * TpShort;
```

The `FAR` qualifier is a relic of 16-bit DOS and compiles away to nothing except in 16-bit DOS builds of the library.

The `TSTime` type is a formal definition of the data type used by SON to hold a time value, similarly the `TAdc` type defines an ADC value. The other types provide formal definitions of pointers to various types of data, both for use as function parameters and to ensure that `FAR` pointers are used where necessary. `FAR` is ignored where it is irrelevant (Mac and 32-bit Windows).

When a `TpCStr` argument is used to pass a text string to an exported function (comments, titles and units), the string is always terminated by a zero byte. The string can be longer or shorter than the maximum length specified. If it is longer, only characters up to the specified length are used.

When a string is read back, we use the `TpStr` type. The storage area supplied must be long enough for the maximum string length plus the zero terminating character.

TDataKind

This type defines the kind of data that is stored in each channel. The `RealWave` type was new at version 6. The `AdcMark` type was extended at version 6 to allow interleaved data.

```
typedef enum
{
    ChanOff=0,          /* the channel is OFF - */
    Adc,                 /* a 16-bit waveform channel */
    EventFall,          /* Event times (falling edges) */
    EventRise,          /* Event times (rising edges) */
    EventBoth,          /* Event times (both edges) */
    Marker,             /* Event time plus 4 8-bit codes */
    AdcMark,            /* Marker plus Adc waveform data */
    RealMark,           /* Marker plus float numbers */
    TextMark,           /* Marker plus text string */
    RealWave            /* waveform of float numbers */
} TDataKind;
```

TSONTimeDate

```
typedef struct
{
    unsigned char ucHun;   /* hundreths of a second, 0-99 */
    unsigned char ucSec;   /* seconds, 0-59 */
    unsigned char ucMin;   /* minutes, 0-59 */
    unsigned char ucHour;  /* hour - 24 hour clock, 0-23 */
    unsigned char ucDay;   /* day of month, 1-31 */
    unsigned char ucMon;   /* month of year, 1-12 */
    WORD wYear;            /* year 1980-65535! */
}TSONTimeDate;
```

TSONCreator

```
typedef struct
{
    char acID[8];
} TSONCreator;                /* application identifier */
```

Marker types

The `TMarkBytes` and `TMarker` definitions together give the definition of a Marker, a time value with 4 bytes of attached data. The `TAdcMark`, `TRealMark` and `TTextMark` types are all markers with an attached array of data, only the type of data varies. The length of the attached data array is variable; the lengths given are nominal maximum lengths. All of these structures are packed on 2-byte boundaries.

```
typedef char TMarkBytes[4]
typedef TMarkBytes FAR * TpMarkBytes;

typedef struct
{
    TSTime mark;              /* Marker time as for events */
    TMarkBytes mvals;         /* the marker values */
} TMarker;

#define SON_MAXADCMARK 1024   /* maximum points in AdcMark (arbitrary) */
#define SON_MAXAMTRACE 4      /* maximum interleaved traces in AdcMark */
typedef struct
{
    TMarker m;                /* the marker structure */
    TAdc a[SON_MAXADCMARK*SON_MAXAMTRACE]; /* the attached ADC data */
} TAdcMark;

#define SON_MAXREALMARK 512   /* maximum points in RealMark (arbitrary) */
typedef struct
{
    TMarker m;                /* the marker structure */
    float r[SON_MAXREALMARK]; /* the attached floating point data */
} TRealMark;

#define SON_MAXTEXTMARK 2048  /* maximum points in Textmark (arbitrary) */
typedef struct
{
    TMarker m;                /* the marker structure */
    char t[SON_MAXTEXTMARK];  /* the attached text data */
} TTextMark;

typedef TMarker FAR * TpMarker;
typedef TAdcMark FAR * TpAdcMark;
typedef TRealMark FAR * TpRealMark;
typedef TTextMark FAR * TpTextMark;
```

Marker filter types

These define the structure used to hold a marker filter; a definition of which markers are wanted. See below for more on marker filtering.

```
#define SON_FMASKSZ 32                      /* # of TFilterElt in mask */
typedef unsigned char TFilterElt;           /* element of a map */
typedef TFilterElt TLayerMask[SON_FMASKSZ]; /* 256 bits in the bitmap */

typedef struct
{
    long lFlags;            /* private flags used by marker filering */
    TLayerMask aMask[4];    /* set of masks for each layer */
} TFilterMask;
typedef TFilterMask FAR *TpFilterMask;


#define SON_FMASK_ORMODE 0x02000000  /* use OR if data rather than AND */
#define SON_FMASK_VALID  0x02000000  /* bits that are valid in the mask */
#define SON_FALLLAYERS  -1
#define SON_FALLITEMS   -1
#define SON_FCLEAR       0
#define SON_FSET         1
#define SON_FINVERT      2
#define SON_FREAD       -1
```

Incrementing pointers to marker types

It is often necessary to iterate through marker data. For example, you might wish to write code that can be used with a buffer of TSTime, TMarker, TAdcMark, TRealMark or TTextMark data. You can use SONItemSize() to get the size of each item, then you need to do some casting to increment your pointer. For example:

```
void DoSomething(short fh, WORD wChan, TTextMark* pTM, int n)
{
    int iSize = SONItemSize(fh, wChan);    /* get size increment /
    while (--n >= 0)   /* print all times and comments */
    {
        printf("%10ld %s\n", pTM->m.mark, pTM->t);
        pTH = (TTextMark*)((char*)pTM + iSize);
    }
}
```

If you do a lot of this, you may prefer to create some macros:

```
#define IncTextMark(p, inc) (p = (TTextMark*)((char*)p + inc))
#define IncRealMark(p, inc) (p = (TRealMark*)((char*)p + inc))
#define IncAdcMark(p, inc) (p = (TAdcMark*)((char*)p + inc))
```

Then you can rewrite the line that moves the pointer as:

```
        IncTextMark(pTH, iSize);
```

Filtering markers

Filtering of markers is built into the SON library; all read operations on a Marker or extended Marker channel can use a filter so that only some markers are read. Every Marker, TextMark, RealMark and AdcMark data item has 4 marker bytes attached. The filter specifies which values of these codes are wanted.

Marker filtering is implemented through the TFilterMask data structure (defined above). For example, the GetExtMarkData function has the parameter pFiltMask, which is a pointer to the filter data structure. When GetExtMarkData is called, the data in the filter structure is used to check each marker to see if it is required; only those markers passing this test are returned to the calling program. If the marker filter pointer passed to GetExtMarkData is NULL, all markers are accepted.

A marker filter consists of four layers, these correspond to the four marker bytes attached to a marker time. Each layer contains 256 items, these correspond to the 256 different values that a marker byte can have. Each item in a layer can be set, meaning that markers with the corresponding value are wanted, or clear for unwanted.

The SONFMode function switches filtering between AND mode, where all four marker byte values must be passed by the corresponding layer, and OR mode, where only the first layer is used, and the marker is accepted if any of the four marker byte values are passed by the first layer.

The function SONFControl manipulates marker filter data without knowledge of the filter structure. For example, to produce a filter where only the first byte is checked (other three layers are all set) and only markers with the first byte as zero or 1 are passed, do the following:

```
TFilterMask  sFM;                              /* Marker filter structure */

SONFControlsFM, SON_FALLLAYERS, SON_FALLITEMS, SON_FSET);    /* all-pass */
SONFControl(&sFM, 0, SON_FALLITEMS, SON_FCLEAR);        /* Clear layer 0 */
SONFControl(&sFM, 0, 0, SON_FSET);                    /* Allow byte 0 = 0 */
SONFControl(&sFM, 0, 1, SON_FSET);                    /* Allow byte 0 = 1 */
.
.                                 /* The filter structure can now be used */
```

List of C functions

This is a list of all the functions in the SON library, ordered roughly by function. If you only need to read data from SON files you will require very few of these functions. Only users who need the fastest possible use of the system and who intend to write directly to the files will need to use all the functions described below.

`SONAppID`	Get or set application identifier in file header
`SONBlocks`	Get number of data blocks for a channel
`SONChanBytes`	Get bytes held in a particular channel
`SONChanBytesD`	Get bytes held in a channel as a double (for big files)
`SONCanWrite`	Is the file read/write or write only
`SONChanDelete`	Delete a channel from a file
`SONChanDivide`	Get the divide down for a waveform channel
`SONChanInterleave`	Get the interleave for an `AdcMark` channel
`SONChanKind`	Get the channel type
`SONChanMaxTime`	Return time of last data item in the channel
`SONCleanUp`	Clean up SON system on application exit
`SONCloseFile`	Close an open SON file and release resources used
`SONCommitFile`	Flush all buffered data to disk (as far as possible)
`SONCommitIdle`	Does nothing
`SONCreateFile`	Create a new SON file
`SONCreateFileEx`	Create a new SON file with big file option
`SONDelBlocks`	Get the number of deleted blocks for a channel
`SONEmptyFile`	Junk all data, preserve channel settings
`SONFControl`	Routine to fill in a marker filter
`SONFEqual`	Test if two marker filters are equivalent
`SONExtMarkAlign`	Handle extended marker type buffer alignment
`SONFileBytes`	Get number of data bytes written and buffered to the file
`SONFileSize`	Get size of disk file (includes buffered data)
`SONFileSizeD`	Get size of disk file as a double (for big files)
`SONFilter`	Test a marker against a marker filter
`SONFMode`	Set and get the filter maker mode
`SONGetADCData`	Read `Adc` or `AdcMark` data to a buffer
`SONGetADCInfo`	Get data on `Adc`, `AdcMark` or `RealWave` channels
`SONGetChanComment`	Get a channel comment
`SONGetChanTitle`	Get a channel title
`SONGetEventData`	Read event or marker data as times
`SONGetExtMarkData`	Read extended marker data
`SONGetExtMarkInfo`	Read information about extended marker data
`SONGetExtraData`	Read/write to the extra data area
`SONGetExtraDataSize`	Get the size of the extra data area
`SONGetFileComment`	Get a file comment
`SONGetFreeChan`	Search for an unused channel number
`SONGetIdealLimits`	Deprecated; use `SONIdealRate`, `SONYRange` instead
`SONGetMarkData`	Read Marker data from marker or extended marker
`SONGetRealData`	Read `RealWave`, `Adc` or `AdcMark` data as floating point
`SONGetTimePerADC`	Get the `timePerADC` value
`SONGetusPerTime`	Get the base time units per clock tick
`SONGetVersion`	Get the version of the file system
`SONIdealRate`	Get/set the ideal sample rate or expected event rate
`SONInitFiles`	One off SON system initialise call
`SONIsBigFile`	Is the file in big file mode or not
`SONIsSaving`	Is data save to disk on or off for a channel
`SONItemSize`	Get size of data item for a channel
`SONKillRange`	Mark a time range as not wanted on disk
`SONLastPointsTime`	To find data when searching backwards n points
`SONLastTime`	Find last item on a channel before a given time
`SONLatestTime`	Mark data before a time as valid as is for save/not save
`SONMarkerItem`	Move extended marker data to and from buffer

SONMaxChans	Get maximum channel supported by this file
SONMaxItems	Get maximum items per buffer for this channel
SONMaxTime	Get the time of last data item in the file from file header
SONOpenNewFile	Deprecated; use SONCreateFile. Create a SON file.
SONOpenOldFile	Open an existing SON file.
SONPhyChan	Get the physical channel associated with a data channel.
SONPhySz	Get the size of the channel buffer.
SONSave	Mark data after a particular time is to be saved to disk
SONSaveRange	Mark a time range of data is to be saved to disk
SONSetADCChan	Deprecated; use SONSetWaveChan
SONSetADCMarkChan	Deprecated; use SONSetWaveMarkChan
SONSetADCOffset	Set Adc or AdcMark user units equivalent of 0
SONSetADCScale	Set Adc or AdcMark user units scale factor
SONSetADCUnits	Set units for Adc, AdcMark, RealWave and RealMark
SONSetBuffering	Suggest memory to use for write buffering
SONSetBuffSpace	Prepare buffer space for channels
SONSetChanComment	Set channel comment
SONSetChanTitle	Set channel title
SONSetEventChan	Create Event or Marker channel
SONSetFileClock	Set basic file timings in base time units
SONSetFileComment	Set a file comment
SONSetInitLow	Set initial state of EventBoth channel
SONSetMarker	Edits the data in a Marker or extended marker
SONSetRealChan	Create a RealWave channel
SONSetRealMarkChan	Create a RealMark channel
SONSetTextMarkChan	Create a TextMark channel
SONSetWaveChan	Create an Adc channel
SONSetWaveMarkChan	Create an AdcMark channel
SONTimeBase	Set/get the base time units for the file
SONTimeDate	Set/get the time stamp for the file
SONUpdateStart	Write the file header to the disk
SONWriteADCBlock	Write Adc data
SONWriteEventBlock	Write Event data
SONWriteExtMarkBlock	Write extended marker data types
SONWriteMarkBlock	Write marker data
SONWriteRealBlock	Write RealWave data
SONYRange	Get y range of data for a channel
SONYRangeSet	Set expected min/max range of RealWave and RealMark

Internal functions

These functions are used internally by the SON library. They were visible in old versions of the library and some were used by specialised programs. However, you should not use these functions; we make no promise to maintain them in future versions of the library.

SONBookFileSpace	Reserve disk space for use by the file
SONChanPnt	Deprecated; gives access to internal data structures
SONExtendMaxTime	Increases the maximum file time
SONFileHandle	Deprecated; gives access to the OS file handle
SONFindBlock	Locates a data block in a time range
SONGetBlock	Reads first 512 bytes of data block to internal buffer
SONGetFirstData	Returns offset to start of the data area (past headers)
SONGetPred	Get previous data block (using linked list)
SONGetSucc	Get next data block (using linked list)
SONIntlChanMaxTime	Get last time in last channel block (reads data from disk)
SONIntlMaxTime	Search channels for last time in file
SONRead	Low level routine to read from SON file
SONRead64	Low level routine to read at a 64-bit offset

SONReadBlock	Read channel data block to channel data buffer
SONSetPhySz	Set size of a channel buffer – very specialised
SONSetSucc	Set the link list pointer on disk to next channel block
SONUpdateMaxTimes	Updates the maximum file time in the file header
SONWrite	Low level routine to write to SON file
SONWrite64	Low level routine to write to SON file at 64-bit offset
SONWriteBlock	Appends the current channel block to channel data

Alphabetical function list

This section of the manual lists the public exported functions in alphabetical order. There is a short description of each routine in the previous chapter.

SONAppID

This routine will read back and set the optional application identifier that is stored in the file header. The identifier is stored as a `TSONCreator` structure, which contains 8 ASCII characters, one byte per character. You can put anything you like into these 8 bytes, they are not used by the SON library in any way. A typical use might be to identify the application that created the file.

```
int SONAppID(short fh, TSONCreator* pCGet, const TSONCreator* pCSet);
```

`fh`	The file identifier.
`pCGet`	If not `NULL`, the current identifier is copied to here.
`pCSet`	If not `NULL`, the identifier source. The two pointers should not be the same!
Returns	0, or an error code.
Errors	`SON_NO_FILE`

SONBlocks

This routine returns the number of data blocks that have been written to disk for this channel. The returned value does not include blocks held in write buffers or deleted blocks that belong to this channel.

```
int SONBlocks(short fh, WORD wChan);
```

`fh`	The file handle.
`chan`	The channel to get the information for, in the range 0 to `SONMaxChans(fh)`.
Returns	The number of data blocks for the channel that are on disk or 0 if any error.

SONCanWrite

Use this to test if a SON file is read-only. Read only files can be read without problems, but any attempt to change the data in the file will cause a `SON_READ_ONLY` error.

```
BOOLEAN SONCanWrite(short fh);
```

`fh`	The file handle.
Returns	`TRUE` if the file can be written to, otherwise `FALSE`.

SONChanBytes

This returns the channel data size or `0xffffffff` if the size exceeds this value (being the largest positive value returnable in a `DWORD`). This takes account of data not yet written, but still in the buffers. The intended use is to detect an empty channel; it does not allow for partially filled buffers or blocks so is not a precise measure of channel size. You should use `SONChanBytesD()` for big files.

```
DWORD SONChanBytes(short fh, WORD chan);
```

fh	The file handle.
chan	The channel number.
Returns	The estimated number of bytes of data for the channel or 0 if there is any error.

SONChanBytesD

This returns the estimated number of bytes held by the library for a channel, including bytes in buffers not yet written to disk. It assumes that all blocks on disk and buffers in memory are full.

```
double SONCHanBytesD(short fh, WORD chan);
```

fh The file handle.

chan The channel number.

Returns The estimated number of bytes of data for the channel or a negative error code if there is any error.

Errors `SON_NO_FILE`, `SON_NO_CHANNEL`

SONChanDelete

This function marks a channel in an existing file as deleted. This allows you to re-use the channel, but only using the block size that the channel was originally written with. When more information is written, the original space used by the channel is used first. Do not use this function with a file which has been opened with `OpenNewFile`; it will not operate correctly. If the channel is not in use nothing is done. It is not possible to un-delete a channel using the C SON library.

```
short SONChanDelete(short fh, WORD chan);
```

`fh` The file handle.

`chan` The channel number to be marked as deleted.

Returns Zero if the deletion was accomplished without error or a negative error code.

Errors `SON_BAD_READ`, `SON_BAD_WRITE`

SONChanDivide

This function is used to get the number of clock ticks per waveform conversion for an `Adc` or `AdcMark` channel. To get the time per conversion in base time units you must multiply this value by the value returned by the `SONGetusPerTime()` function.

```
TSTime SONChanDivide(short fh, WORD chan);
```

`fh` The file handle.

`chan` The channel to get the information from.

Returns The interval, in clock ticks, between waveform conversions on this channel. If this is not a waveform channel, then the return value is 1.

Errors This function returns no errors.

SONChanInterleave

This function returns the interleave factor for `AdcMark` channels. This will be 1 for all files created before version 6 and can be up to 4 for version 6 files.

```
SONAPI(int) SONChanInterleave(short fh, WORD chan)
```

`fh` The file handle.

`chan` The channel to get the information from.

Returns The interleave for an `AdcMark` channel or 1 for any other (including channels that are off).

Errors `SON_NO_FILE`.

SONChanKind

This function returns the channel type as a `TDataKind` value.

```
TDataKind SONChanKind(short fh, WORD chan);
```

`fh` The file handle.

`chan` The channel number for which we require the kind.

Returns The kind of the channel.

Errors The function does not return any errors. It returns `ChanOff` if the channel does not exist or if the file handle is invalid.

SONChanMaxTime

This function is used to obtain the maximum time for data stored in a particular channel.

```
TSTime SONChanMaxTime(short fh, WORD chan);
```

`fh` The file handle.

`chan` The number of the channel for which we require the maximum time.

Returns The maximum time for the channel, in clock ticks.

Errors No errors are returned (no checks in non-debug build).

SONCleanUp

This function must be called after all use of the SON library is over and all files are closed. It frees memory allocated to hold the file tables and carries out any other tidying required. It is not necessary to call this function from 32-bit Windows programs, as the `SON32 DLL` makes this call automatically. In other situations, if this function is not called memory problems or leaks may occur.

```
void SONCleanUp(void);
```

Errors None.

SONCloseFile

This function is used to close a file that has been opened for reading or for writing. If a file was opened for reading, or for writing in `NormalWrite` mode, the file header information is written back to the file. For all files the file is closed and the memory areas used to hold file information and data are released.

If the file was opened for writing in `FastWrite` mode, all the backward links are written in addition to the above operations.

```
short SONCloseFile(short fh);
```

`fh` The file handle.

Returns The function returns 0 if all went well or an error code if the file close operation failed.

Errors `SON_NO_FILE`, `SON_BAD_READ`, `SON_BAD_WRITE`

SONCommitFile

This function writes all buffered data to disk, leaving the file open. It is intended to be called at intervals while data is being written to a new file, as time permits.

```
short SONCommitFile(short fh, BOOLEAN bDelete);
```

`fh` The file handle.

`bDelete` Set `TRUE` to delete all the channel data buffers, `FALSE` otherwise. This argument should be set `TRUE` when all data has been written to the file.

Returns Zero or an error code.

Errors SON_NO_FILE

SONCommitIdle

This function does nothing in the current build, but is intended to be called intermittently during creation of a new file to allow the SON library to respond to passing time.

```
short SONCommitIdle(short fh);
```

fh The file handle.

Returns Zero or an error code.

Errors SON_NO_FILE

SONCreateFile

This function was new at version 6 and replaces SONOpenNewFile. It creates a new SON file with a user-defined number of channels and extra data space. The header block for the file and the channel description tables are cleared and written to the disk and any extra data area is reserved. The number of channels or the extra data space is fixed after this call. The file opened is not a big file.

```
short SONCreateFile(TpStr pcName, int nChans, WORD extra)
```

pcName The name of the file to create. This can optionally include a drive and path. If a file of this name already exists it is truncated to zero length.

nChans The number of channels to set for this file in the range 32-451. It is an error to set more that 451 channels. If you request less than 32 channels, space is reserved for 32 channels.

extra The number of bytes to reserve as extra space in the file header. This space can be accessed through the SONGetExtraData function.

Returns The function returns a positive file handle to be used in all future accesses if the new file was opened without error, or a negative error code.

Errors SON_NO_FILE, SON_OUT_OF_MEMORY, SON_NO_HANDLES, SON_NO_ACCESS

SONCreateFileEx

This function is new at version 9 and creates a new data file with the option of opening in big file mode, which allows files of up to 512 GB in size and saves lookup tables of channel positioning information as part of the file. Big files cannot be read by older versions of the library.

```
short SONCreateFileEx(TpStr pcName, int nChans, WORD extra, int iBig)
```

pcName The name of the file to create. This can optionally include a drive and path. If a file of this name already exists it is truncated to zero length.

nChans The number of channels to set for this file in the range 32-451. It is an error to set more that 451 channels. If you request less than 32 channels, space is reserved for 32 channels.

extra The number of bytes to reserve as extra space in the file header. This space can be accessed through the SONGetExtraData function.

iBig If this is 0, the file is created as a version 8 file with a size limit of 2 GB. If this is non-zero, the file is created as version 9 with a size limit of 512 GB.

Returns The function returns a positive file handle to be used in all future accesses if the new file was opened without error, or a negative error code.

Errors SON_NO_FILE, SON_OUT_OF_MEMORY, SON_NO_HANDLES, SON_NO_ACCESS

SONDelBlocks

This routine returns the number of deleted data on disk for this channel. Deleted blocks are created when a channel is deleted. SON reuses deleted blocks if the channel is used again. Beware that this can cause inefficiencies as the block size is set by the previous channel use and may not be appropriate for the new use.

```
int SONDelBlocks(short fh, WORD wChan);
```

`fh` The file handle.

`chan` The channel to get the information for, in the range 0 to `SONMaxChans(fh)`.

Returns The number of deleted data blocks for the channel.

Errors This function returns no errors.

SONEmptyFile

This function deletes all of the data from all of the channels in a SON file, and resets the file length to the space occupied by the file headers only. This returns the file to the state it was in immediately after the `SONSetBuffSpace` function was used. This function is intended to allow data sampling to be quickly restarted when necessary.

```
short SONEmptyFile(short fh);
```

`fh` The file handle.

Errors `SON_BAD_WRITE`

SONExtMarkAlign

It is possible for TextMark and AdcMark channels to have data items that are not a multiple of 4 bytes in length. This can make access to the Marker time problematic on some systems as not all processors allow non-aligned data access. From version 7 onwards, you can set a flag in the file header to force all channels to round up the item size (returned by `SONItemSize()`) to a multiple of 4 bytes. Of course, to use this you must make sure that all use of the data in such channels uses the `SONItemSize()` value when indexing through data.

```
int SONExtMarkAlign(short fh, int n);
```

`fh` The file handle.

`n` This should have one of the following values:

- -2 Check the alignment of all the channels in the file and return 1 if they are aligned and 0 if not aligned. This ignores the flag in the file header.
- -1 Report the state of the alignment flag in the file header.
- 0 Set the flag in the file header to make the file unaligned. You cannot do this if the file is marked to align and there are any channels already in use that are rounding up the item size. This returns 0 if the file is now not aligned and 1 if you cannot change the state.
- 1 Set the flag in the file header to make the file aligned. You cannot set the file to be aligned if there are any channels that are unaligned. This returns 1 if the file is now aligned and 0 if you cannot change the state.

Returns A negative error code or an alignment state: 0=not aligned, 1=aligned.

Errors `SON_NO_FILE, SON_BAD_PARAM`

SONFControl

The `SONFControl` function will read, set, clear or invert individual bits in one or all layers of a filter mask structure. It is supplied to allow programmers to manipulate filter masks without using knowledge about the filter mask data structure.

```
int SONFControl (TpFilterMask pFM, int layer, int item, int set);
```

pFM	Pointer to the filter mask structure to be modified.
layer	The layer in the filter mask to be modified, in the range 0 to 3. Use the defined constant SON_FALLLAYERS to select all layers and all items; in this case the item parameter is ignored.
item	The item to modify in the layer, in the range 0 to 255. Use the defined constant SON_FALLITEMS to select all items.
set	This defines the operation to carry out on the filter mask. Use one of these predefined constants: SON_FREAD to read the item, SON_FCLEAR to clear items (remove corresponding marker from required set), SON_FSET to set items (add corresponding marker(s) to required set) and SON_FINVERT to invert items.
Returns	The state of the specified item or a negative error code for a SON_FREAD operation: zero if the item is clear, 1 if it is set. For reads of whole layers and all layers returns 1 if layer(s) all set, else zero. For write operations returns zero or a negative error code.
Errors	If the layer, item or set values supplied were illegal, returns -1.

SONFEqual

The SONFEqual function tests two filter masks for equality.

```
BOOLEAN SONFEqual(TpFilterMask pMask1, TpFilterMask pMask2)
```

pMask1	Points to the first marker filter mask being tested for equality.
pMask2	Points to the second marker filter mask being tested.
Returns	FALSE if the marker filter masks are not the same, TRUE if they are.
Errors	This function returns no errors.

SONFileBytes

This function returns the number of bytes of data either written to the file or buffered and ready to be written. It is intended for use as an unequivocal test for a non-empty file, use SONFileSize if you are interested in the overall file size on disk.

```
long SONFileBytes(short fh);
```

fh	The file handle.
Returns	The calculated number of bytes of data in the file, or an error code.
Errors	SON_NO_FILE

SONFileSize

This function returns the expected disk size of a new data file if it were closed now.

```
long SONFileSize(short fh);
```

fh	The file handle.
Returns	The size of the data file in bytes or an error code.
Errors	SON_NO_FILE

SONFilter

The SONFilter function is used to decide if a given marker is part of a set of required markers defined by a filter mask. This function is normally only used internally.

```
int SONFilter(TpMarker pM, TpFilterMask pFM);
```

pM	Points to the marker data to be tested.

pFM Points to the marker filter mask used to define the required set.

Returns Zero if the marker is not part of the set defined by the filter mask, 1 if it is.

Errors This function returns no errors.

SONFMode

This function reads and changes the marker filtering mode between AND mode and OR mode. In AND mode each of the 4 marker codes must match each of the 4 filter layers. In OR mode only the first filter layer is used and the marker matches the filter if any code matches the filter. In OR mode a marker code of 0 is only accepted if is the first code.

```
long SONFMode(TpFilterMask pFM, long lNew)
```

pFM Pointer to the filter mask to be affected.

lNew This is used to change the filter mode unless it is –1, which means no change. Valid new modes are SON_FMASK_ORMODE and SON_FMASK_ANDMODE.

Returns The previous marker filter mode value.

Errors None.

SONGetADCData

This function reads contiguous waveform data from an Adc, AdcMark or RealWave channel in the current file between two times into a user-defined buffer. If you read an AdcMark channel, the data returned can be from more than one AdcMark structure if the AdcMark records overlap so that a longer contiguous record is available. If you read a RealWave channel, the float data in the file is converted to 16-bit integers using the scale and offset of the channel.

```
long SONGetADCData(short fh, WORD chan, TpAdc psData, long max,
              TSTime sTime, TSTime eTime, TpSTime pbTime,
              TpFilterMask pFltMask);
```

fh The file handle.

chan The data channel in the file to search for the data.

psData Points to the area of memory in which any data found is to be returned.

max The maximum number of data points which may be returned into the area pointed at by adcDataP.

sTime The start time of the search in clock ticks. No data point will be returned which was timed before this time.

eTime The end time of the search. No data point will be returned which was timed after this time.

pbTime Points to a TSTime variable to hold the time of the first data point located.

pFltMask Only used when reading from an AdcMark channel. Points to the TFilterMask structure to be used to filter the data. If this parameter is NULL, no filter is used and all AdcMark data is accepted.

Returns The number of data points returned or a negative error code.

Note that this function only returns contiguous data. If the end of the data area does not reach the end of the time period you should request more data from beyond the last data point returned to you:

```
newSTime = bTime + (points * SONChanDivide(fh, chan);
```

where points is the number of data points that were returned.

Errors SON_BAD_READ, SON_NO_CHANNEL

SONGetADCInfo

This function retrieves information about an Adc, AdcMark or RealWave channel.

```
void SONGetADCInfo(short fh, WORD chan, TpFloat scale, TpFloat offset,
                   TpStr pcUnt, TpWORD points, TpShort preTrig);
```

fh	The file handle.
chan	The channel number of an Adc or AdcMark channel.
scale	NULL or pointer to a float to hold the channel scale factor.
offset	NULL or pointer to a float to hold the channel data offset.
pcUnt	NULL or pointer to a character array to hold the channel units. This array must be at least SON_UNITSZ+1 characters long.
points	NULL or pointer to a WORD to hold the number of ADC points for an AdcMark channel. For a simple Adc channel, this variable is set to 1.
preTrig	NULL or pointer to a short to hold the pre-trigger value. For an AdcMark channel this is the number of waveform points sampled prior to the trigger point. For an Adc channel, the result is 0.

SONGetChanComment

This function retrieves the channel comment string for a given channel. The comment consists of a string up to SON_CHANCOMSZ (71) characters long.

```
void SONGetChanComment(short fh, WORD chan, TpStr pcCom, short max);
```

fh	The file handle.
chan	The channel number for which the comment is required.
pcCom	Points to a character array to hold the returned string.
max	The maximum number of characters that can be stored in the array pointed to by comment. The character array should be 1 larger than max, to leave space for the terminating zero byte.

SONGetChanTitle

This function is used to retrieve the channel title for a given channel. The title consists of a string up to SON_TITLESZ (9) characters long.

```
void SONGetChanTitle(short fh, WORD chan, TpStr pcTitle);
```

fh	The file handle.
chan	The channel number for which the title is required.
pcTitle	This points to a character array to hold the returned string. This array must be large enough to hold the title, SON_TITLESZ+1 (10) characters.

SONGetEventData

This function will read event data between two times from an Event, Marker, AdcMark, RealMark or TextMark channel into a user defined buffer.

```
long SONGetEventData(short fh, WORD chan, TpSTime plTimes, long max,
                     TSTime sTime, TSTime eTime, TpBOOL levLow,
                     TpFilterMask pFltMask);
```

fh	The file handle.
chan	The channel number of an Event, Marker or extended Marker.
plTimes	Points to the user defined buffer in which the event data is to be stored. Each event requires 4 bytes for the long number returned.
max	The maximum number of events to be returned, which should be the size of the long array used to hold the data.

`sTime` The start of the time range for which we require events in clock ticks.

`eTime` The end of the time range. We will accept events up to and including this time.

`levLow` This is for events of kind `EventBoth`. The `BOOLEAN` variable pointed to is set `TRUE` if the first event after `sTime` was a transition from a high to low state, otherwise it is set `FALSE`.

`pFltMask` Points to a `TFilterMask` structure defining a marker filter. This structure will be used to filter all types of `Marker` channel. If this parameter is `NULL`, no filter will be used and all markers accepted.

Returns The number of events returned, or a negative error code.

Errors `SON_BAD_READ`, `SON_NO_CHANNEL`

SONGetExtMarkData

Use this function to collect any type of extended or simple `Marker` data from a specified channel between two times into a user-defined buffer.

```
long SONGetExtMarkData(short fh, WORD chan, TpMarker pMark, long max,
              TSTime sTime, TSTime eTime, TpFilterMask pFltMask);
```

`fh` The file handle.

`chan` The number of a `Marker`, `AdcMark`, `RealMark` or `TextMark` channel.

`pMark` Points to a buffer to hold the data read. Remember that you must calculate the size of the extended `Marker` item or use the `SONItemSize` function to return this size, as the size can vary according to the channel definition. The buffer must be at least `SONItemSize * max` bytes long.

`max` The maximum number of extended `Marker` items to read.

`sTime` The start time of the search in clock ticks. No data will be returned which has a time before this time.

`eTime` The end time of the search in clock ticks. No data point will be returned which was timed after this time.

`pFltMask` Points to the structure used to filter markers. This structure will be used to filter all types of `Marker` channel. If this parameter is `NULL`, no filter will be used and all markers accepted.

Returns The number of items read into the buffer, or a negative error code.

Errors `SON_BAD_READ`, `SON_NO_CHANNEL`

SONGetExtMarkInfo

This function is used to retrieve information about any extended `Marker` (`AdcMark`, `RealMark` or `TextMark`) channel.

```
void SONGetExtMarkInfo(short fh, WORD chan, TpStr pcUnt, TpWORD points,
              short* preTrig);
```

`fh` The file handle.

`chan` The channel number of an `Adc` or `AdcMark` channel.

`pcUnt` Points to a character array to hold the channel units. This array must be at least `SON_UNITSZ`+1 characters long.

`points` Points to a `WORD` to hold the number of points of extended data attached to the `Marker`.

`preTrig` Points to a short to hold the pre-trigger value for an `AdcMark` channel, as for `SONGetAdcInfo`.

SONGetExtraData

Use this function to read and write the extra data in the file header of a SON file. The size of the extra data area is set when the file is created by the SONCreateFile call. This function can be used both on newly created files, primarily to write to the extra data, and on old files, to read the data.

```
short SONGetExtraData(short fh, TpVoid buff, WORD bytes,
                                          WORD offset, BOOLEAN  writeIt);
```

fh	The file handle.
buff	A pointer to the memory region to be used for the transfer.
bytes	The number of bytes to be moved between the disk and the buffer.
offset	The byte offset into the extra data area on disk at which the transfer is to start. If bytes+offset is greater than the size of the extra data region the transfer is not done and error SON_NO_EXTRA is returned.
writeIt	TRUE (non zero) to write data to the file, FALSE to read data into the buffer.
Returns	Zero if the transfer was completed without error, or a negative error code.
Errors	SON_NO_FILE, SON_BAD_READ, SON_BAD_WRITE, SON_NO_EXTRA

SONGetExtraDataSize

Use this function to find the amount of extra data available from a SON file. The size of the extra data area is set when the file is created by the SONOpenNewFile call, and cannot be changed afterwards.

```
long SONGetExtraDataSize(short fh)
```

fh	The file handle.
Returns	The number of bytes of extra data available, or zero.
Errors	None

SONGetFileComment

This function is used to get the strings making up the SON file comment. The comment consists of five strings, each up to SON_COMMENTSZ (currently 79) characters long.

```
void SONGetFileComment(short fh, WORD which, TpStr pcFCom, short sMax);
```

fh	The file handle.
which	The string number to return, from 0 to 4.
pcFCom	Points to a character array to hold the returned string.
sMax	The maximum number of characters that can be stored in the array pointed to by comment. The character array should be 1 larger than sMax, to leave space for the terminating zero byte.

SONGetFreeChan

This function will search the channel tables of a file for an unused data channel.

```
short SONGetFreeChan(short fh);
```

fh	The file handle.
Returns	The first free channel number found, or a negative error code.
Errors	SON_NO_CHANNEL

SONGetIdealLimits

This function returns ideal values for a channel; floats holding the theoretical or expected number of samples per second, the expected minimum value and the expected maximum

value. The minimum and maximum values are derived from the ADC data scaling for `Adc` and `AdcMark` channels, from the expected rate for `Event`, `Marker` and `TextMark` channels and from the channel header for `RealMark` channels (set by `SONSetRealMarkChan`). This function is now replaced by `SONIdealRate` and `SONYRange` and may be removed in future releases.

```
void SONGetIdealLimits(short fh, WORD chan, TpFloat pfRate,
                       TpFloat pfMin, TpFloat pfMax);
```

`fh` The file handle.

`chan` The channel number for which we require the ideal rate.

`pfRate` Either `NULL` or points to a float to set to the ideal rate for this channel.

`pfMin` Either `NULL` or points to a float to set to the expected channel minimum value.

`pfMax` Either `NULL` or points to a float to set to the expected maximum channel value.

SONGetMarkData

This function will collect markers into a buffer from the file between two specific times. The channel must hold `Marker`, `AdcMark`, `RealMark` or `TextMark` data.

```
long SONGetMarkData(short fh, WORD chan, TpMarker pMark, long max,
              TSTime sTime, TSTime eTime, TpFilterMask pFltMask);
```

`fh` The file handle.

`chan` The data channel in the file to read the `Marker` data from. The channel must exist and hold `Marker` or extended `Marker` data.

`pMark` Points to a memory region to hold the data read from the file.

`max` The maximum number of markers to be read. Though a `WORD` value, max cannot be greater than 32767. Each `Marker` requires 8 bytes of memory, so the buffer region must be of size at least 8*max.

`sTime` The start time for the search in clock ticks.

`eTime` The end time for the search in clock ticks. Note that `sTime` and `eTime` are both included in the search.

`pFltMask` Points to the structure to be used to filter markers. This structure will be used to filter all types of `Marker` channel. If this parameter is `NULL`, no filter will be used and all markers accepted.

Returns The number of markers transferred to the buffer, or a negative error code.

Errors `SON_BAD_READ`, `SON_NO_CHANNEL`

SONGetRealData

This function reads contiguous waveform data from a `RealWave`, `Adc` or `AdcMark` channel in the current file between two set times. The data is transferred to a user-defined buffer in `float` format. If an `AdcMark` channel is used, the data returned can be from more than one `AdcMark` structure if the `AdcMark` records overlap so that a longer contiguous record is available. Data read from `Adc` or `AdcMark` channels is converted to floating point using the `scale` and `offset` defined for the channel.

```
long SONGetRealData(short fh, WORD chan, TpFloat pFloat, long max,
              TSTime sTime, TSTime eTime, TpSTime pbTime,
              TpFilterMask pFltMask);
```

`fh` The file handle.

`chan` The data channel in the file to search for the data.

`pFloat` Points to the area of memory in which any data found is to be returned.

`max` The maximum number of data points which may be returned into the area pointed at by `pFloat`.

sTime The start time of the search in clock ticks. No data point will be returned which was timed before this time.

eTime The end time of the search. No data point will be returned which was timed after this time.

pbTime Points to a TSTime variable to hold the time of the first data point located.

pFltMask Only used when reading from an AdcMark channel. Points to the TFilterMask structure to be used to filter the data. If this parameter is NULL, no filter is used and all AdcMark data is accepted.

Returns The number of data points returned or a negative error code.

Note that this function only returns contiguous data. If the end of the data area does not reach the end of the time period you should request more data from beyond the last data point returned to you:

```
newSTime = bTime + (points * SONChanDivide(fh, chan);
```

where points is the number of data points that were returned.

Errors SON_BAD_READ, SON_NO_CHANNEL

SONGetTimePerADC

This returns the number of clock ticks per ADC conversion set by SONSetFileClock.

```
WORD SONGetTimePerADC(short fh);
```

fh The file handle.

Returns The number of clock ticks per ADC conversion.

Errors This function returns no errors.

SONGetusPerTime

This function returns the file clock tick interval in base time units as defined by SONTimeBase(). The base time units are normally microseconds. The clock tick period in seconds is: SONGetusPerTime(fh)*SONTimeBase(fh, 0.0).

```
WORD SONGetusPerTime(short fh);
```

fh The file handle.

Returns The number of base time units in the clock tick interval.

Errors This function returns no errors.

SONGetVersion

This returns the file system version of the file as stored on disk. Files are saved as the oldest format compatible with the data in them so that old software can read the files.

```
int SONGetVersion(short fh);
```

fh The file handle.

Returns The function returns the file version of -1 if no file is open.

SONIdealRate

This function gets and/or sets the ideal waveform sample rate for Adc and RealWave channels and the expected average event rate for all other channel.

```
SONAPI(float) SONIdealRate(short fh, WORD chan, float fIR);
```

fh The file handle.

chan The data channel in the file to search for the data.

`pIR` If this value is >= 0.0 the functions sets a new value. If it is <0, the current value is not changed.

Returns The value at the time of the call or 0.0 if the channel does not exist.

SONInitFiles

This function must be used before any other function in the library to initialise the file descriptor tables. It is almost certain that a program which uses the SON library and which fails to call this function will crash in an unpredictable fashion. Note however that Windows programs do not need to call `SONInitFiles` as library initialisation is carried out automatically when the `SON DLL` is loaded.

```
void SONInitFiles( void );
```

Errors None.

SONItemSize

This function is used to get the size, in bytes, of a single item from the specified channel. This function is most useful when dealing with the various types of extended marker data, whose size can vary from file to file.

```
WORD SONItemSize(short fh, WORD chan);
```

`fh` The file handle.

`chan` The channel to get the information for.

Returns The size of a data item, in bytes. If you request the size of a channel that is off, the result is 1.

Errors This function returns no errors.

SONIsBigFile

This tests if an open file is in Big file mode or not.

```
int SONIsBigFile(short fh);
```

`fh` The file handle.

Returns 1 if a big file, 0 if not or a negative error code.

Errors `SON_NO_FILE`

SONIsSaving

If you are turning writing of data on and off with `SONKIllRange()` or `SONSave()` you can use this routine to tell you if data is currently being saved.

```
short SONIsSaving(short fh, int nChan);
```

`fh` The file handle.

`nChan` A channel number.

Returns 0 if the channel does not exist or is not saving, non-zero if it is saving.

Errors None

SONKillRange

This function sets the write state for a given channel so that all data within a specified time range is discarded. When a new data file is being written, it is expected that all data is written into the `SON` file using the `SONWriteXxxxBlock` functions regardless of whether the data should be written to disk. This allows data that will be discarded

eventually to appear to be part of the disk file (usually for online display) and also allows the decision on keeping or discarding the data to be altered retrospectively.

```
short SONKillRange(short fh, int nChan, TSTime sTime, TSTime eTime);
```

fh	The file handle.
nChan	A channel number or -1 to specify all channels.
sTime	The start time, in clock ticks, of the range within which data is discarded. This time can be before or after the current maximum time for the channel or file. If sTime is long before the current time the oldest data may not be discarded if it has already been saved to disk.
eTime	The end of the time range within which data is discarded.
Returns	Zero if all OK, 1 if changes were lost, or a negative error code.
Errors	SON_NO_FILE, SON_READ_ONLY

SONLastPointsTime

Find the time on any type of channel at which a read of lPoints points from sTime to eTime terminates. If it is not possible to read that many points, but fewer are OK, then we return the time for whatever points are available. Only contiguous ADC or RealWave data allowed, as per usual.

```
TSTime SONLastPointsTime(short fh, WORD wChan, TSTime sTime, TSTime eTime,
                         long lPoints, BOOL bAdc, TpFilterMask pFltMask);
```

fh	The file handle.
wChan	The channel to search
sTime	The time to start the search at.
eTime	Search up to and including this time (NOTE: eTime < sTime!)
lPoints	The maximum number of points required.
bAdc	Set TRUE to handle AdcMark data as Adc. This is only valid if the channel interleave is 1. Otherwise, this is a bad parameter error.
pFltMask	Pointer to marker filter, or NULL to accept all marker data.
Returns	the time for the last point or -1 if none found or an error.
Errors	SON_NO_FILE, SON_BAD_PARAM

SONLastTime

This function returns the time and details of the last data point on a given channel before a specified time.

```
TSTime SONLastTime(short fh, WORD chan, TSTime time, TpVoid pvVal,
            TpMarkBytes pMB, TpBOOL pbMark, TpFilterMask pFltMask);
```

fh	The file handle.
chan	The channel number.
time	The time at which to start the search backwards for data.
pvVal	This is used to return information about the last data point found. It can be NULL or a pointer to a short or a pointer to a float depending on the channel type. For an EventBoth channel, the short will be updated with the levLow value as 0 or 1. For an Adc channel, the short is set to the waveform value. For a RealWave channel you must provide a float value to be updated with the waveform value. The pointer is not used for other channel types.
pMB	The marker byte values, an array of four bytes. This is returned holding the marker byte values for any type of marker channel, for other types of channel it is unused.

pbMark Points to a BOOLEAN variable which will be updated with a TRUE value if the channel is a marker channel, otherwise FALSE.

pFltMask Points to the structure to be used to filter markers. This structure will be used to filter all types of marker channel. If this parameter is NULL, no filter will be used and all markers accepted.

Returns The time of the previous data item, or a negative error code.

Errors SON_BAD_READ, SON_NO_CHANNEL

SONLatestTime

This function sets the latest known valid time for a channel. This is the time before which there are no missing data values and no possible upcoming save/discard changes. This function allows the SON library to automatically tidy up the write data buffers as time passes; flushing data to disk or discarding it as appropriate.

```
short SONLatestTime(short fh, int nChan, TSTime sTime);
```

fh The file handle.

nChan A channel number or -1 to specify all channels.

sTime The latest valid time, in clock ticks.

Returns Zero if all OK, or a negative error code.

Errors SON_NO_FILE, SON_READ_ONLY

SONMarkerItem

This function helps languages like Visual Basic that have trouble with extended marker types because of their varying sizes. Either read extended marker data into a buffer, then use this routine to extract each item, or use this routine to build the buffer of extended marker data for writing. This function is also exported as SONTextMarkItem by the type library so that String conversions get done. See the Visual Basic section for an example.

```
SONAPI(int) SONMarkerItem(short fh, WORD wChan, TpMarker pBuff, int n,
                          TpMarker pM, TpVoid pvData, BOOLEAN bSet);
```

fh The file handle.

nChan A channel number of a Marker, RealMark, TextMark or AdcMark channel. The file handle and channel information are used to calculate the size of each data item and the number of bytes to copy.

pBuff Points at the buffer to use for the transfer. This is exported by the type library as void* so that Visual Basic sees this As Any.

n The index number (0 based) of the extended marker type in the buffer.

pM The marker part of the data to copy to or from the buffer.

pvData Points at the extra data attached to the marker to copy to or from the buffer. The number of bytes copied depends on the data type. If this is a TextMark channel, only bytes up to and including the zero terminating character are copied.

bSet 0 (FALSE) means copy from the buffer, 1 or non-zero (TRUE) means copy to the buffer.

Returns Zero if all OK, or a negative error code.

Errors SON_NO_FILE, SON_NO_CHANNEL, SON_BAD_PARAM

SONMaxChans

This returns the number of channels that the file supports. It is a fatal error to use this function if the file handle does not refer to an open file. Version 8 and later files can

support from 32 up to 451 channels. The version 6 upper limit was 255 channels. Previous versions always had 32 channels.

```
int SONMaxChans(short fh)
```

`fh` The file handle.

Returns The number of channels that this file supports in the range 32-451.

SONMaxItems

This returns the maximum number of data items that can be stored per disk buffer in this channel. This is only meaningful after you have created the channel! This information was used to optimise writing to disk by always writing blocks of this size. As the library now buffers up data, there is no need for this optimisation.

```
int SONMaxItems(short fh, WORD chan)
```

`fh` The file handle.

`chan` The channel to report on.

Returns The maximum number of data items that can be stored in each buffer of data for this channel.

SONMaxTime

This function is used to obtain the maximum time for data in a file.

```
TSTime SONMaxTime(short fh);
```

`fh` The file handle.

Returns The maximum time, in clock ticks.

Errors No errors are returned (no checks in non-debug build).

SONOpenNewFile

This creates a new file and returns a file handle. This is for backwards compatibility; new software should use `SONCreateFile()`. The function is implemented by calling `SONCreateFile` with 32 channels. The Macintosh version requires more arguments.

Windows/MS_DOS

```
short SONOpenNewFile(TpStr pcName, short fMode, WORD extra);
```

`pcName` The name of the file to be created, optionally including a drive and path. If a file of this name already exists it is truncated to zero length.

`fMode` A flag indicating the writing method for the file that you should set to `NormalWrite` (0) and which is ignored from version 6 onwards of the library.

In previous versions of the library if `fMode` was set to `NormalWrite`, the forward pointer of the previous block on a channel is updated for each block written. If you selected `FastWrite`, the forward pointer of the previous block was not updated, which saved disk head movement and speeded up the process at the cost of the `SONCloseFile` function running much more slowly as it backtracked to fill in the missing links.

However, from version 6 of the SON library, data is automatically buffered and is not written until the position of the next block is known, so there is no need for `FastWrite` mode. `FastWrite` mode was used by old applications that wrote data to the file themselves, bypassing the library. The current library will identify an old file with missing forward links and will automatically fill them in when the file is opened.

`extra` The number of bytes to reserve as extra space in the file header. This space can be accessed through the `SONGetExtraData` function.

Returns The function returns a positive file handle to be used in all future accesses if the new file was opened without error, or a negative error code.

Errors `SON_NO_FILE`, `SON_OUT_OF_MEMORY`, `SON_NO_HANDLES`, `SON_NO_ACCESS`

Macintosh This function creates a new SON format file on the Macintosh.

```
short SONOpenNewFile(ConstStr255Param name, short fMode, WORD extra,
              short vRefNum, long dirID, SignedByte perm, OSType creator,
              OSType fileType);
```

`name` A Pascal-style string holding the name of the file to be created.

`fMode` A flag indicating the writing method for the file, see the description above.

`extra` The number of bytes to reserve as extra space in the file header, see above.

`vRefNum` The reference number of the volume on which the file is to be created.

`dirID` The ID of the directory in which the file is to be created.

`perm` This specifies the access permissions required to the file.

`creator` An `OStype` identifier specifying the program which is creating this file.

`fileType` An `OSType` identifier that should be set to "`SON` " so that the file is identified as a SON type file.

Returns The function returns a valid file handle if the SON file was opened without error, or a negative error code.

Errors `SON_NO_FILE`, `SON_OUT_OF_MEMORY`, `SON_NO_PATH`, `SON_NO_HANDLES`, `SON_NO_ACCESS`, `SON_INVALID_DRIVE`, Mac OS error codes

SONOpenOldFile

This function opens an existing SON format file for read/write operations.

```
short SONOpenOldFile(TpStr pcName, int iMode);
```

`pcName` The name of the file to be opened. This can be any legal MS-DOS file name optionally including a drive and path. The file must be a SON type file.

`iMode` The file open mode. Set to 0 for read/write access, 1 for read only and 2 for attempt read/write, but accept read only if read/write open fails.

Returns The function returns a valid file handle if the SON file was opened without error, or a negative error code.

Errors `SON_NO_FILE`, `SON_OUT_OF_MEMORY`, `SON_NO_HANDLES`, `SON_NO_ACCESS`, `SON_CORRUPT_FILE`

Macintosh form This function opens an existing file in the Macintosh version of the SON library.

```
short SONOpenOldFile(ConstStr255Param name, short vRefNum, long dirID,
              SignedByte perm);
```

`name` A Pascal-style string holding the name of the file to be opened. This can be any legal Mac file name, the file must be a SON type file to open successfully.

`vRefNum` The reference number of the volume holding the file.

`dirID` The ID of the directory holding the file to be opened.

`perm` This specifies the access permissions required.

Returns The function returns a valid file handle if the SON file was opened without error, or a negative error code.

Errors `SON_NO_FILE`, `SON_OUT_OF_MEMORY`, `SOSON_NO_PATH`, `SON_NO_HANDLES`, `SON_NO_ACCESS`, `SON_INVALID_DRIVE`, Mac OS error codes

SONPhyChan

Returns the physical channel number associated with this channel. The physical channel is set when the channel is created. It has no function in the SON system except to document the source of the data.

```
int SONPhyChan(short fh, WORD wChan);
```

`fh` The file handle.

Returns A positive physical channel number or a negative number if there is no physical channel or if the channel is off.

SONPhySz

Return the data buffer size for this channel. Most programs do not need this information.

```
int SONPhySz(short fh, WORD wChan);
```

`fh` The file handle.

Returns The size of the channel buffer or 0 if the channel is off.

SONSave

This function sets the write state for a given channel so that all data at or after a specified time is kept or discarded. When a new data file is being written, it is expected that all data is written into the `SON` file using the `SONWriteXxxxBlock` functions regardless of whether the data should be written to disk. This allows data that will be eventually discarded to appear to be part of the disk file (for display purposes) and also allows the decision on keeping or discarding the data to be altered retroactively.

```
short SONSave(short fh, int nChan, TSTime sTime, BOOLEAN bKeep);
```

`fh` The file handle.

`nChan` A channel number or -1 to specify all channels.

`sTime` The time, in clock ticks, at which the save/discard change takes effect. This time can be before or after the current maximum time for the channel or file. If `sTime` is long before the current time the save/discard decision may not be able to affect the oldest data as it may no longer be in the write buffers.

`bKeep` A `BOOLEAN` value defining if the data at or after `sTime` is to be kept or discarded. Set `TRUE` to keep the data, `FALSE` to discard. It is not an error to discard data that is already discarded or vice-versa.

Returns Zero if all OK, 1 if changes were lost, or a negative error code.

Errors `SON_NO_FILE, SON_READ_ONLY`

SONSaveRange

This function sets the write state for a given channel so that all data within a specified time range is kept. When a new data file is being written, it is expected that all data is written into the `SON` file using the `SONWriteXxxxBlock` functions regardless of whether the data should be written to disk. This allows data that will be discarded eventually to appear to be part of the disk file (for display purposes mostly) and also allows the decision on keeping or discarding the data to be altered retroactively.

```
short SONSaveRange(short fh, int nChan, TSTime sTime, TSTime eTime);
```

`fh` The file handle.

`nChan` A channel number or -1 to specify all channels.

`sTime` The start time, in clock ticks, of the range within which data is kept. This time can be before or after the current maximum time for the channel or file. If `sTime` is long before the current time the oldest data may not be saved if it has already been discarded from the write buffers.

`eTime` The end of the time range within which data is kept.

Returns Zero if all OK, 1 if changes were lost, or a negative error code.

Errors `SON_NO_FILE, SON_READ_ONLY`

SONSetADCChan

This function is used to define a new `Adc` data channel in a file opened using `SONOpenNewFile`. It sets all of the constant channel data. This routine is here for backwards compatibility; new programs should use `SONSetWaveChan()`.

```
short SONSetADCChan(short fh, WORD chan, short sPhyCh, short dvd,
              WORD wBufSz, TpCStr szCom, TpCStr szTitle, float fRate,
              float scl, float offs, TpCStr szUnt);
```

`fh` The file handle.

`chan` The channel number in the file to use. This channel must not already be in use.

`sPhyCh` The physical channel from which the data was recorded. No check is made on this, it provides a record of the signal source. Use –1 for no physical channel.

`dvd` The divide down from the time per ADC rate for this channel. The sampling interval is `dvd*timePerADC` clock ticks.

`wBufSz` The physical size in bytes of the disk buffer to be used for writing data blocks on this channel. For efficiency reasons this is rounded up to a multiple of 512 bytes by the library. You can write smaller blocks than this, but at the cost of wasted disc space if the blocks are not contiguous (i.e. have gaps between them). The buffer size should not be larger than 32768 bytes.

`szCom` The channel comment as a text string, up to `SON_CHANCOMSZ` (71) characters.

`szTitle` The channel title as a text string of up to `SON_TITLESZ` (9) characters.

`fRate` A float containing the sampling rate requested for this channel. For files created by Spike2 the actual sampling rate used will be the best available approximation of this rate, other programs may always achieve the ideal rate.

`scl,offs` The scale and offset values to convert the short data to real user units by the equation `real=(short*scl*(10/65536))+offs`. See the section of this manual on simple use of the SON library for a full description.

`szUnt` The channel units as a text string of up to `SON_UNITSZ` (5) characters.

Returns Zero or a negative error code.

Errors `SON_NO_CHANNEL`, `SON_CHANNEL_USED`

SONSetAdcMarkChan

This function is used to create an `AdcMark` channel. This is here for backwards compatibility, new software should use `SONSetWaveMarkChan()`.

```
short SONSetADCMarkChan(short fh, WORD chan, short sPhyCh, short dvd,
              WORD wBufSz, TpCStr szCom, TpCStr szTitle, float fRate,
              float scl, float offs, TpCStr szUnt,
              WORD points, short preTrig);
```

`fh` The file handle.

`chan` The channel number in the file. This channel must not already be in use.

`sPhyCh` The physical channel from which the data was recorded. No check is made on this, it provides a record of the signal source. Use –1 for no physical channel.

`dvd` The divide down from the time per ADC rate for this channel. The sampling interval is `dvd *timePerADC` clock ticks.

wBufSz	The physical size in bytes of the disk buffer to be used for writing data blocks on this channel. For efficiency reasons this is rounded up to a multiple of 512 bytes by the library. You can write smaller blocks than this, but at the cost of wasted disc space if the blocks are not contiguous (i.e. have gaps between them). The buffer size should not be larger than 32768 bytes.
szCom	The channel comment as a text string, up to SON_CHANCOMSZ (71) characters.
szTitle	The channel title as a text string of up to SON_TITLESZ (9) characters.
fRate	A float containing the sampling rate requested for this channel. For files created by Spike2 the actual sampling rate used will be the best available approximation of this rate, other programs may always achieve the ideal rate.
scl,offs	The scale and offset values to convert the short data to real user units by the equation real=(short*scl*(10/65536))+offs. See the section of this manual on simple use of the SON library for a full description.
szUnt	The channel units as a text string of up to SON_UNITSZ (5) characters.
points	The number of waveform points in each marker.
preTrig	The number of waveform points sampled before the occurrence of the trigger causing the waveform capture.
Returns	Zero or a negative error code.
Errors	SON_NO_CHANNEL, SON_CHANNEL_USED

SONSetADCOffset

This function is used to change the offset scaling factor for an Adc or AdcMark channel. This is one of the two factors used to convert the underlying 16-bit integer values stored as the channel data into calibrated values

```
void SONSetADCScale(short fh, WORD chan, float offset);
```

fh	The file handle.
chan	The channel number.
offset	The new channel offset value.

SONSetADCScale

This function is used to change the scaling factor for an Adc or AdcMark channel. This is one of the two factors used to convert the underlying 16-bit integer values stored as the channel data into calibrated values

```
void SONSetADCScale(short fh, WORD chan, float scale);
```

fh	The file handle.
chan	The channel number.
scale	The new channel scale factor.

SONSetADCUnits

This function is used to change the units for an Adc, AdcMark or RealMark channel.

```
void SONSetADCUnits(short fh, WORD chan, TpCStr szUnt);
```

fh	The file handle.
chan	The channel number.
szUnt	The new units string, not more than SON_UNITSZ (5) characters long.

SONSetBuffering

Call this function after defining all the channels for a new file. It is used to specify how much memory is to be used to buffer writes to the data channels. Normally, SON provides one data block of buffering per channel; this ensures that only full blocks are written and optimises disk space. More data buffering allows faster retrieval of new data and allows retroactive decisions on what data is to be saved or discarded. For example, if the buffers for a channel hold 30 seconds worth of data, then the decision to save the data to disk or not can be changed up to 30 seconds after the data was sampled allowing peri-triggered sampling.

```
short SONSetBuffering(short fh, int nChan, long nBytes);
```

`fh`	The file handle.
nChan	The channel number or -1 to select all channels. Using -1 allows the application to specify the total amount of buffer memory that should be used, while allowing SON to apportion this memory between channels according to the expected data rates.
nBytes	The amount of buffer memory to be used, in bytes.
Returns	Zero if no errors or a negative error code.
Errors	`SON_NO_FILE`

SONSetBuffSpace

Call this function after defining all the channels for a new file. It allocates the data transfer buffer space used to build data blocks for transfer to disc. All the channels used are scanned, and a space is allocated equal to the biggest buffer size used by any channel. This space is used by all channels for all calls that transfer blocks of data to disk. `SONOpenOldFile` calls this for you, so there is no need to use it after opening an old file.

```
short SONSetBuffSpace(short fh);
```

`fh`	The file handle.
Returns	Zero if no errors or a negative error code.
Errors	`SON_NO_FILE`, `SON_OUT_OF_MEMORY`

SONSetChanComment

This function is used to set the comment string stored with the channel information. This comment is the same as the comment set by the `SONSetXXXChan` functions and is provided to allow the comment to be changed after the channel has been defined.

```
void SONSetChanComment(short fh, WORD chan, TpCStr szCom);
```

`fh`	The file handle.
`chan`	The channel number.
`szCom`	Points to a string containing the new comment for the channel, a maximum of `SON_CHANCOMSZ` characters long.

SONSetChanTitle

This function is used to change the channel title for a given channel. The title consists of a string up to `SON_TITLESZ` (9) characters long.

```
void SONSetChanTitle(short fh, WORD chan, TpCStr szTitle);
```

`fh`	The file handle.
`chan`	The channel number for which the title is required.
`szTitle`	The channel title as a text string of up to `SON_TITLESZ` (9) characters.

SONSetEventChan

This function sets the fixed channel information for a new Event channel. SONSetEventChan is used to set up all types of Event channel as well as simple marker channels.

```
short SONSetEventChan(short fh, WORD chan, short sPhyCh, WORD wBufSz,
                      TpCStr szCom, TpCStr szTitle, float fRate, TDataKind
evtKind);
```

fh	The file handle.
chan	The data channel in the file to use. This channel must not already be in use.
sPhyCh	The physical channel from which the data was recorded. No check is made on this, it provides a record of the signal source. Use –1 for no physical channel.
wBufSz	The physical size in bytes of the disk buffer to be used for writing data blocks on this channel. For efficiency reasons this is rounded up to a multiple of 512 bytes by the library. You can write smaller blocks than this, but at the cost of wasted disc space if the blocks are not contiguous (i.e. have gaps between them). The buffer size should not be larger than 32768 bytes.
szCom	The channel comment as a text string, up to SON_CHANCOMSZ (71) characters.
szTitle	The channel title as a text string of up to SON_TITLESZ (9) characters.
fRate	A float containing the ideal or expected rate of events for this channel. The actual rate found will not necessarily be close to this.
evtKind	The type of event which will be stored on this channel. If the data type is EventBoth it is your responsibility to set the initLow field of the channel descriptor of this channel to the initial state of the channel.
Returns	Zero if no errors or a negative error code.
Errors	SON_NO_CHANNEL, SON_CHANNEL_USED

SONSetFileClock

This function is used to set the number of base time units in a clock tick and the number of clock ticks per ADC conversion for this file.

```
void SONSetFileClock(short fh, WORD usPerT, WORD tPerADC);
```

fh	The file handle.
usPerT	Sets the number of base time units in the new file's clock tick interval.
tPerADC	The clock ticks per ADC conversion. For compatibility with SON library versions 1 to 5, the time interval between Adc and AdcMark channel waveform points must be a multiple of tPerAdc. From version 6 onward, this value is for information only; set it to 1 if you don't need to use it.

SONSetFileComment

This function is used to store data in the strings making up the SON file comment. The comment consists of five strings, each up to SON_COMMENTSZ (currently 79) characters. Strings are terminated by a 0, so the maximum length including the zero is 80 characters.

```
void SONSetFileComment(short fh, WORD which, TpCStr szFCom);
```

fh	The file handle.
which	The string number to set, from 0 to 4.
szFCom	This points to a character array holding the string, if it contains more than SON_COMMENTSZ characters it will be truncated.

SONSetInitLow

This function sets the initial state for an `EventBoth` channe. `SONSetInitLow` cannot be used after the first block of data has been written to the file, if it is called after data has been written it will do nothing.

```
void SONSetInitLow(short fh, WORD chan, BOOLEAN bLow);
```

`fh` The file handle.

`chan` An `EventBoth` data channel in the file.

`bLow` This defines the direction of the first transition written to the file. If `TRUE`, the first transition is high to low, if `FALSE` it is low to high. The filing system keeps track of the high/low state for the rest of the file by counting events. If events are deleted, they should be deleted in pairs.

Returns No return value.

Errors This function returns no errors.

SONSetMarker

This function changes the data and, within limits, the time associated with a `Marker`. Because markers must be stored in time order, the time for a given `Marker` cannot be changed to overlap the time for the previousor next `Marker` in the file.

```
short SONSetMarker(short fh, WORD chan, TSTime time, TpMarker pMark,
                   WORD size);
```

`fh` The file handle.

`chan` The channel number for the marker.

`time` The time of the marker to be changed.

`pMark` Points to the marker data that will replace the existing data in the file.

`size` The number of bytes to copy from `pMark` into the file marker. This value must be at least 4 (`pMark` must point at a valid time at least) and not more than the size of the marker item.

Returns 1 if the marker data has been replaced, zero if the replacement was not carried out (a marker was not found at time or unable to change marker time that much), or a negative error code.

Errors `SON_BAD_READ`, `SON_BAD_WRITE`, `SON_NO_CHANNEL`, `SON_READ_ONLY`

SONSetRealChan

This function creates a `RealWave` data channel. It sets all of the constant channel data.

```
int SONSetRealChan(short fh, WORD chan, short sPhyCh, TSTime dvd,
                   WORD wBufSz, TpCStr szCom, TpCStr szTitle,
                   float scl, float offs, TpCStr szUnt);
```

`fh` The file handle.

`chan` The channel number in the file. This channel must not already be in use.

`sPhyCh` The physical channel from which the data was recorded. No check is made on this, it provides a record of the signal source. Use –1 for no physical channel.

`dvd` The interval in clock ticks between waveform samples for this channel.

`wBufSz` The physical size in bytes of the disk buffer to be used for writing data blocks on this channel. For efficiency reasons this is rounded up to a multiple of 512 bytes by the library. You can write smaller blocks than this, but at the cost of wasted disc space if the blocks are not contiguous (i.e. have gaps between them). The buffer size should not be larger than 32768 bytes.

`szCom` The channel comment as a text string, up to `SON_CHANCOMSZ` (71) characters.

`szTitle` The channel title as a text string of up to `SON_TITLESZ` (9) characters.

scl,offs The scale and offset values to convert between real data stored in the channel and integers by the equation `int = (real-offs)*6553.6/scl`. See the section of this manual on simple use of the SON library for a full description.

szUnt The channel units as a text string of up to `SON_UNITSZ` (5) characters.

Returns Zero or a negative error code.

Errors `SON_NO_CHANNEL`, `SON_CHANNEL_USED`

SONSetRealMarkChan

This function creates a `RealMark` channel.

```
short SONSetRealMarkChan(short fh, WORD chan, short sPhyCh, WORD wBufSz,
              TpCStr szCom, TpCStr szTitle, float fRate,
              float min, float max, TpCStr szUnt, WORD points);
```

fh The file handle.

chan The channel number in the file. This channel must not already be in use.

sPhyCh The physical channel from which the data was recorded. No check is made on this, it provides a record of the signal source. Use –1 for no physical channel.

wBufSz The physical size in bytes of the disk buffer to be used for writing data blocks on this channel. For efficiency reasons this is rounded up to a multiple of 512 bytes by the library. You can write smaller blocks than this, but at the cost of wasted disc space if the blocks are not contiguous (i.e. have gaps between them). The buffer size should not be larger than 32768 bytes.

szCom The channel comment as a text string, up to `SON_CHANCOMSZ` (71) characters.

szTitle The channel title as a text string of up to `SON_TITLESZ` (9) characters.

fRate The expected mean data rate for this channel. This is used by `SONSetBuffering()` to divide the available buffering space between the channels in proportion to the bytes per second of each channel.

min, max The expected minimum and maximum values for this channel. These values will be used to provide default display scaling.

szUnt The channel units as a text string of up to `SON_UNITSZ` (5) characters.

points The number of real numbers in each marker.

Returns Zero or a negative error code.

Errors `SON_NO_CHANNEL`, `SON_CHANNEL_USED`

SONSetTextMarkChan

This function is used to create a `TextMark` channel.

```
short SONSetTextMarkChan(short fh, WORD chan, short sPhyCh, WORD wBufSz,
             TpCStr szCom, TpCStr szTitle, float fRate,
             TpCStr szUnt, WORD points);
```

fh The file handle.

chan The channel number in the file. This channel must not already be in use.

sPhyCh The physical channel from which the data was recorded. No check is made on this, it provides a record of the signal source. Use –1 for no physical channel.

wBufSz The physical size in bytes of the disk buffer to be used for writing data blocks on this channel. For efficiency reasons this is rounded up to a multiple of 512 bytes by the library. You can write smaller blocks than this, but at the cost of wasted disc space if the blocks are not contiguous (i.e. have gaps between them). The buffer size should not be larger than 32768 bytes.

szCom The channel comment as a text string, up to `SON_CHANCOMSZ` (71) characters.

szTitle The channel title as a text string of up to `SON_TITLESZ` (9) characters.

`fRate` The expected mean data rate for this channel. This is used by `SONSetBuffering()` to divide the available buffering space between the channels in proportion to the bytes per second of each channel.

`szUnt` The channel units as a text string of up to `SON_UNITSZ` (5) characters.

`points` The maximum number of characters including the null terminator to be stored with a marker. The disk space used per item is rounded up to a multiple of 4 bytes to avoid alignment problems.

Returns Zero or a negative error code.

Errors `SON_NO_CHANNEL`, `SON_CHANNEL_USED`

SONSetWaveChan

This function defines a new `Adc` data channel. It sets all the constant channel data.

```
short SONSetWaveChan(short fh, WORD chan, short sPhyCh, TSTime dvd,
                WORD wBufSz, TpCStr szCom, TpCStr szTitle,
                float scl, float offs, TpCStr szUnt);
```

`fh` The file handle.

`chan` The channel number in the file to use. This channel must not already be in use.

`sPhyCh` The physical channel from which the data was recorded. No check is made on this, it provides a record of the signal source. Use –1 for no physical channel.

`dvd` The number of clock ticks between waveform samples for this channel.

`wBufSz` The physical size in bytes of the disk buffer to be used for writing data blocks on this channel. For efficiency reasons this is rounded up to a multiple of 512 bytes by the library. You can write smaller blocks than this, but at the cost of wasted disc space if the blocks are not contiguous (i.e. have gaps between them). The buffer size should not be larger than 32768 bytes.

`szCom` The channel comment as a text string, up to `SON_CHANCOMSZ` (71) characters.

`szTitle` The channel title as a text string of up to `SON_TITLESZ` (9) characters.

`scl,offs` The scale and offset values to convert the short data to real user units by the equation `real=(short*scl*(10/65536))+offs`. See the section of this manual on simple use of the SON library for a full description.

`szUnt` The channel units as a text string of up to `SON_UNITSZ` (5) characters.

Returns Zero or a negative error code.

Errors `SON_NO_CHANNEL`, `SON_CHANNEL_USED`

SONSetWaveMarkChan

This function creates an `AdcMark` channel. If `nTrace * points` is odd, the disk space used by each item is rounded up to a multiple of 4 bytes to avoid alignment problems.

```
short SONSetWaveMarkChan(short fh, WORD chan, short sPhyCh, TSTime dvd,
             WORD wBufSz, TpCStr szCom, TpCStr szTitle, float fRate,
             float scl, float offs, TpCStr szUnt,
             WORD points, short preTrig, int nTrace);
```

`fh` The file handle.

`chan` The channel number in the file. This channel must not already be in use.

`sPhyCh` The physical channel from which the data was recorded. No check is made on this, it provides a record of the signal source. Use –1 for no physical channel.

`dvd` The clock ticks between waveform samples for this channel.

`wBufSz` The physical size in bytes of the disk buffer to be used for writing data blocks on this channel. For efficiency reasons this is rounded up to a multiple of 512

bytes by the library. You can write smaller blocks than this, but at the cost of wasted disc space if the blocks are not contiguous (i.e. have gaps between them). The buffer size should not be larger than 32768 bytes.

`szCom` The channel comment as a text string, up to `SON_CHANCOMSZ` (71) characters.

`szTitle` The channel title as a text string of up to `SON_TITLESZ` (9) characters.

`fRate` A float containing the sampling rate requested for this channel. For files created by Spike2 the actual sampling rate used will be the best available approximation of this rate, other programs may always achieve the ideal rate.

`scl,offs` The scale and offset values to convert the short data to real user units by the equation `real=(short*scl*(10/65536))+offs`. See the section of this manual on simple use of the SON library for a full description.

`szUnt` The channel units as a text string of up to `SON_UNITSZ` (5) characters.

`points` The number of waveform points in each marker.

`preTrig` The number of waveform points sampled before the occurrence of the trigger causing the waveform capture.

`nTrace` The number of interleaved traces in the channel in the range 1 to 4.

Returns Zero or a negative error code.

Errors `SON_NO_CHANNEL`, `SON_CHANNEL_USED`, `SON_BAD_PARAM`

SONTimeBase

This function gets and/or sets the base time units for the file. All files have base time units of microseconds (`1.0e-6`) when they are created. Nothing in the SON library makes any use of this value; it is an overall scale factor. All timings in the file system are based on clock ticks. The clock tick period in seconds is `usPerTime` times the base time unit. The ability to change the basic time units was added at version 6.

```
double SONTimeBase(short fh, double dTB);
```

`fh` The file handle.

`dTB` A new value for the base time unit if > 0.0, ignored if <= 0.0. The standard value is 1.0e-6 seconds (1 microsecond).

Returns The current value of seconds per tick or 0.0 if the file handle does not refer to an open file.

SONTimeDate

This gets and/or sets the time and date stamp from the file header. This did not exist in the file header before version 6.

```
int SONTimeDate(short fh, TSONTimeDate* pTDGet,
                                          const TSONTimeDate* pTDSet);
```

`fh` The file handle.

`pTDGet` If not `NULL`, the time and date stamp from the file header is copied here.

`pTDSet` If not `NULL` and the date is valid or all 0, the time and date field of the file is set from here. The old value is copied first so `pTDSet` and `pTDGet` should not point at the same memory.

Returns If you attempt to set an invalid time you get a SON error code. If you read a time back that is not all zeros, the result is 1, otherwise the result is 0.

Errors `SON_BAD_PARAM`, `SON_NO_FILE`

SONUpdateStart

This function writes the latest state of the file header and the channel descriptions to disk. It is called by SONCloseFile and SONCommitFile. It should also be used after all the channels have been defined and SONSetBuffSpace has been called, but before any data is written, to make sure that the file header is written in case the system crashes during data capture. If the file header is present there is a chance that the file can be recovered by the SMRFIX utility.

```
short SONUpdateStart(short fh);
```

fh	The file handle.
Returns	Zero if no error, or a negative error code.
Errors	None at present.

SONWriteADCBlock

SONWriteADCBlock is used to write Adc data points to a file. The channel information is updated on disc when the file is closed or SONUpdateStart is called.

```
TSTime SONWriteAdcBlock(short fh, WORD chan, TpAdc psBuf, long count,
              TSTime sTime);
```

fh	The file handle.
chan	The data channel in the file which the data is to be written to. This channel must have been set up as an Adc channel!
psBuf	Points to the data buffer to be written to the file.
count	The number of Adc values in the buffer to be written. This can be any number, but the most efficient number to write is a multiple of (wBufSz-20)/2 (wBufSz is as defined in the SONSetADCChan function) as this optimises the use of the disk space.
sTime	The time of the first value in the buffer in clock ticks. Every data block written holds the times of the first and last data items to help the data retrieval system. It is important that this time is exact, as it is this which determines if consecutive Adc data blocks are contiguous.
Returns	The time, in clock ticks, of the next block start on this channel (assuming that the data will be contiguous) as a positive number. If the command fails, it returns a negative error code.
Errors	SON_NO_CHANNEL, SON_BAD_WRITE, SON_NO_FILE, SON_READ_ONLY

SONWriteEventBlock

This function is used to write a block of event times to a channel. Events must be stored in time order in the file. If you write events out of order the filing system will not be able to find data correctly. We do not check for the time order of events.

```
short SONWriteEventBlock(short fh, WORD chan, TpSTime plBuf, long count);
```

fh	The file handle.
chan	The data channel in the file. This channel must have been set up as an event channel. If you skip events on an EventBoth channel you must miss an even number to allow the system to keep track of the state of the event.
plBuf	Points to the data buffer holding the TSTime event times.
count	The number of event times in the buffer to be written. This can be any number, but the most efficient number to write is a multiple of (wBufSz-20)/4 (wBufSz is as defined in the SONSetEventChan function) as this optimises the use of the disc space.
Returns	Zero if no errors, otherwise a negative error code.
Errors	SON_NO_CHANNEL, SON_BAD_WRITE, SON_NO_FILE, SON_READ_ONLY

SONWriteExtMarkBlock

This writes a block of extended Marker (AdcMark, RealMark and TextMark) data. The data must be in time order. If you write the markers out of order the filing system will not be able to find data correctly. We do not check for the time order of events.

```
short SONWriteExtMarkBlock(short fh, WORD chan, TpMarker pM, long count);
```

fh	The file handle.
chan	The data channel in the file to write the data to. This channel must be an extended Marker channel of the appropriate type.
pM	Points to the data buffer holding the AdcMark structures.
count	The number of structures in the buffer to write. The most efficient number to write is a multiple of (wBufSz-20)/SONItemSize(). wBufSz is as defined in SONSetXXXMarkChan and SONItemSize is the result of SONItemSize.
Returns	Zero if no errors, otherwise a negative error code.
Errors	SON_NO_CHANNEL, SON_BAD_WRITE, SON_NO_FILE, SON_READ_ONLY

SONWriteMarkBlock

This function writes a block of Marker data. The markers must be in time order. If you write markers out of time sequence, the SON library will not be able to find them.

```
short SONWriteMarkBlock(short fh, WORD chan, TpMarker pM, long count);
```

fh	The file handle.
chan	The data channel in the file to hold the Marker data. This channel must have been set up to hold Marker data.
pM	Points to the memory region that holds the Marker data.
count	The number of markers to be written to the file. This can be any number, but the most compact storage is obtained when count is a multiple of (wBufSz-20)/8 where wBufSz is this channel's buffer size.
Returns	Zero if no errors, otherwise a negative error code.
Errors	SON_NO_CHANNEL, SON_BAD_WRITE, SON_NO_FILE, SON_READ_ONLY

SONWriteRealBlock

SONWriteADCBlock writes RealWave data points to the file.

```
TSTime SONWriteRealBlock(short fh, WORD chan,
                         TpFloat pfBuf, long count, TSTime sTime);
```

fh	The file handle.
chan	The data channel in the file which the data is to be written to. This channel must have been set up as a RealWave channel!
pfBuf	Points to the data buffer to be written to the file.
count	The number of float values in the buffer to be written. The library buffers written data and saves it to disk when the buffers are full. It is an error to write data at a time that is earlier than the last written data.
sTime	The time of the first value in the buffer in clock ticks. Every data block written holds the times of the first and last data items to help the data retrieval system. It is important that this time is exact, as it determines if consecutive RealWave data blocks are contiguous.
Returns	The time, in clock ticks, of the next block start on this channel (assuming contiguus data) as a positive number or a negative error code.
Errors	SON_NO_CHANNEL, SON_BAD_WRITE, SON_NO_FILE, SON_READ_ONLY

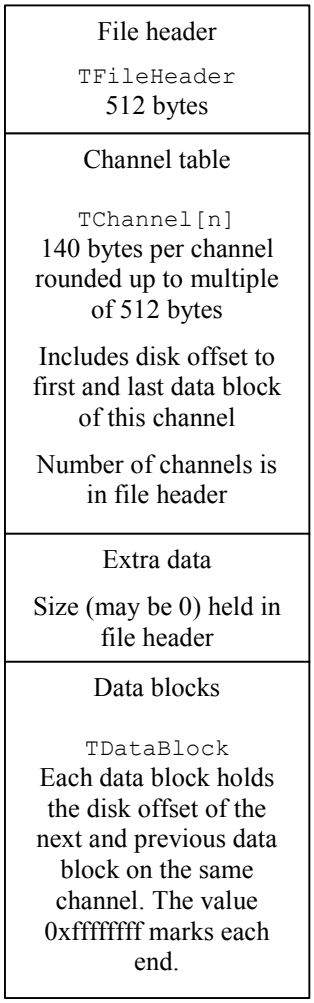
Internal library information

Internal structures and functions

In addition to the structures and functions declared in `SON.H`, there is a separate include file, `SONINTL.H`, declaring structures and functions normally only used internally. This file includes the definition of all of the header structures used in SON files and the master table containing details of all the files open, a SON file handle is an index into this table. It is not intended that any of these structures or functions should be used by programmers writing applications using SON, they are documented for completeness and for the use of systems programmers extending the SON library.

Structure on disk

All son files start with a header that is 512 bytes long. This corresponds to the `TFileHead` structure described below. The header has information to identify the file and the application that wrote it, the number of data channels, the offset to the start of the data area, the size of the extra data area and the basic time base information for the file.



The channel table follows the header. This corresponds to an array of `TChannel`, with one `TChannel` structure per channel. The size of `TChannel` is 140 bytes. The channel area is rounded up in size to a multiple of 512 bytes.

The information stored for each channel includes the size of each channel data block, the number of data blocks and the disk offset of the first and last data block for the channel. Channels can be deleted, in which case the number of deleted blocks is saved in the channel block and the disk offset to the first deleted block. If a channel is reused, the deleted blocks can either be re-used, or they can be abandoned (leaving gaps in the file). The only way to remove deleted channel data is to rewrite the file.

An optional extra data area that is reserved for application specific data follows the channel information. The SON filing system provides routines to read and write this area, but has nothing to say about the structure of the extra data area. The size of the extra data area is held in the file header.

The remainder of the file is composed of data blocks and deleted data blocks. These correspond to the `TDataBlock` structure. The size of each data block is requested by the application that writes the file. The SON library rounds up the size to the next multiple of 512 bytes. Data is always arranged on 512 byte boundaries because many physical disk systems use this as a sector size (so it is efficient), and it allows us to scan damaged files to find blocks headers knowing that all blocks start on a 512 byte boundary.

Each data block has a 20 byte header that is followed by the channel data. The header has a pointer to the previous and next data block on the same channel (or 0xffffffff to mark the end of the list), the time in clock ticks of the first and last data item in the block, the number of data items in the block, the channel number, and a flag for `EventLevel` data to indicate the level of the first data item in the block. Blocks are not changed if the channel they belong to is deleted.

SONINTL.H contents

This description of the internal data file omits structures that are not part of the visible disk format of the file. Purely internal structures are maintained here because earlier versions of the filing system made them visible, but we may change them in future releases. The only reason for using this information should be to further your knowledge of the system or to write code to read a SON file using of the SON library.

TDOF

This type is defined for use as Type Disk OFfset. It holds all references to positions in the data file. It is defined as `typdef long TDOF;` though for testing purposes in a C++ environment it can be advantageous to define it as an opaque type to detect all attempts to do maths with it. In version 9 and above files, if `pos` is of type `TDOF`, the disk offset is `pos*DISKBLOCK`. In version 8 and previous files, the disk offset is `pos`. We also define `TDOF64` internally for use as a 64-bit disk offset.

Simple constants

These constants and types are defined in `SONINTL.H` for use within the SON library.

```
#define LSTRING(size) union{unsigned char len;char string[size+1];}
#define REVISION 6
#define MAXFILES 32              /* Max no. of files (ignored for WIN32) */
#define MAXLOOK 512                    /* Lookup table entries per file */
#define MAXWBUF 64              /* Write buffers per channel in new file */
#define CHANGES 8              /* Stored changes per channel in new file */
#define DISKBLOCK 512                          /* Size of a disk block */
#define ROUND_TO_DB(num) (((num)+DISKBLOCK-1)&0xfe00)
#define LENCOPYRIGHT  10       /* Length of copyright and serial strings */
#define LENSERIALNUM   8
#define COPYRIGHT "(C) CED 87"              /* The copyright string used */
```

File header

The `TFileHead` structure is an image of the first 512 bytes of a `SON` file. It contains the SON file identification, general information about the file and channels and the file comment. From version 6, the serial number field (never used in anger) is replaced by the `creator` field, which is supported by `SONAppId()`. The `dTimeBase` and `timeDate` fields were added at version 6. `LUTable` was added at version 9. All `padx[]` bytes are 0.

```
typedef struct      /* first disk block of file */
{
    short systemID;               /* filing system revision level */
    char copyright[LENCOPYRIGHT]; /* space for "(C) CED 87" */
    TSONCreator creator;          /* optional application identifier */
    WORD usPerTime;               /* microsecs per time unit */
    WORD timePerADC;              /* time units per ADC interrupt */
    short fileState;              /* condition of the file */
    TDOF firstData;               /* offset to first data block */
    short channels;               /* maximum number of channels */
    WORD chanSize;                /* memory size to hold chans */
    WORD extraData;               /* No of bytes of extra data in file */
    WORD bufferSz;                /* Not used on disk; bufferP in bytes */
    WORD osFormat ;               /* 0x0101 for Mac, or 0x0000 for PC */
    TSTime maxFTime;              /* max time in the data file */
    double dTimeBase;             /* time scale factor, normally 1.0e-6 */
    TSONTimeDate timeDate;        /* time that corresponds to tick 0 /
    pad0[3];                      /* align next item to 4 bytes */
    TDOF LUTable;                 /* 0, or the TDOF to a saved lut */          i
    char pad[44];                 /* padding for the future */
    TFileComment fileComment;     /* what user thinks of it so far */
} TFileHead;
typedef TFileHead FAR * TpFileHead;
```

When the SON library opens a file, it loads the file header into memory, then if the `system` ID is not the latest version, it upgrades the header to the latest version. When the file closes, the library checks the file contents and sets the header to the oldest version that is compatible with the data in the file so that old applications can still read files that do not use new features.

Channel header

The `TChannel` structure is an image of the channel information stored on disk after the file header. A SON file contains an array of `TChannel` structures starting immediately after the file header. The rest of the file consists of data blocks, which are forward and backwards linked into lists, one list per channel (plus an optional second list of deleted blocks). The channel structure contains pointers to the start and end of the linked list of blocks.

```
typedef struct
{
    WORD delSize;       /* number of blocks in deleted chain, 0=none */
    TDOF nextDelBlock;  /* if deleted, first block in chain pointer */
    TDOF firstBlock;    /* points at first block in file */
    TDOF lastBlock;     /* points at last block in file */
    WORD blocks;        /* number of blocks in file holding data */
    WORD nExtra;        /* Number of extra bytes attached to marker */
    short preTrig;      /* Pre-trig points for ADC Marker data */
    short blocksMSW;    /* Hi word of block count in version 9 */
    WORD phySz;         /* physical size of block written =n*512 */
    WORD maxData;       /* maximum number of data items in block */
    TChanComm comment;  /* string commenting on this data */
    long maxChanTime;   /* last time on this channel */
    long lChanDvd;      /* waveform divide from usPerTime, 0 for others */
    short phyChan;      /* physical channel used */
    TTitle title;       /* user name for channel */
    float idealRate;    /* ideal rate:ADC, estimate:event */
    TDataKind kind;     /* data type in the channel */
    unsigned char delSizeMSB;  /* MSB of deleted channels in version 9*/

    union                       /* Section that changes with the data */
    {
        struct
        {                       /* Data for ADC and ADCMark channels */
            float scale;
            float offset;       /* to convert to units */
            TUnits units;       /* channel units */
            WORD divide;        /* V5:ADC divide, V6:AdcMark interleave */
        } adc;
        struct
        {                       /* only used by EventBoth channels */
            BOOLEAN initLow;    /* initial event state */
            BOOLEAN nextLow;    /* expected state of next write */
        } event;
        struct
        {                       /* RealMark and RealWave */
            float min;          /* expected minimum value */
            float max;          /* expected maximum value */
            TUnits units;       /* channel units */
        } real;                 /* NB this is laid out as for adc data */
    } v;

} TChannel;
typedef TChannel FAR * TpChannel;
```

When the SON library opens a file, it reads the channel information for all channels into memory. If the file is not the latest version, the library updates the channel information to the latest standard. When upgrading to V6, the `lChanDvd` field for `Adc` and `AdcMark` channels is set to `divide`* `timePerADC` and `divide` is set to 1. When a file closes, the file version is down-graded to the oldest format that would not lose information.

`blocksMSW` and `delSizeMSB` were added at version 9 to support files with huge numbers of blocks. When writing as a version 8 or previous file, if these values are non-zero, the `blocks` or `delSize` locations are set to 0xffff (i.e. as large as possible).

Data block

The TDataBlock structure defines a data block used to hold channel data. Data blocks are arranges as doubly linked lists, with each block holding a pointer to the next and previous block in the chain. Actually these are not pointers, but offsets from the start of the file. Data blocks are of variable size, the fixed-size data types are defined as arrays of length 1 so that standard array indexing mechanisms can be used to retrieve data.

```
typedef struct
{
    TDOF   predBlock;     /* predecessor block in the file */
    TDOF   succBlock;     /* following block in the file */
    TSTime startTime;     /* first time in the file */
    TSTime endTime;       /* last time in the block */
    WORD   chanNumber;    /* The channel number in the block */
    WORD   items;         /* Actual number of data items found */
    union
    {
        TAdc      int2Data [ADCdataBlkSize];    /* ADC data */
        TSTime    int4Data [timeDataBlkSize];   /* time data */
        TMarker   markData [markDataBlkSize];   /* marker data */
        TAdcMark  adcMarkData;                  /* ADC marker data */
        float     realData [realDataBlkSize];   /* RealWave data */
    } data ;
} TDataBlock;
typedef TDataBlock FAR * TpDataBlock;         /* Pointer to a data block */
#define SONDBHEADSZ 20                      /* size of TDataBlock header */
```

The chanNumber is stored in an unpleasant way as bit 8 of it is used to store the initial level in a data block for a level event channel. The channel number is stored as a 9 bit number with bits 0-7 as bits 0-7 and bit 8 stored on disk as bit 9. Also, channel n is stored on disk as n+1. There are macros to do the mapping: CHFRDISK(n) and CHTODISK (n).

File data

The TFH type is defined to be whatever type acts as a file handle so that other functions can use this type without having to care about what it is.

```
#ifdef _IS_WINDOWS_
#define TFH HANDLE
#endif
#ifdef LINUX
#define  TFH int
#endif
```

Look up table

The file header LUTable value, if non-zero and the file version is 9, points at stored look up tables on disk. The tables are always written at the end of the file as:

TLUID	\	An ID structure for the channel
TSonLUTHead	| per channel	A header to describe the table - see sonpriv.h
TLookup[nUsed]	/	The table data - see sonpriv.h
TLUID	An ID with a channel number of -1 to mark table end	
padding	Padding up to a multiple of BLOCKSIZE bytes	

```
typedef struct tagTLUTID  /* structure used to identfy a LUT on disk */
{
    unsigned long ulID;   /* set to 0xfffffffe to identify */
    int chan;             /* channel number or -1 if no more entries */
    unsigned long ulXSum; /* checksum of TSonLUTHead and TLookup */

} TLUTID;
```

The checksum is a simple sum treating the header data and the TLookup[] data as an array of long. Reading of the lookup table is abandoned if any problem is found with the table or if the table does not match the channel. The table holds disk pointers and block times arranged in ascending time order that are used to speed up disk access. If the tables are not present, the library builds them as the file is used.

SONRead

This function reads a block of data from a SON file.

```
short SONRead(short fh, TpVoid buffer, WORD bytes, TDOF offset);
```

`fh`	The file handle.
`buffer`	Points to an area of memory to hold the data read.
`bytes`	The number of bytes to read.
`offset`	The offset in the file to the first byte to be read from disk.
Returns	Zero or a negative error code.
Errors	`SON_BAD_READ`

SONRead64

This function reads a block of data from a SON file.

```
short SONRead(short fh, TpVoid buffer, DWORD bytes, TDOF64 offset);
```

`fh`	The file handle.
`buffer`	Points to an area of memory to hold the data read.
`bytes`	The number of bytes to read.
`offset`	The offset in the file to the first byte to be read from disk.
Returns	Zero or a negative error code.
Errors	`SON_BAD_READ`

SONWrite

This function writes a block of data to a SON file.

```
short SONWrite(short fh, TpVoid buffer, WORD bytes, TDOF offset);
```

`fh`	The file handle.
`buffer`	Points to an area of memory holding the data to be written.
`bytes`	The number of bytes to write.
`offset`	The offset in the file to the first byte to be written to disk.
Returns	Zero or a negative error code.
Errors	`SON_BAD_WRITE`, `SON_BAD_READ`

SONWrite64

This function writes a block of data to a SON file.

```
short SONWrite(short fh, TpVoid buffer, DWORD bytes, TDOF64 offset);
```

`fh`	The file handle.
`buffer`	Points to an area of memory holding the data to be written.
`bytes`	The number of bytes to write.
`offset`	The offset in the file to the first byte to be written to disk.
Returns	Zero or a negative error code.
Errors	`SON_BAD_WRITE`, `SON_BAD_READ`

SONGetBlock

This function should only be of interest to advanced users of the SON library. It reads a block of data into the internal buffer assigned to a file (see `SONGetBuffSpace`).

```
short SONGetBlock(short fh, long offset);
```

fh	The file handle.
offset	The file offset to the start of the block to be read. No check is made to see if this points at a data block.
Returns	Zero or a negative error code.
Errors	SON_NO_FILE, SON_BAD_READ

SONGetPred

This function returns the file offset of the previous data block on the same channel as the data block indicated. The function reads the start of the data block, and returns the predBlock field from the data block header.

```
long SONGetPred(short fh, long offset);
```

fh	The file handle.
offset	The file offset to the start of a data block in the file. No check is made to see if this points at a data block.
Returns	A pointer (file offset) to the following data block, or -1 if there are no following blocks.
Errors	None, use with care.

SONGetSucc

This function returns the file offset of the next data block on the same channel as the data block indicated. The function reads the start of the indicated data block, and returns the succBlock field from the data block header.

```
long SONGetSucc(short fh, long offset);
```

fh	The file handle.
offset	The file offset to the start of a data block in the file. No check is made to see if this points at a data block.
Returns	Points to the following data block, or -1 if there are no following blocks.
Errors	None, use with care.

SONSetSucc

This function sets the forward pointer of the data block pointed at by offset to the value held in succOffs. It is used internally to fill in the chain of forward pointers in NormalWrite mode, and when the file is closed in FastWrite mode. Very advanced users may use it to fill in the links after using their own functions to build a SON file.

```
short SONSetSucc(short fh, long offset, long succOffs);
```

fh	The file handle.
offset	Should point to the start of a data block on disk which needs its forward pointer updating. No check is made that this offset points to a valid data block.
succOffs	The offset of the next data block in the file on the same channel. This is copied to the data block header of the block pointed at by offset.
Returns	Zero or a negative error code.
Errors	SON_NO_FILE, SON_BAD_READ, SON_BAD_WRITE

SONFindBlock

This function returns a pointer to the first data block in the file which holds data values which fall into the range sTime to eTime on the particular channel.

```
long SONFindBlock(short fh, WORD chan, TSTime sTime, TSTime eTime);
```

`fh`	The file handle.
`chan`	The data channel in the file to be searched.
`sTime`	The start time of the search in clock ticks.
`eTime`	The end time of the search, in clock ticks.
Returns	A pointer into the file to a block holding data in the time range, or 0 if no data was found, or a negative number specifying an error code.
Errors	`SON_CHANNEL_UNUSED`

SONReadBlock

This function reads a data block on the given channel at the given position into the data buffer assigned for a file.

```
short SONReadBlock(short fh, WORD chan, long position);
```

`fh`	The file handle.
`chan`	The data channel in the file. This is used to get the physical size of the data block to be read. It may be that the written block was smaller, but this will not cause an error.
`position`	The position in the file from which to read. No check is made that this points to the start of a block holding data for the given channel.
Returns	This returns 0 if all went well, otherwise a negative error code.
Errors	`SON_NO_FILE`, `SON_BAD_READ`

SONWriteBlock

This internal function copies an array of data for a channel into the write buffers for that channel, flushing buffered data to disk or discarding data as required.

```
short SONWriteBlock(short fh, WORD chan, , BYTE* buf, long items,
                    int nSize, TSTime sTime, TSTime eTime);
```

`fh`	The file handle.
`chan`	The channel number.
`buf`	A pointer to the array of data values.
`items`	The number of data items pointed at by `buf`.
`nSize`	The size of a data item, in bytes.
`sTime`	The time for the first data item.
`eTime`	The time for the last data item.
Returns	Zero if all was OK, otherwise a negative error code.
Errors	`SON_NO_FILE`, `SON_BAD_WRITE`, `SON_BAD_READ`

SONChanPnt

This function is provided as an efficient method of getting a pointer to the channel description block for a given channel in the file. It is here because old programs use it, however it should be avoided as any use of it to gain access to the channel structures means that a change to the library will very likely break your code.

```
TpChannel SONChanPnt(short fh, WORD chan);
```

`fh`	The file handle.
`chan`	The channel number that the pointer is required for.

Returns Pointer to a structure of type `TChannel` holding the information for the channel.

Errors This function does not produce, or trap errors. If channel is out of range a `NULL` pointer is returned.

SONIntlChanMaxTime

This function is used internally to read through the data on a channel and return the maximum time found.

```
TSTime SONIntlChanMaxTime(short fh, WORD chan);
```

`fh` The file handle.

`chan` The channel number for which the maximum time is required.

Returns The maximum time found, in clock ticks, or a negative error code.

Errors `SON_NO_FILE`, `SON_NO_CHANNEL`, `SON_BAD_READ`

SONIntlMaxTime

This function is used internally to read through the data on all channels and return the maximum time found.

```
TSTime SONIntlMaxTime(short fh);
```

`fh` The file handle.

Returns The maximum time found, in clock ticks, or a negative error code.

Errors `SON_NO_FILE`, `SON_BAD_READ`

SONUpdateMaxTimes

This function is used internally to update the file and channel headers stored in memory with the correct maximum times.

```
long SONUpdateMaxTimes(short fh);
```

`fh` The file handle.

Returns Zero.

SONExtendMaxTime

This function is used internally to update the file maximum time as long as the new maximum is greater than that currently stored.

```
void SONExtendMaxTime(short fh, long time);
```

`fh` The file handle.

time The new maximum time, in clock ticks

SONGetFirstData

This function returns the offset within the file to where the first data block, on any channel, is stored.

```
long SONGetFirstData(short fh);
```

`fh` The file handle.

Returns The file offset.

SONFileHandle

This function returns the operating-system file handle for a SON file. The function is provided to allow the SON filing system to be bypassed - use at your own risk or better still not at all.

```
TFH SONFileHandle(short fh);
```

`fh`	The file handle.
Returns	An operating system dependent file handle.
Errors	None – we assume you know what you are doing.

SONBookFileSpace

Deprecated and may be removed without notice. This routine is dangerous and should only be used after creating a file and before any data is written to it. It allows you to reserve disk space (up to 2 GB only), which may be useful if you are writing data very fast. We strongly suggest that you do not use this routine and we only document it for completeness. Used carelessly, this routine can truncate a file, leading to data loss.

```
short SONBookFileSpace(short fh, long lSpace);
```

`fh`	The file handle.
`lSpace`	The file size is set to this number of bytes.
Returns	0 if OK, or a negative error code.
Errors	`SON_READ_ONLY`, `SON_NO_ACCESS`

SONSetPhySz

This function gets and/or sets the buffer size to use for a channel. If you set the buffer size, it also sets the maximum number of data points that can be stored in a buffer. You can use this after a channel has been created to change the physical buffer size that you set in the channel create call. You must do this before calling `SONSetBuffSpace`() and writing data.

```
void SONSetPhySz(short fh, WORD chan, long lSize);
```

`fh`	The file handle.
`chan`	The data channel.
`lSize`	If >0, this is the new buffer size. It is rounded up to the next multiple of 512 before being used. We assume that that channel is in use! If it is 0, the channel should be unused and this sets the buffer size to 0 and the number of data items per buffer to 0. If <0, no change is made.

Visual Basic and Type Libraries

Type library support

The SON library includes information that allows Visual Basic programmers to use it without the need for DECLARE statements. To use the library from Visual Basic the file son32.dll should be in the WINDOWS folder or in a folder that is included in the system PATH (the places where Windows looks for executable files). To make Visual Basic aware of son32.dll open the Tools menu and select References. In the list of Available references check for the entry SON32 library version 9. Cambridge Electronic Design. If it is already present, make sure that it has a tick next to it. If it is not in the list, use the Browse button to locate the DLL and add it to the list.

Once you have done this Visual Basic (VB) should be aware of the library and you can now use it in your code. To check this, start writing code and type:

```
Dim fh as integer
fh = SONOpenOldFile("demo.smr", 0)
```

If you have correctly linked in the DLL, as soon as you type the left hand bracket VB will prompt you for the arguments of the SONOpenOldFile function.

If you are programming in another language that supports type libraries, you will be able to import the function, type and constant definitions you need to use the library without the need to declare special header files.

Data types and functions

The type library support is aimed primarily at Visual Basic. Because VB does not support all combinations of signed and unsigned types, we have had to lie a little to make sure that all exported types can be handled. In the main, this means pretending that a WORD (unsigned 16-bit number, 0-65535) is the same as a signed 16-bit number (–32768 to 32767). The range 0 to 32767 is the same. For 32768 to 65535, use -32768 to -1.

SON type	VB type	Pass	Comments
short	Integer	ByVal	Exact match.
TpShort	Integer	ByRef	Used to return Integer data values.
TpAdc	Integer	ByRef	Exact match. Used for Integer array.
WORD	Integer	ByVal	Unsigned 16-bit integer from 0-65535.
TpWORD	Integer	ByRef	Used when reading back a WORD value.
TpVoid	Any	ByRef	An argument passed as Any type.
BOOLEAN	Byte	ByVal	0 = FALSE, 1 (non-zero) = TRUE.
TpBOOL	Byte	ByRef	Used to return a BOOLEAN value
TpStr	String	ByRef	Variable length string to read back data.
TpCStr	String	ByRef	Passes a String to the library.
int	Long	ByVal	Both are signed 32-bit integers.
long	Long	ByVal	Both are signed 32-bit integers.
TSTime	Long	ByVal	Both are signed 32-bit integers.
DWORD	Long	ByVal	Unsigned 32-bit integer.
TpFilterMask	TFilterMask	ByRef	Passes a TFilterMask to the library
TpMarker	TMarker	ByRef	Passes a TMarker to the library
TpSTime	Long	ByRef	Arrays of Long and to return a Long
float	Single	ByVal	Single precision floating point data.
TpFloat	Single	ByRef	Arrays of Single and to return a Single
double	Double	ByVal	Double precision floating point data

The following are declared for you by the type import library.

```
const SON_NUMFILECOMMENTS As Long = 5;
const SON_COMMENTSZ As Long = 79;
const SON_CHANCOMSZ As Long = 71;
const SON_UNITSZ As Long = 5;
```

```
const SON_TITLESZ As Long = 9;

Type TMarkBytes
    acCode(0 To 3) As Byte
EndType

Type TMarker
    mark As Long
    mvals As TMarkBytes
EndType

Type TSONTimeDate
    ucHun As Byte
    ucSec As Byte
    ucMin As Byte
    ucHour As Byte
    ucDay As Byte
    ucMon As Byte
    wYear As Integer;
End Type

Type TSONCreator
    acID(0 To 7) As Byte
End Type

Enum TDataKind
    ChanOff=0
    Adc
    EventFall
    EventRise
    EventBoth
    Marker
    AdcMark
    RealMark
    TextMark
    RealWave
End Enum

Type TFilterMask
    amask As Opaque
    lFlags As Long
End Type

const SON_FALLLAYERS As Long = -1;
const SON_FALLITEMS As Long = -1;
const SON_FCLEAR As Long = 0;
const SON_FSET As Long = 1;
const SON_FINVERT As Long = 2;
const SON_FREAD As Long = -1;
const SON_FMASK_ANDMODE As Long = 0;
const SON_FMASK_ORMODE As Long = 0x02000000;

Function SONAppID(ByVal fh As Integer, pCGet As TSONCreator,
   pCSet As TSONCreator) As Long
Function SONBlocks(ByVal fh As Integer, ByVal chan As Integer) As Long
Function SONCanWrite(ByVal fh As Integer) As Byte
Function SONChanDelete(ByVal fh As Integer, ByVal chan As Integer)
   As Integer
Function SONChanDivide(ByVal fh As Integer, ByVal chan As Integer) As Long
Function SONChanInterleave(ByVal fh As Integer, ByVal chan As Integer)
   As Long
Function SONChanKind(ByVal fh As Integer, ByVal chan As Integer)
   As TDataKind
Function SONChanMaxTime(ByVal fh As Integer, ByVal chan As Integer) As
Long
Function SONCloseFile(ByVal fh As Integer) As Integer
Function SONCommitFile(ByVal fh As Integer, ByVal bDel As Byte) As Integer
Function SONCommitIdle(ByVal fh As Integer) As Integer
Function SONCreateFile(name As String, ByVal nChannels As Long,
   ByVal extra As Integer) As Integer

Function SONCreateFileEx(name As String, ByVal nChannels As Long,
   ByVal extra As Integer, ByVal isBig As Integer) As Integer
```

```
Function SONDelBlocks(ByVal fh As Integer, ByVal chan As Integer) As Long
Function SONEmptyFile(ByVal fh As Integer) As Integer
Function SONFControl(pFM As TFilterMask, ByVal layer As Long,
   ByVal item As Long, ByVal set As Long) As Long
Function SONFEqual(pFiltMask1 As TFilterMask, pFiltMask2 As TFilterMask)
   As Byte
Function SONFileBytes(ByVal fh As Integer) As Long
Function SONFileSize(ByVal fh As Integer) As Long
Function SONFilter(pM As TMarker, pFM As TFilterMask) As Long
Function SONFMode(pFM As TFilterMask, ByVal lNew As Long) As Long
Function SONGetADCData(ByVal fh As Integer, ByVal chan As Integer,
   psData As Integer, ByVal max As Long, ByVal sTime As Long,
   ByVal eTime As Long, pbTime As Long, pFiltMask As Any) As Long
Sub SONGetADCInfo(ByVal fh As Integer, ByVal chan As Integer,
   scale As float, offset As float, pcUnt As String, points As Integer,
   preTrig As Integer)
Sub SONGetChanComment(ByVal fh As Integer, ByVal chan As Integer,
   pcCom As String, ByVal sMax As Integer)
Sub SONGetChanTitle(ByVal fh As Integer, ByVal chan As Integer,
   pcTitle As String)
Function SONGetEventData(ByVal fh As Integer, ByVal chan As Integer,
   plTimes As Long, ByVal max As Long, ByVal sTime As Long,
   ByVal eTime As Long, levLowP As Byte, pFiltMask As Any) As Long
Function SONGetMarkData(ByVal fh As Integer, ByVal chan As Integer,
   pMark As TMarker, ByVal max As Long, ByVal sTime As Long,
   ByVal eTime As Long, pFiltMask As Any) As Long
Function SONGetExtMarkData(ByVal fh As Integer, ByVal chan As Integer,
   pMark As TMarker, ByVal max As Long, ByVal sTime As Long,
   ByVal eTime As Long, pFiltMask As Any) As Long
Sub SONGetExtMarkInfo(ByVal fh As Integer, ByVal chan As Integer,
   pcUnt As String, points As Integer, preTrig As Integer)
Function SONExtMarkAlign(ByVal fh As Integer, ByVal n As Long) As Long
Function SONGetExtraData(ByVal fh As Integer, buff As Any,
   ByVal bytes As Integer, ByVal offset As Integer, ByVal writeIt As Byte)
   As Integer
Function SONGetExtraDataSize(ByVal fh As Integer) As Long
Sub SONGetFileComment(ByVal fh As Integer, ByVal which As Integer,
   pcFCom As String, ByVal sMax As Integer)
Function SONGetFreeChan(ByVal fh As Integer) As Integer
Sub SONGetIdealLimits(ByVal fh As Integer, ByVal chan As Integer,
   pfRate As float, pfMin As float, pfMax As float)
Function SONGetNewFileNum() As Integer
Function SONGetRealData(ByVal fh As Integer, ByVal chan As Integer,
   pfData As float, ByVal max As Long, ByVal sTime As Long,
   ByVal eTime As Long, pbTime As Long, pFiltMask As Any) As Long
Function SONGetTimePerADC(ByVal fh As Integer) As Integer
Function SONGetusPerTime(ByVal fh As Integer) As Integer
Function SONGetVersion(ByVal fh As Integer) As Long
Function SONIdealRate(ByVal fh As Integer, wByVal chan As Integer,
   ByVal fIR As Single) As Single
Function SONItemSize(ByVal fh As Integer, ByVal chan As Integer) As
Integer
Function SONKillRange(ByVal fh As Integer, ByVal nChan As Long,
   ByVal sTime As Long, ByVal eTime As Long) As Integer
Function SONLastPointsTime(ByVal fh As Integer, ByVal chan As Integer,
   ByVal sTime As Long, ByVal eTime As Long, ByVal lPoints As Long,
   ByVal bAdc As Byte, pFiltMask As Any) As Long
Function SONLastTime(ByVal fh As Integer, wByVal chan As Integer,
   ByVal sTime As Long, ByVal eTime As Long, pvVal As Any,
   pMB As TMarkBytes, pbMk As Byte, pFiltMask As Any) As Long
Function SONLatestTime(ByVal fh As Integer, ByVal chan As Integer,
   ByVal sTime As Long) As Integer
Function SONMarkerItem(ByVal fh As Integer, ByVal chan As Integer,
   pBuff As Any, ByVal n As Long, pMark As TMarker, pvData As Any,
   bSet As Byte) As Long
Function SONMaxChans(ByVal fh As Integer) As Long
Function SONMaxItems(ByVal fh As Integer, ByVal chan As Integer) As Long
Function SONMaxTime(ByVal fh As Integer) As Long
Function SONOpenOldFile(name As String, ByVal iOpenMode As Long) As
Integer
Function SONPhyChan(ByVal fh As Integer, ByVal chan As Integer) As Long
Function SONPhySz(ByVal fh As Integer, ByVal chan As Integer) As Long
Function SONSave(ByVal fh As Integer, ByVal nChan As Long,
   ByVal sTime As Long, bKeep As Byte) As Integer
```

```
Function SONSaveRange(ByVal fh As Integer, ByVal nChan As Long,
   ByVal sTime As Long, ByVal eTime As Long) As Integer
Sub SONSetADCOffset(ByVal fh As Integer, ByVal chan As Integer,
   ByVal offset As Single)
Sub SONSetADCScale(ByVal fh As Integer, ByVal chan As Integer,
   ByVal scale As Single)
Sub SONSetADCUnits(ByVal fh As Integer, ByVal chan As Integer,
   szUnt As String)
Function SONSetBuffering(ByVal fh As Integer, ByVal nChan As Long,
   ByVal nBytes As Long) As Integer
Function SONSetBuffSpace(ByVal fh As Integer) As Integer
Sub SONSetChanComment(ByVal fh As Integer, ByVal chan As Integer,
   szCom As String)
Sub SONSetChanTitle(ByVal fh As Integer, ByVal chan As Integer,
   szTitle As String)
Function SONSetEventChan(ByVal fh As Integer, ByVal chan As Integer,
   ByVal sPhyCh As Integer, ByVal lBufSz As Long, szCom As String,
   szTitle As String, ByVal fRate As Single, ByVal TDataKind evtKind)
   As Integer
Sub SONSetFileClock(ByVal fh As Integer, ByVal usPerTime As Integer,
   ByVal timePerADC As Integer)
Sub SONSetFileComment(ByVal fh As Integer, ByVal which As Integer,
   szFCom As String)
Sub SONSetInitLow(ByVal fh As Integer, ByVal chan As Integer,
   ByVal bLow As Byte)
Function SONSetMarker(ByVal fh As Integer, ByVal chan As Integer,
  ByVal time As Long, pMark As TMarker, ByVal size As Integer) As Integer
Function SONSetRealChan(ByVal fh As Integer, ByVal chan As Integer,
   ByVal sPhyCh As Integer, dvd As Long, ByVal lBufSz As Long,
   szCom As String, szTitle As String, ByVal scl As Single,
   ByVal offs As Single, szUnt As String) As Long
Function SONSetRealMarkChan(ByVal fh As Integer, ByVal chan As Integer,
   ByVal sPhyCh As Integer, ByVal lBufSz As Long, szCom As String,
   szTitle As String, ByVal fRate As Single, ByVal min As Single,
   ByVal max As Single, szUnt As String, ByVal points As Integer)
   As Integer
Function SONSetTextMarkChan(ByVal fh As Integer, ByVal chan As Integer,
   ByVal sPhyCh As Integer, ByVal lBufSz As Long, szCom As String,
   szTitle As String, ByVal fRate As Single, szUnt As String,
   ByVal points As Integer) As Integer
Function SONSetWaveChan(ByVal fh As Integer, ByVal chan As Integer,
   ByVal sPhyCh As Integer, ByVal dvd As Long, ByVal lBufSz As Long,
   szCom As String, szTitle As String, ByVal scl As Single,
   ByVal offs As Single, szUnt As String) As Integer
Function SONSetWaveMarkChan(ByVal fh As Integer, ByVal chan As Integer,
   ByVal sPhyCh As Integer, ByVal dvd As Long, ByVal lBufSz As Long,
   szCom As String, szTitle As String, ByVal fRate As Single,
   ByVal scl As Single, ByVal offs As Single, szUnt As String,
   ByVal points As Integer, ByVal preTrig As Integer, ByVal nTrace As
Long)
   As Integer
Function SONTextMarkItem(ByVal fh As Integer, ByVal chan As Integer,
   pBuff As Any, ByVal n As Long, pMark As TMarker, pvData As String,
   bSet As Byte) As Long
Function SONTimeBase(ByVal fh As Integer, ByVal dTB As Double) As Double
Function SONTimeDate(ByVal fh As Integer, pTDGet As TSONTimeDate,
   pTDSet As TSONTimeDate) As Long
Function SONUpdateStart(ByVal fh As Integer) As Integer
Function SONWriteADCBlock(ByVal fh As Integer, ByVal chan As Integer,
   psBuf As Integer, ByVal count As Long, ByVal sTime As Long) As Long
Function SONWriteEventBlock(ByVal fh As Integer, ByVal chan As Integer,
   plBuf As Long, ByVal count As Long) As Integer
Function SONWriteExtMarkBlock(ByVal fh As Integer, ByVal chan As Integer,
   pM As TMarker, ByVal count As Long) As Integer
Function SONWriteMarkBlock(ByVal fh As Integer, ByVal chan As Integer,
   pM As TMarker, ByVal count As Long) As Integer
Function SONWriteRealBlock(ByVal fh As Integer, ByVal chan As Integer,
   pfBuff As float, ByVal count As Long, ByVal sTime As Long) As Long
Sub SONYRange(ByVal fh As Integer, ByVal chan As Integer, pfMin As float,
   pfMax As float)
Function SONYRangeSet(ByVal fh As Integer, ByVal chan As Integer,
   ByVal fMin As Single, ByVal fMax As Single) As Long
```

The declarations are equivalent to those given in the C/C++ description of the manual.

Reading back String data

You must be very careful when using any routine that reads back string data to be certain that the string you pass is long enough to hold all the returned data. This applies to the following routines: `SONGetFileComment`, `SONGetChanComment`, `SONGetChanTitle`, `SONGetADCInfo`, `SONGetExtMarkInfo`. Here's how to do it safely:

```
'A utility to tidy up a 0 terminated string from our DLL
Function StrTrim(s As String) As String
Dim i As Integer
i = InStr(s, vbNullChar)
If i > 0 Then StrTrim = Left$(s, i - 1) Else StrTrim = s
End Function
...
...
Dim szTitle As String
Dim sByVal fh As Integer
sFh = SONOpenOldFile("demo.smr", 0)                          'open a file
if sFh < 0 Then End                                        'quit if a problem
szTitle = String(SON_TITLESZ + 1, vbNullChar)        ' fill with NULL chars
SONGetChanTitle sFh, 0, szWork               'read back the channel 0 title
szTitle = StrTrim(szTitle)               'discard any extra NULL characters
SONCloseFile sFh                               'Tidy up by closing the file
```

A Visual Basic string has a length that is independent of the string contents and that is maintained by Visual Basic. Our DLL does not know it is dealing with Visual Basic and just assumes that the string is large enough to hold all the characters plus a zero terminator.

To work safely, we must ensure that any string passed to a DLL function is long enough for the maximum number of characters plus a zero terminating character. We do this by setting the string to a known size and filled with `Null` characters with `String()`. In the example above, when we get the string back, Visual Basic still thinks it is `SON_TITLESZ + 1` characters in length, so we use the `StrTrim` function to find the terminating `Null` character inserted by the DLL and return a string of the correct length.

The NULL pointer problem

Most routines that return values state that if you pass a `NULL` pointer, nothing is returned. This is to save you from the bother of defining variables just to ignore the results. Unfortunately, Visual Basic does not allow you to define a type safe argument and then pass a `NULL` pointer to it. In most cases you will have to define variables just so that you can call the function even if you don't need the returned value.

In a few cases we have declared such arguments `As Any`. In most cases this is where the argument is a `TFilterMask`, for example in the `SONGetADCData()` function. This allows you to pass `ByVal 0&` as the last argument rather or declare and fill in a `TFilterMask` variable. You must not pass anything else as an argument as this could lead to an illegal memory reference.

The `SONLastTime()` function has a special problem. The `pvVal` argument is also passed `As Any`. It is always safe to use `ByVal 0&` for this argument. However, if you wish to return a value, the variable you use must match the type for the channel (`Integer` for `Adc` and `EventBoth` channels, Single for `RealWave` channels).

Read waveform data

This example reads through all the `Adc` data in a channel of a file, and finds the mean level of the channel. It reads up to 1000 points at a time until end of file or an error. `SONGetADCData` returns continuous data, any gap in the file terminates the read before the maximum number of points that will fit in the buffer. The routine returns the number of points read and the time of the first point. To read more data we must calculate the

time of the next point after the end of the buffer. To get the result in user units, read back the scale and offset with `SONGetADCInfo()` and use these to convert the result.

```
Function AverageAdc(ByVal fh As Integer, ByVal chan As Integer) As Double
If SONChanKind(fh, 0) <> Adc Then Exit Sub 'Check a useful channel
Const NBUFSZ As Long = 1000  'The buffer size to use
Dim intBuff(0 To NBUFSZ-1) As Integer 'Buffer to read data into
Dim lTime As Long            'time to read from and to
Dim dAcc As Double           'to accumulate data for mean
Dim lPoints As Long          'number of points
Dim nRead As Long            'number of points read
Dim i As Integer             'loop counter

lPoints = 0                  'initialise the points
dAcc = 0#                    'initialise the accumulator
lTime = 0                    'start searching here
nRead = 1                    'so loop works as expected
While nRead > 0
    nRead = SONGetADCData(fh, chan, intBuff(0), NBUFSZ, lTime,
                                        SONMaxTime(fh), lTime, ByVal 0&)
    If nRead > 0 Then
        lPoints = lPoints + nRead 'increase count of points read
        For i = 0 To nRead - 1    'accumulate the data
            dAcc = dAcc + intBuff(i)
        Next
        lTime = lTime + nRead * SONChanDivide(fh, 0) ' next read time
   End If
Wend

If (lPoints > 0) Then dAcc = dAcc / lPoints
AverageAdc = dAcc          'return the value
End Function
```

Real RealWave data

This example is the same are the previous one for `Adc` data, but it works for `RealWave` and `Adc` channels. No scaling is required as the results are already in user units.

```
Function AverageReal(ByVal fh As Integer, ByVal chan As Integer) As Double
Dim kind As TdataKind       'the type of data
kind = SONChanKind(fh, 0)
if (kind <> RealWave) And (kind <> Adc) Then Exit Sub 'Check channel type
Const NBUFSZ As Long = 1000  'The buffer size to use
Dim rBuff(0 To NBUFSZ-1) As Single 'Buffer to read data into
Dim lTime As Long            'time to read from and to
Dim dAcc As Double           'to accumulate data for mean
Dim lPoints As Long          'number of points
Dim nRead As Long            'number of points read
Dim i As Integer             'loop counter
lPoints = 0                  'initialise the points
dAcc = 0#                    'initialise the accumulator
lTime = 0                    'start searching here
nRead = 1                    'so loop works as expected
While nRead > 0
    nRead = SONGetRealData(fh, chan, rBuff(0), NBUFSZ, lTime,
                                            SONMaxTime(fh), lTime,
ByVal 0&)
    If nRead > 0 Then
        lPoints = lPoints + nRead 'increase count of points read
        For i = 0 To nRead - 1    'accumulate the data
            dAcc = dAcc + rBuff(i)
        Next
        lTime = lTime + nRead * SONChanDivide(fh, 0) ' next read time
   End If
Wend

If (lPoints > 0) Then dAcc = dAcc / lPoints
AverageReal = dAcc         'return the value
End Function
```

Read event data

This next routine will count the number of events, markers or extended markers between two times. If you try to run this on a channel that does not contain suitable data, you will get a negative error code returned by `SONGetEventData()`.

```
Function CountEvents(ByVal fh As Integer, ByVal chan As Integer,
                              ByVal lFrom As Long, ByVal lTo As Long) As
Long
Dim kind As TDataKind
Const NBUFSZ As Long = 1000   'The buffer size to use
Dim lBuff(0 to NBUFSZ-1) As Long 'Buffer to read data into
Dim nRead As Long             'number of events read
Dim lTotal As Long            'total number of events
Dim bLevLow As Byte           'the EventBoth level flag

lTotal = 0                    'no events yet
nRead = NBUFSZ                'so our loop starts OK
While nRead = NBUFSZ          'no point going on if buffer not full
   nRead = SONGetEventData(fh, chan, lBuff(0), NBUFSZ,
                                                    lFrom, lTo, bLevLow,
ByVal 0&)
   If nRead > 0 Then
      lTotal = lTotal + nRead 'increase number of events we know of
      lFrom = lBuff(nRead - 1) + 1 'to find the next event
   End If
Wend
CountEvents = lTotal          ' return the result
End Function
```

In this case we iterate through the data by moving the start time on to be the time of the last event read plus one clock tick. There is no point running around the loop again if the last read did not fill the buffer.

Read Marker data

You can read any `Marker`, `RealMark`, `TextMark` or `AdcMark` channel into an array of `TMarker` as if it were a marker channel. The following example displays all the first marker code and the marker time in a message box for all markers in a channel.

```
Sub ReadMarker(ByVal fh As Integer, ByVal chan As Integer)
Const NBUFSZ As Long = 1000    'The buffer size to use
Dim Mark(0 To NBUFSZ - 1) As TMarker 'Buffer to read data into
Dim nRead As Long              'number of events read
Dim lFrom As Long              'total number of events
Dim dTick As Double            'Tick length in seconds
Dim i As Integer               'loop counter

dTick = SONGetusPerTime(fh)    '
dTick = dTick * SONTimeBase(fh, -1#)
lFrom = 0                      'start time for reading data
nRead = NBUFSZ                 'so our loop starts OK
While nRead = NBUFSZ           'no point going on if buffer not full
   nRead = SONGetMarkData(fh,chan,Mark(0),NBUFSZ,lFrom,&H7FFFFFFF,ByVal
0&)
   If nRead > 0 Then
      For i = 0 To nRead - 1
         MsgBox ("Code:" & Str(Mark(i).mvals.acCode(0)) & " at "
                                                  & Str(Mark(i).Mark *
dTick))
      Next
      lFrom = Mark(nRead - 1).Mark + 1 'to find the next event
   End If
Wend
End Sub
```

In this case we are searching all the data in the file, so we start at time 0 and ask for all data up to and including `&H7fffffff`, which is the maximum value of a `Long`.

Read TextMark data

To read back a channel of `TextMark` data and access the associated text string is a problem for Visual Basic because the size of a `TextMark` data item in a channel depends on the maximum number of characters set for that channel. This means that you cannot create a suitable array. There are two ways around this:

1. Use the built-in `TTextMark` type and read one marker at a time from the file. This type reads the test into an array of Byte that is big enough for any possible data. You will then have to write code to search the array of bytes to find the first 0 and then copy the data up to that point into a string, one character at a time. This is OK if you only have a few TextMark values to read, or if time is not important.

2. Read the data into a large buffer, then use the `SONTextMarkItem` routine to extract one value as a `TMarker` and a `String`. This is illustrated by the next example.

```
Sub ReadTextmark(ByVal fh As Integer, ByVal chan As Integer)
Dim itemSize As Integer
Const BUFFSIZE As Integer = 2000    'use a buffer this size
Dim buff(0 To BUFFSIZE - 1) As Byte 'make a buffer for 100 items
Dim maxItems As Integer         'number to read at a time
Dim nRead As Long               'number of events read
Dim lFrom As Long               'Used as the start time of each read
Dim i As Integer                'loop counter
Dim Mk As TMarker               'variable to hold each marker
Dim S As String                 'variable to hold each string

If SONChanKind(fh, chan) <> TextMark Then Exit Sub
itemSize = SONItemSize(fh, chan)   'get size of each item
maxItems = BUFFSIZE / itemSize 'calculate maximum items per buffer
lFrom = 0                       'start time of our search
nRead = maxItems                'so our loop starts OK
While nRead = maxItems          'no point going on if buffer not full
   nRead = SONGetExtMarkData(fh, chan, buff(0), maxItems,
                                                    lFrom, &H7FFFFFFF,
ByVal 0&)
   If nRead > 0 Then
      For i = 0 To nRead - 1
         S = String(itemSize, vbNullChan) 'Ensure string is long enough
         SONTextMarkItem fh, chan, buff(0), i, Mk, S, 0
         S = StrTrim(S)         'tidy up the string
         MsgBox (S)             'display the string
      Next
      lFrom = Mk.Mark + 1       'to find the next event
   End If
Wend
End Sub
```

To make sure that the string is long enough, we set it to be the size of the data item. As the data item includes a `TMarker` at the start, and the `TMarker` is 8 bytes long, we could have used `String(itemSize-8, vbNullChan)`, but the length does not matter as long as it is long enough. We could also have used `SONGetExtmarkInfo()` to get the size, as is done for the next example using `AdcMark` data. You must set the size of the string each time you pass it to the SON library. If you do not, and the string is too small, you will crash Visual Basic.

Read AdcMark data

Reading this data type is a problem in Visual Basic because the number of waveform points can vary between channels. We solve this by reading blocks of data into a fixed size buffer, and then we extract individual data items as markers and as waveforms using the `SONMarkerItem()` function.

We iterate through the data exactly as we did for the The `Adc` array is declared with variable size, then the size is set based on the number of points held in the channel. It is vital that the array is the correct size. If it is too small you will crash Visual Basic.

```
Sub ReadAdcMark(ByVal fh As Integer, ByVal chan As Integer)
Dim itemSize As Integer
Const BUFFSIZE As Integer = 8000    'use a buffer this size
Dim buff(0 To BUFFSIZE - 1) As Byte 'make a buffer for reading
Dim maxItems As Integer         'number to read at a time
Dim nRead As Long               'number of events read
Dim lFrom As Long               'Used as the start time of each read
Dim i As Integer                'loop counter
Dim Mk As TMarker               'variable to hold each marker
Dim points As Integer           'for points attached to each marker
Dim Adc() As Integer            'will be the waveform data

If SONChanKind(fh, chan) <> AdcMark Then Exit Sub
itemSize = SONItemSize(fh, chan)   'get size of each item
maxItems = BUFFSIZE / itemSize 'calculate maximum items per buffer
SONGetExtMarkInfo fh, chan, vbNullString, points, i
ReDim Adc(0 To points - 1)      ' correctly sized buffer

lFrom = 0                       'start time of our search
nRead = maxItems                'so our loop starts OK
While nRead = maxItems          'no point going on if buffer not full
   nRead = SONGetExtMarkData(fh, chan, buff(0), maxItems, lFrom,
                                                              &H7FFFFFFF,
ByVal 0&)
   If nRead > 0 Then
      For i = 0 To nRead - 1
         SONMarkerItem fh, chan, buff(0), i, Mk, Adc(0), 0
         'the data is in now in Mk and Adc
      Next
      lFrom = Mk.Mark + 1       'to find the next event
   End If
Wend
End Sub
```

Read RealMark data

This example is exactly the same as the previous one for `AdcMark` data, but it works for a `RealMark` channel.

```
Sub ReadRealMark(ByVal fh As Integer, ByVal chan As Integer)
Dim itemSize As Integer
Const BUFFSIZE As Integer = 8000    'use a buffer this size
Dim buff(0 To BUFFSIZE - 1) As Byte 'make a buffer for reading
Dim maxItems As Integer         'number to read at a time
Dim nRead As Long               'number of events read
Dim lFrom As Long               'Used as the start time of each read
Dim i As Integer                'loop counter
Dim Mk As TMarker               'variable to hold each marker
Dim points As Integer          'for points attached to each marker
Dim Real() As Single           'will be the RealMark data

If SONChanKind(fh, chan) <> RealMark Then Exit Sub
itemSize = SONItemSize(fh, chan)   'get size of each item
maxItems = BUFFSIZE / itemSize 'calculate maximum items per buffer
SONGetExtMarkInfo fh, chan, vbNullString, points, i
ReDim Real(0 To points - 1)    ' correctly sized buffer

lFrom = 0                       'start time of our search
nRead = maxItems                'so our loop starts OK
While nRead = maxItems          'no point going on if buffer not full
   nRead = SONGetExtMarkData(fh, chan, buff(0), maxItems, lFrom,
                                                              &H7FFFFFFF,
ByVal 0&)
   If nRead > 0 Then
      For i = 0 To nRead - 1
         SONMarkerItem fh, chan, buff(0), i, Mk, Real(0), 0
         'the data is in now in Mk and Real
      Next
      lFrom = Mk.Mark + 1       'to find the next event
   End If
Wend
End Sub
```

Writing data

The following example code writes one block of each type of channel. This is straightforward except for the `TextMark`, `AdcMark` and `RealMark` channels, which need special treatment to build up a buffer of data to write.

```
Sub SonCreate()
Dim fh As Integer
fh = SONCreateFile("test.smr", 32, 0)              'Makes a 32 channel file
If fh < 0 Then Exit Sub                            'give up if create fails

SONSetFileClock fh, 10, 1 '10 us per clock tick

'Channel 0 will be simple events, expected rate is 100 Hz
SONSetEventChan fh, 0, -1, 4000, "Comment", "Title", 100#, EventFall

'Channel 1 is a waveform at 500 Hz (200 * 10 us is 2 ms = 500 Hz)
SONSetWaveChan fh, 1, -1, 200, 4000, "Comment", "Title", 1#, 0#, "Volts"

'Channel 2 is an AdcMark at 20 kHz, 32 points, 10 pretrigger, interleave 1
SONSetWaveMarkChan fh, 2, -1, 5, 4000, "Comment", "Title",
                                          30#, 1#, 0#, "Volts", 32, 10, 1

'Channel 3 is a TextMark with 79 characters + 1 terminator
SONSetTextMarkChan fh, 3, -1, 4000, "Comment", "Title", 1#, "Unit!", 79

'Channel 4 is a RealMark with 1 value, expected range -10.0 to 10.0
SONSetRealMarkChan fh, 4, -1, 4000, "Comment", "Title",
                                                1#, -10#, 10#, "User", 1

'Channel 5 is a RealWave at 10 Hz
SONSetRealChan fh, 5, -1, 10000, 4000, "Comment", "Title",   1#, 0#,"Units"

'Channel 6 is a Marker at 1 Hz
SONSetEventChan fh, 6, -1, 4000, "Comment", "Title", 1#, Marker

'Now allocate the file space (we could call SONSetBuffering() first).
SONSetBuffSpace fh

'Now we can start to write the data. Events first
Dim i As Long                   ' Loop counter
Dim Events(0 To 3999) As Long   ' 10 seconds worth of events 10 ms apart
For i = 0 To 999
    Events(i) = i * 1000        ' 10 ms apart
    Next
If SONWriteEventBlock(fh, 0, Events(0), 4000) <> 0 Then MsgBox("Event")

'Adc data
Dim Adc(0 To 4999) As Integer   ' 10 seconds worth of waveform at 500 Hz
For i = 0 To 4999
    Adc(i) = -32768 + i * 12    ' ramp up
    Next
If SONWriteADCBlock(fh, 1, Adc(0), 5000, 0) < 0 Then MsgBox("Adc")

'AdcMark
Dim Spike(32) As Integer
For i = 0 To 31
    Spike(i) = i * 1000
    Next
Const BUFSZ As Integer = 8000
Dim buff(8000) As Byte          ' work space to build extended marker data
Dim maxItems As Integer         ' to hold maximum items in the buffer
Dim Mk As TMarker               ' space for marker part
maxItems = BUFSZ / SONItemSize(fh, 2)
For i = 0 To 3
    Mk.mvals.acCode(i) = i
    Next
For i = 0 To maxItems
    Mk.Mark = i * 2000          ' 2 milliseconds apart
    Mk.mvals.acCode(0) = i And 3 ' set a code
    SONMarkerItem fh, 2, buff(0), i, Mk, Spike(0), 1 'copy data into
buffer
    Next
If SONWriteExtMarkBlock(fh,2,buff(0),maxItems) <> 0 Then MsgBox("AdcMark")
```

```
'Textmark
Dim S As String
maxItems = BUFSZ / SONItemSize(fh, 3) 'calculate maximum in the buffer
For i = 0 To maxItems
    Mk.Mark = i * 2000          ' 2 milliseconds apart
    Mk.mvals.acCode(0) = i And 3 ' set a code
    S = "Text marker #" & Str(i)
    SONTextMarkItem fh, 3, buff(0), i, Mk, S, 1 'copy data into buffer
    Next
If SONWriteExtMarkBlock(fh,3,buff(0),maxItems) <> 0 Then
MsgBox("TextMark")

'RealMark
Dim data As Single              ' the value to add to the marker
maxItems = BUFSZ / SONItemSize(fh, 4) 'calculate maximum in the buffer
For i = 0 To maxItems
    Mk.Mark = i * 2000          ' 2 milliseconds apart
    Mk.mvals.acCode(0) = i And 3 ' set a code
    data = i / 100#             ' set a value
    SONTextMarkItem fh, 4, buff(0), i, Mk, data, 1 'copy data into buffer
    Next
If SONWriteExtMarkBlock(fh,4,buff(0),maxItems) <> 0 Then
MsgBox("RealMark")

'RealWave
Dim Real(0 To 99) As Single     ' 10 seconds worth of data
For i = 0 To 99
    Real(i) = i / 10#
    Next
If SONWriteRealBlock(fh, 5, Real(0), 100, 0) < 0 Then MsgBox("RealWave")

'Marker
Dim marks(0 To 9) As TMarker    'The marker data
Dim j As Integer
For i = 0 To 9
    marks(i).Mark = i * 100000  'One per second
    marks(i).mvals.acCode(0) = i
    For j = 1 To 3
        marks(i).mvals.acCode(j) = 0
        Next
    Next
If SONWriteMarkBlock(fh, 6, marks(0), 10) <> 0 Then MsgBox("Marker")

SONCloseFile fh

End Sub
```

There is an alternative way to write the extended marker types without using a `Byte` buffer. To do this, write one `TTextMark`, `TAdcMark` or `TRealMark` at a time. These types are defined for the largest allowed sizes of data items, so this will work regardless of the real size of a data item on the channel. For example, the code for writing `AdcMark` data could have been written as:

```
Dim Spk As TAdcMark
For i = 0 To 3
    Spk.m.mvals.acCode(i) = i
    Next
For i = 0 To 31
    Spk.a(i) = i * 1000
    Next
For i = 0 To maxItems
    Spk.m.Mark = i * 2000          ' 2 milliseconds apart
    Spk.m.mvals.acCode(0) = i And 3 ' set a code
    If (SONWriteExtMarkBlock(fh, 2, Spk, 1) <> 0) Then MsgBox ("AdcMark")
    Next
```

The version 9 SON library can be built for use under 32-bit Windows and Linux. At the time of writing, the Linux version is untested and likely needs minor adjustment. The version 8 library supported a wider range of environments, including 16-bit DOS and Windows and the Macintosh. CED uses an include file, `MACHINE.H`, to detect the environment and define various types and functions in a machine-independent fashion, the `MACHINE.H` file is required if you are going to build SON.

SON for 32-bit Windows

For 32-bit Windows use, the SON library is supplied as a DLL, `SON32.DLL`. This is linked to your application at run-time, a library stub, `SON32.LIB`, carries out the run-time linkage. You will need `MACHINE.H`, `SON.H`, `SON32.LIB` and `SON32.DLL` to build and run your program. These are all on the SON library distribution disk. The 32-bit SON library can also be directly used from 32-bit C++ programs. Use with non-Microsoft C or C++ may require editing of `MACHINE.H` and `SON.H`. Use with other languages will require a set of external function definitions.

SON for Linux

At the time of writing, to run under Linux you will need to obtain the source code, which means that you will need to sign a non-disclosure agreement. If there is sufficient demand, we may produce a library version that can be used with header files.

Index

—T—

—V—

—W—