Axon Binary File (ABF) Format

Version 2.0.9

User Guide

ABF Version 2.0.9
January 2021

MOLECULAR DEVICES

Patents: http://www.moleculardevices.com/patents

Product manufactured by Molecular Devices, LLC.
3860 N. First Street, San Jose, California, 95134, United States of America.
Molecular Devices, LLC is ISO 9001 registered.

Contents

Chapter 1: Axon Binary File Format Overview

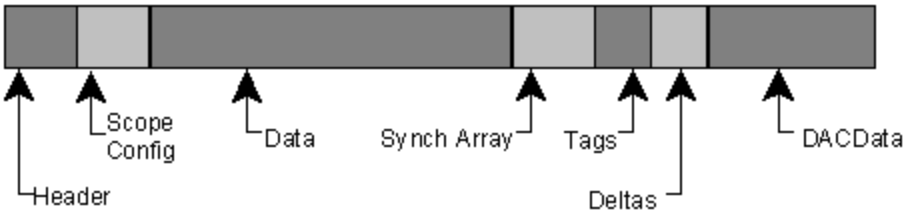
The Axon™ Binary File format (ABF) was created for the storage of binary experimental data. It originated with the pCLAMP suite of data acquisition and analysis programs, but is also supported by AxoScope software.

These files can be created and read on computers running Microsoft Windows. For optimal acquisition performance the binary data are written in the byte order convention of the acquisition computer.

The ABF File Structure

The AXON BINARY FILE has a proprietary format, however the files can be read (and created) by third-party developers by using the ABFFIO library.

An ABF file is made up of a number of sections as follows:



The header and data sections appear in the order shown. The other sections may appear in any order since they are pointed to by parameters in the header. All sections are buffered in blocks of 512 bytes each. The starting location of a section is given as a block number. Block number 0 represents the start of the file. If the block-number pointer to a section (other than the header) is zero, the corresponding section is not written.

ABF version 2 (released with pCLAMP 10 in 2006) is a major upgrade from previous versions of ABF. The major change is that the file header is now of variable length. The impact of this is that it is no longer possible to read the Data (or other sections) directly from the file; this must be done using the ABFFIO.DLL library.

- The ABF Header Section
- The ABF Scope Config Section
- The ABF Data Section
- The ABF Synch Section
- The ABF Tag Section
- The ABF Deltas Section
- The DAC Data Section

History

Prior to pCLAMP software Version 6.0, two types of files were generated : CLAMPEX files for stimulated episodic acquisition and FETCHEX files for gap-free and event detected files. AxoTape for DOS Version 1.x also generated FETCHEX type binary data files. pCLAMP software Version 6.0 merged these two file formats into the ABF file format, which was subsequently adopted by AxoTape for DOS Version 2.0, and AxoScope software for Windows Version 1.0.

For a detailed description of old FETCHEX and CLAMPEX files refer to the manual for pCLAMP software Version 5.x or earlier.

Molecular Devices released pCLAMP software Version 10 in 2006, which also included ABF Format Version 2.0, a major upgrade.

Status

Axon File Support Pack Version 2.0 is the current version for Microsoft Windows. Third parties using these modules should understand that there might be minor changes to the functional interface in future releases. Molecular Devices attempst to document interface changes in the change history, but does not accept any liability for inconvenience caused by changes that are made, whether documented fully or not.

Change History

Version 2.0.9

- Added support for up to 50 waveform epochs.
- Added support for pre-programmable digital or analog output in gap-free mode.
- Increased ABF_STATS_REGIONS from 20 to 24

Version 2.0.6

- Added support for the 1.081 second HumSilencer inter-sweep learning period in episodic stimulation mode.

Version 2.0.5

- Added support for the Digidata 1550A digitizer featuring HumSilencer™ adaptive noise cancellation on Analog In Channel #0.
- Added HumSilencer option to ABFScopeConfig.

Version 2.0.4

- Added support for a configurable timeout interval when using an external trigger.

Version 2.0.3

- Added support for the Digidata 1550 digitizer featuring: 8 analog inputs, 8 analog outputs, 8 digital outputs.

Version 2.0.1

- Added support for file compression.
- Added constants to identify digitized types.

Version 2.0

- Major internal changes
- Added support for 4 waveform output channels.
- Added support for “fast” and “slow” sample rates per epoch in episodic stimulation mode.

Version 1.80

- Added statistics mode for each region: mode is cursor region, epoch etc.

Version 1.79

- Removed data reduction (now MiniDigi only)

Version 1.78

- Added separate entries for alternating DAC and digital outputs

Version 1.77

- Added major, minor and bugfix version numbers

Version 1.76

- Added digital trigger out flag.

Version 1.75

- Added polarity for each channel.

Version 1.74

- Added channel_count_acquired.

Version 1.73

- Added post-processing lowpass filter settings. When filtering is done in pCLAMP software it is stored in the header.

Version 1.72

- Added alternating outputs.

Version 1.71

- Added epoch resistance.

Version 1.70

- Added data reduction.

Version 1.69

- Added user entered percentile levels for rise and decay stats.

Version 1.68

- Expanded ABFScopeConfig.

Version 1.67

- Train epochs, multiple channel and multiple region stats

Version 1.65

- Telegraph support added.

Version 1.6

- Expanded header to 5120 bytes and added extra parameters to support 2 waveform channels.

Version 1.5

- Changed ABFSignal parameters from UUTop & UUBottom to fDisplayGain & fDisplayOffset.
- Added and changed parameters in the 'File Structure', 'Display Parameters', 'DAC Output File', 'Autopeak Measurements' and 'Unused space and end of header' sections of the ABF file header.
- Expanded the ABF API and error return codes.

Version 1.4

- Removed support for big-endian machines.

Version 1.3

- Added support for Bells during before or after acquisitions.
- Added the parameters to describe hysteresis during event detected acquisitions: nLevelHysteresis and lTimeHysteresis.
- Added support for automatic byte reversal.
- Dropped support for BASIC and Pascal.
- Added the ABF Scope Config section to store scope configuration information.

Version 1.2

- Added nDataFormat so that data can optionally be stored in floating point format.
- Added lClockChange to control the multiplexed ADC sample number after which the second sampling interval commences.

Version 1.1 was released in April 1992.

Existing Applications

Third party

Support for Axon's Binary File (ABF) format has been incorporated into the following categories of third-party (non-Axon) products.

(i) Special purpose analysis programs written in laboratories by individual researchers.

(ii) Special purpose commercial analysis programs.

(iii) General purpose commercial graphics and scientific analysis programs.

(iv) Public domain acquisition programs that run on Axon Instruments digitizers.

(v) Special purpose commercial acquisition programs.

(vi) Special purpose commercial data conversion programs.

(viii) Special purpose acquisition programs written in laboratories by individual researchers.

Axon Instruments / Molecular Devices

Raw data acquired by Axon's data acquisition programs are stored in ABF format. Current programs are AxoScope software and Clampex software. Older programs are AxoScope software, AxoTape and the two pCLAMP acquisition programs (Clampex software, Fetchex). All of the pCLAMP programs read ABF data.

A floating point version of the ABF format is used for intermediate storage of analyzed data by Axon Instrument's pCLAMP software (part of the pCLAMP suite) and data exported by Clampex software (version 7 and later).

For data exchange to other programs, current pCLAMP software (AxoScope software , Clampex software , pCLAMP software) creates ATF files, as did the discontinued Axon electrophysiology software (Fetchan, pSTAT, AxoTape, AxoData) and Axon imaging software (Axon Imaging Workbench, AxoVideo). Axon's discontinued Fetchan event-detection software (part of the pCLAMP software DOS suite) also stored the idealized record of data transitions in EVL format files.

Advantages of using the ABF Function API

One of the goals of the ABF reading routines is to isolate the applications programmer from the need to know anything other than the most basic information about the file format. If when working with the ABF reading routines you find that you are overwhelmed by details, stop -- this is a sign that you are not using the proper functions.

In ABF versions 1.x, it was possible to interact with an ABF data file directly, using the information in the header as a ”road map” of the ABF file layout and characteristics, however this was discouraged and Axon built a great deal of useful functionality into its ABF functional interface (the ABF Function API), some of which is documented below.

ABF 2.0 takes this a step further – the format of the information written to the file now uses a header of variable length. This means that it is now essential to use the ABFFIO.DLL library to access the data. The ABF Function API described below allows all the information contained in the file to be readily accessed and insulates the programmer from future changes.

Episodic Timebase Information

The calculation of time-axis values from the header parameters can be complicated due to the possibility of a transition within the sweep from one sampling rate to another faster or slower rate. The ABF routines provide a function (ABF_GetTimebase) that returns the complete timebase in time units from the data file. The ABF_GetTimeBase function can be used for any type of data format: episodic, gap-free , variable-length, etc. All types of data are treated as the same with the ABF routines; therefore it is much clearer and consistent to use the ABF_GetTimeBase in conjunction with the ABF_GetStartTime function to determine the time in which a sample was acquired.

Retrieving Stimulus Waveform Descriptions

Retrieving the stimulus waveform can be difficult if only the ABF routines are used. This is because the waveform may be described either by the Epoch definitions in the header, or by a "DACFile" block at the end of the file. The ABF_ReadWaveform, however, handles either of these cases transparently, and will form the stimulus waveform array as an array of samples corresponding to the time base.

Retrieving Math Signal Data

The Math Signal is an algebraic combination of two ADC channels, described by parameters in the Math Signal section of the ABF header. Math signal data may be retrieved through the ABFH_ReadChannel function, using a channel number of -1. (This channel is only available if the Math channel is enabled, with the nArithmeticEnable flag set in the file header).

What Kind Of Data Are Stored In ABF Files?

Axon Instruments data acquisition programs acquire five types of data, all of which are stored in ABF format files.

(1) Gap Free. (nOperationMode = 3)

Gap-free ABF files contain a single sweep of up to 4 GB of multiplexed data. A uniform sampling interval is used throughout. There is no stimulus waveform associated with gap-free data. Gap-free mode is usually used for the continuous acquisition of data in which there is a fairly uniform activity over time.

(2) Variable-Length Event-Driven. (nOperationMode = 1)

(3) Fixed-Length Event-Driven. (nOperationMode = 2)

In these two event-driven data acquisition modes, data acquisition is initiated in segments whenever a threshold-crossing event is detected. There is no stimulus waveform associated with these two operation modes.

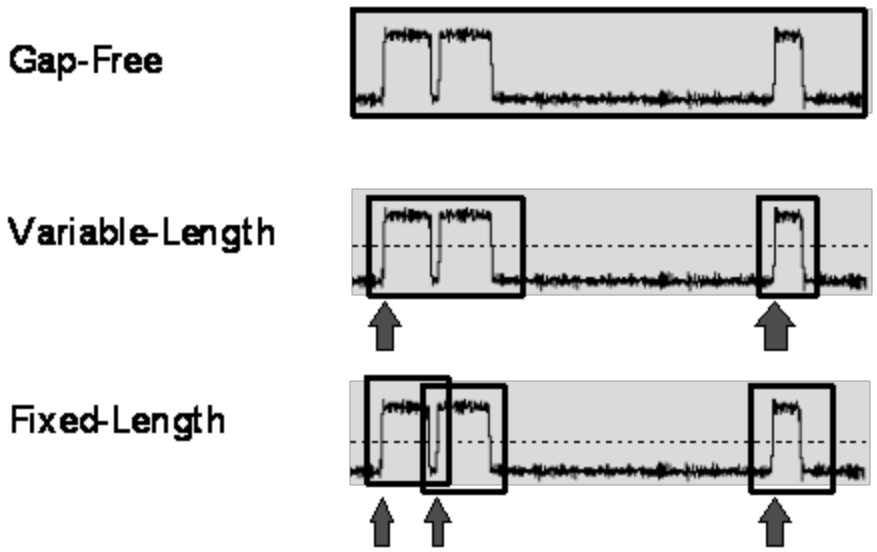


Figure 1-1: Graphical comparison of gap-free acquisition with the event driven modes of acquisition.

In variable-length event-driven acquisition, pre-trigger and trailing portions below threshold are also acquired. The length of the segment of data is determined by the nature of the data, being automatically extended according to the amount of time that the data exceeds the threshold. If the pre-trigger portion of the next event would overlap the trailing portion of the current event, the current segment is extended. There is no storage of overlapping data. The precise start time and length of each segment is stored in the Synch Array.

Variable-length event-driven acquisition is usually used for the continuous recording of "bursting" data in which there are bursts of activity separated by long quiescent periods.

In fixed-length event-driven acquisition, a pre-trigger portion below threshold is acquired. Unlike variable-length event-driven acquisition, the length of each segment of data is a pre-specified constant for all segments. For this reason, the segments are often referred to as sweeps. In this mode, every threshold crossing triggers a sweep; therefore fixed-length event-driven mode is also sometimes referred to as loss-free oscilloscope mode. If a second event occurs before the current sweep is finished, a second sweep is acquired triggered from the second event. This occurrence is referred to as overlap. In this case, consecutive sweeps in the data file contain redundant data.

The precise start time and length of each sweep is stored in the Synch Array. Although the length of each sweep is redundant in this mode, it is stored in order to simplify reading and writing of the Synch Array.

Similarly, the storage of redundant data during overlap is not strictly necessary, but it simplifies analysis and display for each sweep to be returned as a fixed-length sweep with a known and constant trigger time. Since no triggers are lost, fixed-length event-driven acquisition is ideal for the statistical analysis of constant-width events such as action potentials.

(4) High-Speed Oscilloscope Mode. (nOperationMode = 4)

Like fixed-length event-driven acquisition, in high-speed oscilloscope mode a pre-trigger portion below threshold is acquired. Unlike fixed-length event-driven acquisition, in high-speed oscilloscope mode not every threshold crossing triggers a sweep. The emphasis is on allowing the digitizer to be used at the highest possible sampling rate. Like a real high-speed oscilloscope, there is a "dead time" at the end each sweep during which the display is updated and the trigger circuit is re-armed. Threshold crossings that arrive during this dead time are simply ignored. Similarly, second and subsequent threshold crossings during a sweep do not start a new sweep. Thus there is no storage of overlapping (redundant) data.

Although the acquisition conditions are different for fixed-length event-driven and high-speed oscilloscope modes, in practice the data file formats are identical and analysis programs can treat them identically. The only caution is that because of the storage of overlapping data that is possible in fixed-length event-driven acquisition, the start time of a sweep might occur before the end of the previous sweep.

(5) Episodic Stimulation Mode. (nOperationMode = 5)

In this mode, a number of equal-length sweeps (also known as episodes) are acquired. A set of parametrically related sweeps is called a run. Runs can be repeated a specified number of times to form a trial. If runs are repeated, the corresponding sweeps in each run are automatically averaged and the trial contains only the average. The trial is stored in a file. Only one trial can be stored in an ABF file.

Within each sweep a complex stimulus waveform consisting of up to ten epochs can be generated. One output sample is generated for each A/D conversion. This also applies to the multiplexed A/D conversions. For example, if there are three multiplexed A/D channels and the sweeps contain 500 samples for each channel, the D/A converter generates 1500 samples. Thus there is a stimulus waveform sample corresponding to every sample in the de-multiplexed A/D waveform.

The amplitudes and durations of the steps, ramps and digital (i.e. TTL) pulses comprising the epochs can be automatically **incremented** from sweep to sweep (see the Epoch Waveform and Pulses section of the ABF header). Alternatively, instead of creating epoch-based waveforms, the user can choose to read the stimulus waveform from a file. Whichever method is used, a full array containing the stimulus waveform is provided by the ABF routines when the applications programmer requests the stimulus waveform array associated with any sweep.

During epochs, the **sampling interval** can be set to “Fast” or “Slow”. The Fast rate (fADCSequenceInterval) is the actual sampling rate of the digitizer, whereas the “Slow” rate uses decimation (uFileCompressionRatio) to reduce the number of samples saved in the file. If the applications programmer requests the X (i.e. time) array for the sweep, the ABF reading routines provide an array that contains the properly spaced time intervals taking into account the Fast and Slow sampling intervals. In ABF version 2.0, irrespective of whether the acquisition program specifies the sampling interval on a per-channel or a multiplexed basis, the value stored in the ABF file is the per-channel value. In the three channel example used previously, if each channel were sampled at 21 µs, the value stored in the file is 21 µs, even though the multiplexed sampling interval used by the digitizer is 7 µs. This is different from earlier versions (1.x) of ABF.

ABF episodic stimulation data files may also contain a special pseudo channel known as the **Math Signal**. This channel is the result of an arithmetic manipulation of two acquired data channels. In actual fact, the math signal data are not stored in the file. Instead, the formula and the acquired data channels are stored. However, as a practical matter the applications programmer need not know that the math signal data are not stored, since if the math signal is requested, the ABF routines calculate the result and return the math signal array. On the other hand, for more flexible analysis purposes, the applications programmer can take advantage of the fact that the math signal is created on the fly during reading by altering the parameters of the formula before requesting the math signal array.

A correction technique called **P/N leak subtraction** can be applied to one selected ADC channel during acquisition. This sophisticated technique is specific to intracellular voltage-clamp measurements. Using this technique, passive cell membrane responses are removed from the signal on the selected ADC channel before storage. Although P/N leak subtraction is an important acquisition technique, it does not directly affect data analysis, because from the data handling and storage perspective, the P/N leak subtracted ADC channel is not different than the other ADC channels.

In Clampex software , both the raw and the corrected (P/N leak subtracted) data are stored; this allows later analysis on either the raw or corrected data.

Another acquisition technique that does not directly affect data analysis is the application of pre-sweep trains. These are trains of pulses that are applied to condition the cell membrane before the sweep commences. No data are stored during the pre-sweep trains.

Many of the parameters of an acquisition can be arbitrarily specified for each sweep by a comma-separated list of variables. These are stored in the **Variable Parameter User List**. There is one user list for each output channel. A user list (sParamValueList) can only be applied to a single selected parameter (nParamToVary). Analysis programs should consider reading and parsing the user list since it is sometimes useful to plot extracted results in an X-Y plot with the user list values determining the X axis. When a user list is enabled (nListEnable) it overrides the usual specification for the selected parameter.

Source Code

The files included in this package provide Microsoft Windows libraries for accessing data files stored in the Molecular Devices ABF file format.

The source code is no longer included in the File Support Pack. The Windows dynamic linked library ABFFIO.DLL along with the included ‘C’ header files must be used to access the data.

The File Support Pack consists of a .ZIP file (ABF_FileSupportPack.zip). Run WinZIP to unzip the files.

The ABFFIO folder contains the DLL and the required ‘C’ header files.

COPYRIGHT

These libraries are copyrighted by Molecular Devices, LLC.

Molecular Devices permits the use of these libraries for the addition of file I/O support to third-party programs. Modified libraries retain their original copyright.

FUTURE COMPATIBILITY

From time to time the various Axon file formats will be enhanced. It is the intention of Molecular Devices to update the Axon File Support Pack soon after new file formats are released.

Obtaining Support

Molecular Devices is a leading worldwide manufacturer and distributor of analytical instrumentation, software, and reagents. We are committed to the quality of our products and to fully supporting our customers with the highest level of technical service.

Our Support website, www.moleculardevices.com/service-support, has a link to the Knowledge Base, which contains technical notes, software upgrades, safety data sheets, and other resources. If you still need assistance after consulting the Knowledge Base, you can submit a request to Molecular Devices Technical Support.

Please have your instrument serial number or Work Order number and your software version number available when you call.

Chapter 2: The ABF Header

The ABF header is the first block of data at the start of an ABF data file. The header contains parameters that describe the stimulation, the acquisition and the hierarchy of the data. It describes the contents of the data file and contains entries to describe the settings in effect when the data file was acquired.

In version 2.0, the header is of variable length. This depends on the protocol features in use (e.g. number of channels, number of epochs in the command waveform). Third party programs should NOT rely on the size of the header, or retrieve information directly from the file based on a byte offset. The ABFFileHeader is now different to the data written to the file - only use the documented variables defined in the file ABFFIleHeadr.h.

See the file ABFHEADR.H for a “C” definition of the ABFFileHeader structure.

ABFFileHeader sub-sections	
File ID and Size Information	
File Structure	
Trial Hierarchy Information	Application Version Information
Display Parameters	LTP Protocol
Hardware Information	Output Triggers
Environmental Information	Post-processing Actions
Multi-channel information	
Synchronous Timer Outputs	
Epoch Waveform and Pulses	
Stimulus Output File	
Pre-sweep Trains	
Variable Paramewter User List	
Statistics Measurements	
Channel Arithmetic	
Leak Subtraction	
Miscellaneous Parameters	

ADC Channel Numbering

The Axon data acquisition programs distinguish between physical and logical channel numbers. Physical channel numbers are the channel numbers used internally to communicate with the acquisition hardware. Logical channel numbers are the external connector labels on the front panel of the acquisition hardware. Logical channel numbers are used only for presentation to the user. Physical channel numbers are used everywhere else. For example, parameters are stored using physical channel number order (0 to 15) for such structures as the sampling sequence array and the entries for the external lowpass and highpass filters. Similarly, a physical channel number is used for the Trigger channel. Currently, the only digitizer known to have different physical and logical channel numbering is the obsolete TL-2 interface.

Indexing Arrays in the ABF Header

To get a Logical channel number from a Physical channel number, simply index the nADCPtoLchannelMap array by the Channel number you wish to convert. Thus nADCPtoLchannelMap[1] provides the Logical Channel Number for Physical Channel Number 1. This array is always symmetrical, so it can be used in the same way to convert back to Physical Channel Numbers from Logical Channel Numbers.

The first thing to look at is the nADCSamplingSeq array. This tells you which physical ADC channels were acquired and in what order. The first entry in this array is the Physical channel number of the first ADC channel acquired, followed by the second etc. There are nADCNumChannels channels in this array. All ADC arrays except for the nADCSamplingSeq are indexed through Physical channel numbers. These include: sADCChannelName, sADCUnits, etc.

Note: NOTE: All array indexing within the header and within the ABF routines start at 0, except Sweep number, which starts at 1.

```
void ShowFirstAcquiredChannelInfo( int nFile, ABFFileHeader *pFH )
{
       // Get first physical channel number and name
       int nFirstPhysicalChannel = pFH->nADCSamplingSeq[0];
       char *psSignalName = pFH->sADCChannelName[nFirstPhysicalChannel];
       // Get channel number to show to user
       int nFirstLogicalChannel = pFH->nADCPtoLchannelMap
   [nFirstPhysicalChannel];
       fprint("The first acquired channel (%s) comes from ADC channel %d\n",
           psSignalName, nFirstLogicalChannel);
       // Get start time of first episode of first channel
       int nFirstEpisode = 1;
       float fStartTime;
       ABF_GetStartTime(nFile, pFH, nFirstPhysicalChannel, nFirstEpisode,
           &fStartTime, NULL);
}
```

Unused Fields

Unused integer and floating point parameter fields should be filled with zeros. Unused strings should be filled with the space character (ASCII #32).

Parameters for unsampled ADC channels should be filled with the indicated default.

Version Numbers

The file version number consists of a major and a minor number. For example, the "1" in Version 1.0 is the major number, and the "0" is the minor number.

In general, the major version number is updated when there are many changes that can affect the byte offset of the existing parameters. The minor version number is updated when unused parameter space in the ABF structure is used. In most cases, existing programs will not be affected since they should not be dependent upon the unused parameters.

Chapter 3: The ABF File I/O Functions

The ABF file routines are a set of functions for creating and/or accessing ABF data files. Some functions are low level functions that will only be required by users acquiring ABF data files. Other functions provide higher level access to ABF data, returning fully scaled data values in the units of the acquired data.

Note: In version 2.0 of ABF, there is no longer a direct correspondence between the ABF File Header and the binary image of the file. Therefore it is essential that the ABF header structure is accessed through the published header files, NOT by byte offsets within the binary image of the file.

In addition the ABFH_xxx functions should be used to extract data from the header where available.

Routine	Use
ABF_BuildErrorText on page 49	Build an error string from an error number and a file name.
ABF_Close on page 26	Closes an ABF file that was previously opened with either ABF_ReadOpen or ABF_WriteOpen.
ABF_EpisodeFromSynchCount on page 51	Find the sweep that contains a particular synch count.
ABF_FormatDelta on page 43	Builds an ASCII string to describe a delta.
ABF_FormatTag on page 52	This function reads a tag from the TagArray section and formats it as ASCII text.
ABF_GetEpisodeDuration on page 52	Get the duration of a given sweep in ms.
ABF_GetEpisodeFileOffset on page 53	Returns the sample point offset in the ABF file for the start of the given sweep number that is passed as an argument.
ABF_GetFileHandle on page 61	Returns the DOS file handle associated with the specified file.
ABF_GetMissingSynchCount on page 53	Get the count of synch counts missing before the start of this sweep and the end of the previous sweep.
ABF_GetNumSamples on page 55	Get the number of samples in this sweep.
ABF_GetStartTime on page 56	Gets the start time in ms for the specified sweep.
ABF_GetSynchArray on page 61	Returns a pointer to the CSynch object used to buffer the Synch array to disk.
ABF_GetWaveform on page 28	Gets the Waveform that was put out for a particular sweep on a particular ADC channel in User Units.
ABF_GetVoiceTag on page 38	Retrieves a voice tag from the ABF file.

Routine	Use
ABF_HasData on page 57	Checks whether an open ABF file has any data in it.
WINAPI ABF_HasOverlappedData on page 57	Determines if there is any overlapped data in the file.
ABF_IsABFFile on page 58	Checks the data format of a given file.
ABF_MultiplexRead on page 31	Reads a sweep of multiplexed multi-channel ADC samples from the ABF file.
ABF_MultiplexWrite on page 33	Writes a sweep of multiplexed multi-channel ADC samples to the ABF file.
ABF_PlayVoiceTag on page 40	Retrieves a voice tag, builds a WAV file, plays the WAV file and cleans up.
ABF_ReadChannel on page 27	Reads a sweep/chunk of data from a particular ADC channel, returning the data as fully scaled User Units.
ABF_ReadDACFileEpi on page 34	Reads a sweep of multiplexed multi-channel DAC samples from the DACFile section of the ABF file. (Only valid if a DAC file was used for waveform generation.)
ABF_ReadTags on page 36	Reads a segment of the array from the TagArray section of the ABF file.
ABF_ReadOpen on page 21	Opens an ABF file for reading.
ABF_ReadRawChannel on page 35	Reads a complete multiplexed sweep from the data file and then decimates it, returning single de-multiplexed channel in the raw data format.
ABF_ReadScopeConfig on page 45	Retrieves the scope configuration info from the data file.
ABF_ReadTags on page 36	Reads a segment of the tag array from the TAGArray section.
ABF_SaveVoiceTag on page 39	Saves a voice tag to the ABF file.
ABF_SetErrorCallback on page 59	This routine sets a callback function to be called in the event of an error occurring.
ABF_SynchCountFromEpisode on page 59	Find the synch count at which a particular sweep started.
ABF_UpdateHeader on page 24	Updates the file header and writes the synch array out to disk if required.
ABF_UpdateAfterAcquisition on page 63	Update the ABF internal housekeeping after data has been written into a data file without using the ABF file I/O routines.

Routine	Use
ABF_WriteDACFileEpi on page 47	Writes a sweep of multiplexed multi-channel DAC samples to the DACFile section of the ABF file. This function should only be used after all acquired data has been written to the file.
ABF_WriteDelta on page 43	Writes the details of a delta to a temporary file. The deltas are written to the ABF file by ABF_Update.
ABF_WriteOpen on page 23	Opens an ABF file for writing.
ABF_WriteRawData on page 48	Writes a raw data buffer to the ABF file at the current file position.
ABF_WriteScopeConfig on page 45	Saves the current scope configuration info to the data file.
ABF_WriteStatisticsConfig on page 46	Saves the current statistics window configuration info to the data file.
ABF_WriteTag on page 37	Writes a tag value to the TAGArray section.

Notes:

Error Return Values on page 21

The ABF File I/O Functions by category

Notes About ABF File I/O Functions

Altering Existing Raw Data Files

Molecular Devices does not easily allow users to change or append data to ABF raw data files, in the belief that raw data is sacrosanct and will often need to be analyzed many times in the future. We recommend that third-party developers do not allow users to easily delete or modify ABF files.

Compilers

The ABF File Support Libraries routines are written in C++. For pCLAMP 10, it is built using the Microsoft Visual C++ version 7.0 compiler (Visual Studio .NET 2003).

Error Return Values

The return type for all ABF API functions is “BOOL”. The interpretation of this value is that TRUE = Success, and FALSE = Failure of the function. Should a function call fail, an error number indicating the reason for failure is returned in the pnError parameter. If the reason for the error is not required, NULL may be passed for the pnError parameter.

File Open/Close

The ABF API functions provides two functions for opening files, one for opening files for reading, the other for opening files for writing. Files opened for writing may not be read from, and files opened for reading may no be written to. The ABF_Close function must always be called to close a file successfully opened with either ABF_ReadOpen or ABF_WriteOpen.

Routine	Use
ABF_ReadOpen on page 21	Opens an ABF file for reading.
ABF_WriteOpen on page 23	Opens an ABF file for writing.
ABF_UpdateHeader on page 24	Updates the file header and writes the synch array out to disk if required.This routine should always be called before closing a file opened with ABF_WriteOpen.
ABF_Close on page 26	Closes an ABF file that was previously opened with either ABF_ReadOpen or ABF_WriteOpen.

ABF_ReadOpen

#include "abffiles.h"

BOOL ABF_ReadOpen(char **szFileName*, **int** **phFile*, **UINT** *uFlags*,

ABFFileHeader **pFH*, **UINT** **puMaxSamples*,
DWORD **pdwMaxEpi*, **int** **pnError*);

Opens an existing ABF data file for reading. Reads the acquisition parameters from the file header into the passed ABFFileHeader structure.

Parameter	Description
szFileName	Name of data file to open.
phFile	Pointer to ABF file handle of this file.
uFlags	Flag to indicate whether file is parameter file or not.
pFH	Pointer to acquisition parameters read from data file.
puMaxSamples	Pointer to requested size of data blocks to be returned.
pdwMaxEpi	Pointer to number of sweeps that exist in the data file.
pnError	Address of error return code. May be NULL.

Legal values for uFlags	
ABF_DATAFILE	File is data file.
ABF_PARAMFILE	File is parameter file.

Legal values for uFlags	
ABF_ALLOWOVERLAP	Permit return of overlapping data.

Comments

The **ABF_ReadOpen** function opens the data file *szFileName*, allocates an ABF file handle for it and assigns this number to **phFile*. Data is read from the file header into **pFH*. If ABF_PARAMFILE is set in *uFlags* then no further processing is performed, otherwise internal buffers are allocated in preparation for file reading.

For ABF_GAPFREEFILE and ABF_VARLENEVENTS files, **puMaxSamples* is passed in as a requested maximum size of the blocks of data returned by the ABF_ReadMultiplex and ABF_ReadChannel routines. For all modes, the actual value that will be used is returned in this location.

For Event Detected modes, on calling ABF_ReadOpen, the parameter *pdwMaxEpi* points to the maximum number of sweeps to read from the file. If it is zero the maximum will be 8192 sweeps, depending on RAM availability. The total number of data blocks of the size returned in **puMaxSamples* is returned in **pdwMaxEpi*.

Possible Error Codes

One of the following error codes may be returned on error (defined in ABFFILES.H).

Constant	Meaning
ABF_TOOMANYFILESOPEN	Too many files are already open.
ABF_EOPENFILE	Failed DOS open file.
ABF_EUNKNOWNFILETYPE	Could not recognise file type, possibly not an ABF file.
ABF_EBADPARAMETERS	Could not read parameter header, possibly corrupted header.
ABF_EEPISODESIZE	*pdwMaxSamples out of range i.e. below 128.
ABF_OUTOFMEMORY	Could not allocate internal buffer.

Example

```
#include "abffiles.h"
BOOL FindAnEpisode( char *pszFileName, DWORD *pdwSample, DWORD *pdwEpisode )
{
    int hFile;
    int nError = 0;
    ABFFileHeader FH;

    DWORD dwMaxEpi = 0;
    UINT uMaxSamples = 16 * 1024;

    if (!ABF_ReadOpen( pszFileName, &hFile, ABF_DATAFILE, &FH, &uMaxSamples,
    &dwMaxEpi, &nError ))
        return ShowABFError(pszFileName, nError);
    if (!ABF_EpisodeFromSynchCount( hFile, &FH, pdwSynchCount, pdwEpisode,
    &nError ))
```

```
    {
            ABF_Close( hFile, NULL );
            return ShowABFError(pszFileName, nError);
    }
    if (!ABF_Close( hFile, &nError ))
        return ShowABFError(pszFileName, nError);
    return TRUE;
}
```

ABF_WriteOpen

#include "abffiles.h"

BOOL ABF_WriteOpen(char **szFileName*, **int** **phFile*, **UINT** *uFlags*,

ABFFileHeader **pFH*, **int** **pnError*);

Opens an existing data fileFile for writing. Writes the acquisition parameters.

Parameter	Description
szFileName	Name of data file to open.
phFile	Pointer to ABF file handle of this file.
uFlags	Flag to indicate whether file is parameter file or not.
pFH	Pointer to acquisition parameters to be written to data file.
pnError	Address of error return code. May be NULL.

Comments

The **ABF_WriteOpen** function opens the data file *szFileName*, allocates an ABF file handle for it and assigns this number to **phFile*. The contents of **pFH* are written to the file header.

Possible Error Codes

One of the following error codes may be returned on error (defined in ABFFILES.H).

Constant	Meaning
ABF_TOOMANYFILESOPEN	Too many data files are already open.
ABF_EOPENFILE	Failed DOS open file.
ABF_EWRITEPARAMETERS	Could not write parameter header.
ABF_OUTOFMEMORY	Could not allocate decollation buffer.
ABF_EDISKFULL	Not enough space on disk.

Example

```
#include "abffiles.h"
BOOL Acquisition( char *pszFileName, ABFFileHeader *pFH )
{
    int hFile;
    HANDLE hHandle;
    int nError = 0;
```

```
    DWORD dwEpisodes, dwSamples;

    if (!ABF_WriteOpen( pszFileName, &hFile, ABF_DATAFILE, pFH, &nError ) )
        return ShowABFError(pszFileName, nError);
    if (!ABF_GetFileHandle( hFile, &hHandle, &nError ))
    {
        ABF_Close( hFile, NULL );
        return ShowABFError(pszFileName, nError);
    }
    AcquireAndWriteData( hHandle, pFH, &dwEpisodes, &dwSamples );
    if (!ABF_UpdateAfterAcquisition( hFile, pFH, dwEpisodes, dwSamples, &nError
    ))
    {
        ABF_Close( hFile, NULL );
        return ShowABFError(pszFileName, nError);
    }
    if (!ABF_UpdateHeader( hFile, pFH, &nError ))
    {
        ABF_Close( hFile, NULL );
        return ShowABFError(pszFileName, nError);
    }
    if (!ABF_Close( hFile, &nError ))
        return ShowABFError(pszFileName, nError);
    return TRUE;
}
```

ABF_UpdateHeader

#include "abffiles.h"

BOOL ABF_UpdateHeader(int *hFile*, **ABFFileHeader** **pFH*, **int** **pnError*);

Updates the file header to reflect the data newly written into an ABF data file.

Parameter	Description
hFile	ABF file handle.
pFH	Pointer to acquisition parameters.
pnError	Address of error return code. May be NULL.

Comments

The **ABF_UpdateHeader** function updates the file header and writes the synch array out to disk if required. This function should always be called before closing a file opened with ABF_WriteOpen.

Possible Error Codes

One of the following error codes may be returned on error (defined in ABFFILES.H).

Constant	Meaning
ABF_EWRITEPARAMETERS	Could not write header parameters.

Example

```
#include "abffiles.h"
BOOL Acquisition( char *pszFileName, ABFFileHeader *pFH )
{
    int hFile;
    HANDLE hHandle;
    int nError = 0;
    DWORD dwEpisodes, dwSamples;
    if (!ABF_WriteOpen( pszFileName, &hFile, ABF_DATAFILE, pFH, &nError ) )
        return ShowABFError(pszFileName, nError);
    if (!ABF_GetFileHandle( hFile, &hHandle, &nError ))
    {
        ABF_Close( hFile, NULL );
        return ShowABFError(pszFileName, nError);
    }
    AcquireAndWriteData( hHandle, pFH, &dwEpisodes, &dwSamples );
    if (!ABF_UpdateAfterAcquisition( hFile, pFH, dwEpisodes, dwSamples, &nError
    ))
    {
        ABF_Close( hFile, NULL );
        return ShowABFError(pszFileName, nError);
    }
    if (!ABF_UpdateHeader( hFile, pFH, &nError ))
    {
        ABF_Close( hFile, NULL );
        return ShowABFError(pszFileName, nError);
    }
    if (!ABF_Close( hFile, &nError ))
        return ShowABFError(pszFileName, nError);
    return TRUE;
}
```

ABF_Close

#include "abffiles.h"

BOOL ABF_Close(int *hFile*, **int** **pnError*);

Closes the specified data file.

Parameter	Description
hFile	ABF file handle.
pnError	Address of error return code. May be NULL.

Comments

The **ABF_Close** function closes the data file specified in *hFile*.

Possible Error Codes

One of the following error codes may be returned on error (defined in ABFFILES.H).

Constant	Meaning
ABF_EBADFILEINDEX	Invalid ABF file handle specified.
ABF_EBADFILE	Could not close file.

Example

```
#include "abffiles.h"
int ReadChannelEpisode( char *pszFileName, int nChannel,
              DWORD dwEpisode, float *pfBuffer,
              UINT *puNumSamples )
{
    int hFile;
    int nError;
    ABFFileHeader FH;
    DWORD dwMaxEpi = 0;
    UINT uMaxSamples = 16 * 1024;
    if (!ABF_ReadOpen(pszFileName, &hFile, ABF_DATAFILE, &FH,
            &uMaxSamples, &dwMaxEpi, &nError))
        return ShowABFError(pszFileName, nError);

    if (!ABF_ReadChannel( hFile, &FH, nChannel, dwEpisode, pfBuffer,
            puNumSamples, &nError ))
    {
        ABF_Close( hFile, NULL );
        return ShowABFError(pszFileName, nError);
    }
    if (!ABF_Close( hFile, &nError ))
        return ShowABFError(pszFileName, nError);
    return TRUE;
}
```

High Level File Reading

The high level file reading routines return data from the ABF file in fully scaled 4-byte floats, in the units specified by the user (User Units on page 69) at the preparation.

Routine	Use
ABF_ReadChannel on page 27	Reads a sweep/chunk of data from a particular ADC channel, returning the data as fully scaled UserUnits.
ABF_GetWaveform on page 28	Gets the Waveform that was put out for a particular sweep on a particular DAC channel in UserUnits.

ABF_ReadChannel

#include "abffiles.h"

BOOL ABF_ReadChannel(**int** *hFile*, **ABFFileHeader** *pFH,

int *nChannel*, **DWORD** *dwEpisode*, **float** **pfBuffer*,
UINT **puNumSamples*, **int** **pnError*);

Reads a sweep of data for a particular channel from a previously opened data file.

Parameter	Description
hFile	ABF file handle.
pFH	File header for the file being read.
nChannel	Physical channel number to be read.
dwEpisode	Sweep number to be read.
pfBuffer	Data buffer for the data.
puNumSamples	Number of valid points in the data buffer.
pnError	Address of error return code. May be NULL.

Comments

The **ABF_ReadChannel** function reads sweep number *dwEpisode* of channel *nChannel* from *hFile* into *pfBuffer*. The actual number of points read into the buffer is returned in **puNumSamples*. If the data in the file is in two-byte binary format, it is converted into fully scaled 4-byte floats in User Units on page 69.

It is up to the user of this routine to ensure that the buffer passed in as pfBuffer points to an array of sufficient size to contain the returned sweep.

Possible Error Codes

One of the following error codes may be returned on error (defined in ABFFILES.H).

Constant	Meaning
ABF_EBADFILEINDEX	Invalid ABF file handle specified.
ABF_EWRITEONLYFILE	This file is write-only.
ABF_EINVALIDCHANNEL	Channel number is invalid.

Example

```
#include "abffiles.h"
int ReadChannelEpisode( char *pszFileName, int nChannel,
          DWORD dwEpisode, float *pfBuffer,
          UINT *puNumSamples )
{
    int hFile;
    int nError;
    ABFFileHeader FH;

    DWORD dwMaxEpi = 0;
    UINT uMaxSamples = 16 * 1024;
    if (!ABF_ReadOpen(pszFileName, &hFile, ABF_DATAFILE, &FH, &uMaxSamples,
    &dwMaxEpi, &nError))
        return ShowABFError(pszFileName, nError);
    if (!ABF_ReadChannel( hFile, &FH, nChannel, dwEpisode, pfBuffer,
        puNumSamples, &nError ))
    {
        ABF_Close( hFile, NULL );
        return ShowABFError(pszFileName, nError);
    }
    if (!ABF_Close( hFile, &nError ))
        return ShowABFError(pszFileName, nError);
    return TRUE;
}
```

ABF_GetWaveform

#include "abffiles.h"

BOOL ABF_GetWaveform(int *nFile*, **ABFFileHeader** *pFH, **int** *nChannel*,

DWORD *dwEpisode*, **float** **pfBuffer*, **int** **pnError*);

Gets the DAC output waveform for the specified sweep.

Parameter	Description
hFile	ABF file handle.
pFH	File header for the file as returned by ABF_ReadOpen.
nChannel	DAC channel of interest.
dwEpisode	Sweep number to return the start time for.
pfBuffer	Address of buffer to fill with DAC output waveform.
pnError	Address of error return code. May be NULL.

Comments

The ABF_GetWaveform function returns the DAC output waveform for a particular sweep, in DAC User Units.

Possible Error Codes

One of the following error codes may be returned on error (defined in ABFFILES.H).

Constant	Meaning
ABF_EEPISODERANGE	Sweep number out of range.
ABF_EBADFILEINDEX	Invalid ABF file handle specified.

Example

```
#include "abffiles.h"
BOOL ShowWaveforms(char *pszFileName, int nFile,
          ABFFileHeader *pFH, int nChannel)
{
    int nError;
    DWORD I;
    UINT uNumSamples = (UINT)pFH->lNumSamplesPerEpisode /
          pFH->nADCNumChannels;
    float *pfBuffer = (float *)malloc(uNumSamples *
              sizeof(float));
    if (!pfBuffer)
    {
        printf("Out of memory!\n");
        return FALSE;
    }
    for (DWORD i=1; i<=(DWORD)pFH->lActualEpisodes; I++)
    {
        if (!ABF_GetWaveform(nFile, pFH, nChannel, i, pfBuffer,
              &nError))
    {
        free(pfBuffer);
        return ShowABFError(pszFileName, nError);
    }
    printf("Episode %lu\n", i);
    for (UINT j=0; j<uNumSamples; j++)
        printf("%g\n", pfBuffer[j]);
    }
    free(pfBuffer);
    return TRUE;
}
```

Low Level File Read/Write

The low level file I/O routines read and write raw data in two-byte ADC/DAC samples.

Note: Note: To avoid the complexity of doing the ADC to User Units conversion, Molecular Devices strongly recommends that third-party developers use the high-level file reading routines instead of the following low-level routines.

If the low-level routines are used, the functions ABFH_GetADCtoUUFactors() and ABFH_GetDACtoUUFactors() should be used to retrieve the composite scale and offset factors used to convert ADC/DAC values to UserUnits.

Routine	Use
ABF_MultiplexRead on page 31	Reads a sweep of multiplexed multi-channel ADC samples from the ABF file.
ABF_MultiplexWrite on page 33	Writes a sweep of multiplexed multi-channel ADC samples to the ABF file.
ABF_ReadDACFileEpi on page 34	Reads a sweep of multiplexed multi-channel DAC samples from the DACFile section of the ABF file (only valid if a DAC file was used for waveform generation).
ABF_ReadRawChannel on page 35	Reads a complete multiplexed sweep from the data file and then decimates it, returning single de-multiplexed channel in the raw data format.
ABF_ReadTags on page 36	Reads a segment of the tag array from the TAGArray section.
ABF_WriteTag on page 37	Writes a tag value to the TAGArray section.
ABF_GetVoiceTag on page 38	Retrieves a voice tag from the ABF file.
ABF_SaveVoiceTag on page 39	Saves a voice tag to the ABF file.
ABF_PlayVoiceTag on page 40	Retrieves a voice tag, builds a WAV file, plays the WAV file and cleans up.
ABF_ReadDeltas on page 41	Reads a Delta array from the DeltaArray section of the ABF file.
ABF_WriteDelta on page 43	Writes the details af a delta to a temporary file. The deltas are written to the ABF file by ABF_Update.
ABF_FormatDelta on page 43	Builds an ASCII string to describe a delta.
ABF_ReadScopeConfig on page 45	Retrieves the scope configuration info from the data file.
ABF_WriteScopeConfig on page 45	Saves the current scope configuration info to the data file.

Routine	Use
ABF_WriteStatisticsConfig on page 46	Saves the current statistics window configuration info to the data file.
ABF_WriteDACFileEpi on page 47	Writes a sweep of multiplexed multi-channel DAC samples to the DACFile section of the ABF file. This function should only be used after all acquired data has been written to the file.
ABF_WriteRawData on page 48	Writes a raw data buffer to the ABF file at the current file position.

ABF_MultiplexRead

#include "abffiles.h"

BOOL ABF_MultiplexRead(int *hFile*, **ABFFileHeader** *pFH*,

DWORD *dwEpisode*, **void** **pvBuffer*, **UINT** **puNumSamples*,
int **pnError*);

Reads a sweep of data from a previously opened data file. The data is returned with all channels multiplexed together.

Parameter	Description
hFile	ABF file handle.
pFH	File header for the file being read.
dwEpisode	Sweep number to be read.
pvBuffer	Data buffer for the data.
puNumSamples	Number of valid points returned in the data buffer.
pnError	Address of error return code. May be NULL.

Comments

The **ABF_MultiplexRead** function reads sweep number *dwEpisode* from *hFile* into *pvBuffer*. The actual number of points read into the buffer is returned in **puNumSamples*. Only in the case of ABF_VARLENEVENTS mode or at the end of an ABF_GAPFREEFILE file will **puNumSamples* differ from the value returned by ABF_ReadOpen in **puMaxSamples*.

It is up to the user of this routine to ensure that the buffer passed in as pvBuffer points to an array of at least pFH->lNumSamplesPerEpisode samples in length, where the file header pFH was returned by the ABF_ReadOpen command.

Possible Error Codes

One of the following error codes may be returned on error (defined in ABFFILES.H).

Constant	Meaning
ABF_EEPISODERANGE	Sweep number out of range.
ABF_EREADDATA	Could not read sweep data from file.
ABF_EBADFILEINDEX	Invalid ABF file handle specified.

Example

```
#include "abffiles.h"
BOOL CopyDataFile(char *pszFileIn, int nFileIn, ABFFileHeader *pFI,
          char *pszFileOut, int nFileOut, ABFFileHeader *pFO)
{
    UINT uNumSamples = (UINT)pFI->lNumSamplesPerEpisode;
    DWORD dwEpiStart, dwMissingSamples;

    short *pnBuffer = (short *)malloc(uNumSamples * sizeof(short));
    if (!pnBuffer)
    {
        printf("Out of memory!\n");
        return FALSE;
    }
    for (DWORD i=1; i<=(DWORD)pFI->lActualEpisodes; i++)
    {
    UINT uFlag = 0;
    int nError = 0;
    if (!ABF_MultiplexRead( nFileIn, pFI, i, pnBuffer, &uNumSamples, &nError ))
        return ShowABFError(pszFileIn, nError);
    if (!ABF_SynchCountFromEpisode( nFileIn, pFI, i, &dwEpiStart, &nError ))
        return ShowABFError(pszFileIn, nError);
    if (pFI->nOperationMode == ABF_VARLENEVENTS)
    {
        if (!ABF_GetMissingSynchCount( nFileIn, pFI, I, &dwMissingSynchCount,
        &nError ))
        return ShowABFError(pszFileIn, nError);
        if (dwMissingSynchCount == 0)
            uFlag = ABF_APPEND;
    }
    if (!ABF_MultiplexWrite( nFileOut, pFO, uFlag, pnBuffer, dwEpiStart,
    uNumSamples, &nError ))
        return ShowABFError(pszFileOut, nError);
    }
    return TRUE;
}
```

ABF_MultiplexWrite

#include "abffiles.h"

BOOL ABF_MultiplexWrite(int *hFile*, **ABFFileHeader** *pFH,

UINT *uFlags*, **void** **pvBuffer*, **DWORD** *dwEpiStart*,
UINT *uNumSamples*, **int** **pnError*);

Writes a sweep of data into a previously opened data file. The data buffer must contain all channels multiplexed together.

Parameter	Description
hFile	ABF file handle.
pFH	File header for the file being written.
uFlags	Flags governing the write process.
pvBuffer	Data buffer for the data.
dwEpiStart	Start time in samples of this sweepSweep.
uNumSamples	Number of valid points in the data buffer.
pnError	Address of error return code. May be NULL.

Comments

The **ABF_MultiplexWrite** function writes the sweep of data from *pvBuffer* into *hFile*. If the ABF_APPEND flag is set for an ABF_VARLENEVENTS mode file the data is appended to the previous sweep in the data file being written.

Possible Error Codes

One of the following error codes may be returned on error (defined in ABFFILES.H).

Constant	Meaning
ABF_EDISKFULL	Not enough space on disk.

Example

```
#include "abffiles.h"
BOOL CopyDataFile(char *pszFileIn, int nFileIn, ABFFileHeader *pFI,
          char *pszFileOut, int nFileOut, ABFFileHeader *pFO)
{
    UINT uNumSamples = (UINT)pFI->lNumSamplesPerEpisode;
    DWORD dwEpiStart, dwMissingSamples;

    short *pnBuffer = (short *)malloc(uNumSamples * sizeof(short));
    if (!pnBuffer)
    {
        printf("Out of memory!\n");
        return FALSE;
    }
```

```
    for (DWORD i=1; i<=(DWORD)pFI->lActualEpisodes; i++)
    {
        UINT uFlag = 0;
        int nError = 0;
        if (!ABF_MultiplexRead( nFileIn, pFI, i, pnBuffer, &uNumSamples,
                &nError ))
            return ShowABFError(pszFileIn, nError);

        if (!ABF_SynchCountFromEpisode( nFileIn, pFI, i, &dwEpiStart,
                &nError ))
            return ShowABFError(pszFileIn, nError);
        if (pFI->nOperationMode == ABF_VARLENEVENTS)
        {
            if (!ABF_GetMissingSynchCount( nFileIn, pFI, I,
                &dwMissingSynchCount, &nError ))
                   return ShowABFError(pszFileIn, nError);
            if (dwMissingSynchCount == 0)
                uFlag = ABF_APPEND;
        }
    if (!ABF_MultiplexWrite( nFileOut, pFO, uFlag, pnBuffer,
                dwEpiStart, uNumSamples, &nError ))
        return ShowABFError(pszFileOut, nError);
    }
    return TRUE;
}
```

ABF_ReadDACFileEpi

#include "abffiles.h"

BOOL ABF_ReadDACFileEpi(int *hFile*, ABFFileHeader **pFH*,

short **pnDACArray*, **DWORD** *dwEpisode*, **int** **pnError*);

Reads a sweep from the DAC file section of an ABF file.

Parameter	Description
hFile	ABF file handle.
pFH	Pointer to acquisition parameters.
pnDACArray	Data buffer for the data.
dwEpisode	Sweep number to be read.
pnError	Address of error return code. May be NULL.

Comments

The **ABF_ReadDACFileEpi** function reads sweep number *dwEpisode* from the DAC file section of *hFile* into *pnDACArray*..

Possible Error Codes

One of the following error codes may be returned on error (defined in ABFFILES.H).

Constant	Meaning
ABF_EREADDACEPISODE	Could not read data.

Example

```
#include "abffiles.h"
BOOL ShowDACFileData(char *pszFileName, int nFile, ABFFileHeader *pFH,
              short *pnBuffer)
{
    int nError;
    DWORD i;
    UINT j;
    UINT uNumSamples = (UINT)pFH->lNumSamplesPerEpisode;

    for (i = 0; i < (DWORD)pFH->lDACFileNumEpisodes; i++)
    {
        if (!ABF_ReadDACFileEpi( nFile, pFH, pnBuffer, i, &nError ))
            return ShowABFError(pszFileName, nError);
        for (j = 0; j < uNumSamples; j++)
            printf( "%d\n", pnBuffer[j] );
    }
    return TRUE;
}
```

ABF_ReadRawChannel

#include "abffiles.h"

BOOL ABF_ReadRawChannel(int *nFile*, **ABFFileHeader** **pFH*, **int** *nChannel*, **DWORD** *dwEpisode*,

void **pvBuffer*, **UINT** **puNumSamples*, **int** **pnError*);

Reads a complete multiplexed sweep from the data file and then decimates it, returning single de-multiplexed channel in the raw data format.

Parameter	Description
hFile	ABF file handle.
pFH	Pointer to acquisition parameters.
nChannel	Channel to read the data for.
dwEpisode	Sweep/chunk number to read.
pvBuffer	Buffer to return the raw, de-multiplexed data.
puNumSamples	Size of buffer pointed to by *pvBuffer*.
pnError	Address of error return code. May be NULL.

Comments

The required size of the passed buffer is:

pFH->lNumSamplesPerEpisode / pFH->nADCNumChannels (shorts)

Possible Error Codes

One of the following error codes may be returned on error (defined in ABFFILES.H).

Constant	Meaning
ABF_EINVALIDCHANNEL	The requested channel was not in the sampling list.
ABF_OUTOFMEMORY	Insufficient memory was available for use internally.
ABF_EEPISODERANGE	Sweep number out of range.
ABF_EREADDATA	Could not read sweepSweep data from file.
ABF_EBADFILEINDEX	Invalid ABF file handle specified.

Example

```
#include "abffiles.h"
```

ABF_ReadTags

#include "abffiles.h"

BOOL ABF_ReadTags(**int** *hFile*, **ABFFileHeader** **pFH*,

DWORD *dwFirstTag*, **ABFTag** **pTagArray*, **UINT** *uNumTags*,
int **pnError*);

Reads a segment of the tag array from the TagArray section of the ABF file.

Parameter	Description
hFile	ABF file handle.
pFH	Pointer to acquisition parameters.
dwFirstTag	Index of the start of the sub array to retrieve
pTagArray	Data buffer for the tag array.
uNumTags	Number of tag entries to retrieve.
pnError	Address of error return code. May be NULL.

Comments

The **ABF_ReadTags** function reads a tag array from the TagArray section of the ABF file.

Possible Error Codes

One of the following error codes may be returned on error (defined in ABFFILES.H).

Constant	Meaning
ABF_EREADTAG	Could not read data.

Example

```
#include "abffiles.h"
```

```
#define TAG_BLOCKSIZE  10
int PrintTags( int hFile, char *pszFileName, ABFFileHeader *pFH )
{
    ABFTag *pTagArray;
    UINT i;
    int nError;

    if (pFH->lNumTagEntries < 1)
        return TRUE;
    pTagArray = (ABFTag *)calloc( TAG_BLOCKSIZE, sizeof(ABFTag) );
    DWORD dwTagCount = pFH->lNumTagEntries;
    DWORD dwFirstTag = 0;
    while (dwTagCount)
    {
         UINT uTags = (TAG_BLOCKSIZE > dwTagCount ?
                (UINT)dwTagCount : TAG_BLOCKSIZE);
        if (!ABF_ReadTags( hFile, pFH, dwFirstTag, pTagArray, uTags,
                &nError ))
        {
            free(pTagArray);
            return ShowABFError(pszFileName, nError);
        }
        for (i = 0; i < uTags; i++)
           printf( "\nTime: %ld Type: %d\n%56.56s\n",
            pTagArray[i].lTagTime, pTagArray[i].nTagType,
            pTagArray[i].sComment );
        dwTagCount -= uTags;
        dwFirstTag += uTags;
    }
    free(pTagArray);
    return TRUE;
}
```

ABF_WriteTag

#include "abffiles.h"

BOOL ABF_WriteTag(**int** *hFile*, **ABFFileHeader** **pFH*, **ABFTag** **pTag*,

int **pnError*);

Writes a tag value to the TAGArray section.

Parameter	Description
hFile	ABF file handle.
pFH	Pointer to acquisition parameters.

Parameter	Description
pTag	Data buffer of the tag array.
pnError	Address of error return code. May be NULL.

Comments

The ABF_WriteTag function writes a single ABFTag structure to the ABF file. All tags are internally buffered to disk inside the ABFFILES module and written out to the file when ABF_UpdateHeader() is called.

Possible Error Codes

One of the following error codes may be returned on error (defined in ABFFILES.H).

Constant	Meaning
ABF_EWRITETAG	Could not write data.

ABF_GetVoiceTag

BOOL ABF_GetVoiceTag(int nFile, **const ABFFileHeader** *pFH, **UINT** uTag, **LPCSTR** pszFileName,

long lDataOffset, **ABFVoiceTagInfo** *pVTI, **int** *pnError)

Retrieves a voice tag from the ABF file.

Parameter	Description
nFile	ABF file handle.
pFH	ABF file header.
uTag	Tag number.
pszFileName	File name of file to extract voice tag to.
lDataOffset	Position of voice tag in file .
pVTI	Voice Tag Info struct
pnError	Address of error return code. May be NULL.

Comments

The **ABF_GetVoiceTag** function retrieves a voice tag from the ABF file.

Possible Error Codes

One of the following error codes may be returned on error (defined in ABFFILES.H).

Constant	Meaning
ABF_EREADDATA	Error reading data from file.
ABF_EREADTAG	Error reading tag from file.

Example

```
#include "abffiles.h"
BOOL SaveVoiceTag( int nFile, ABFFileHeader *pFH, ABFTag *pTag )
{
```

```
    char szWAVFile[_MAX_PATH];
    if( !ABFU_GetTempFileName("wav", 0, szWAVFile) )
        return FALSE;
    // Extract the voice tag to the temp file.
    ABFVoiceTagInfo VTI;
    BOOL bReturn ABF_GetVoiceTag( nFile, pFH,
           pTag->nVoiceTagNumber, szWAVFile, 0, &VTI, NULL );
    if( !bReturn)
    {
        DeleteFile( szWAVFile );
        return FALSE;
    }
    // and save it to the pending list
    bReturn = ABF_SaveVoiceTag( m_hABFHandle, szWAVFile, 0, &VTI, NULL);
    if( !bReturn)
        DeleteFile( szWAVFile );

    return bReturn;
}
```

ABF_SaveVoiceTag

BOOL ABF_SaveVoiceTag(int nFile, **LPCSTR** pszFileName, long lDataOffset,
ABFVoiceTagInfo *pVTI, **int** *pnError);

Saves a voice tag to the ABF file.

Parameter	Description
hFile	ABF file handle.
pszFileName	File containing voice tag.
lDataOffset	Position of voice tag in file .
pVTI	Voice Tag Info struct
pnError	Address of error return code. May be NULL.

Comments

The **ABF_SaveVoiceTag** function saves a voice tag from a temporary file to the ABF file.

Possible Error Codes

One of the following error codes may be returned on error (defined in ABFFILES.H).

Constant	Meaning
ABF_OUTOFMEMORY	Could not allocate internal buffer.

Example

```
#include "abffiles.h"
BOOL SaveVoiceTag( int nFile, ABFFileHeader *pFH, ABFTag *pTag )
```

```
{
    char szWAVFile[_MAX_PATH];
    if( !ABFU_GetTempFileName("wav", 0, szWAVFile) )
        return FALSE;
    // Extract the voice tag to the temp file.
    ABFVoiceTagInfo VTI;
    BOOL bReturn ABF_GetVoiceTag( nFile, pFH,
                pTag->nVoiceTagNumber, szWAVFile, 0, &VTI, NULL );
    if( !bReturn)
    {
        DeleteFile( szWAVFile );
        return FALSE;
    }
    // and save it to the pending list
    bReturn = ABF_SaveVoiceTag( m_hABFHandle, szWAVFile, 0, &VTI, NULL);
    if( !bReturn)
        DeleteFile( szWAVFile );

    return bReturn;
}
```

ABF_PlayVoiceTag

BOOL ABF_PlayVoiceTag(int nFile, const **ABFFileHeader** *pFH, **UINT** uTag, **int** *pnError)

Retrieves a voice tag, builds a WAV file, plays the WAV file and cleans up.

Retrieves a voice tag from the ABF file.

Parameter	Description
nFile	ABF file handle.
pFH	ABF file header.
uTag	Tag number.
pnError	Address of error return code. May be NULL.

Comments

The **ABF_PlayVoiceTag** function retrieves a voice tag from the ABF file, builds a WAV file, plays the WAV file and cleans up.

Possible Error Codes

One of the following error codes may be returned on error (defined in ABFFILES.H).

Constant	Meaning
ABF_BADTEMPFILE	Error creating WAV file.

Example

```
#include "abffiles.h"
```

```
void ProcessVoiceTags(char *pszDataFile, ABFFileHeader *pFH)
{
    int   nFile;
    int   nErrorNum   = 0;
    UINT  uMaxSamples = 0;
    DWORD dwMaxEpi    = 0;

    if (!ABF_ReadOpen(pszDataFile, &nFile, ABF_DATAFILE, pFH,
              &uMaxSamples, &dwMaxEpi, &nErrorNum))
    {
        ShowABFError(nErrorNum, pszDataFile);
        return;
    }

    if ((pFH->lVoiceTagPtr == 0) || (pFH->lVoiceTagEntries == 0))
    {
        ABF_Close(nFile, NULL);
        Pause_printf( "Data file does not contain any voice tags.\n");
        return;
     }
    for (UINT i=0; i< UINT(pFH->lVoiceTagEntries); i++)
        if (!ABF_PlayVoiceTag( nFile, pFH, i, &nErrorNum))
            break;
    ABF_Close(nFile, NULL);
    if (nErrorNum)
        ShowABFError(nErrorNum, g_szDataFile);
}
```

ABF_ReadDeltas

BOOL ABF_ReadDeltas(int nFile, const **ABFFileHeader** *pFH, **DWORD** dwFirstDelta,
ABFDelta *pDeltaArray, **UINT** uNumDeltas, **int** *pnError)

This function reads a Delta array from the DeltaArray section of the ABF file.

Parameter	Description
nFile	ABF file handle.
pFH	ABF file Header.
dwFirstDelta	The first delta to read.
pDeltaArray	ABFDelta structure.
uNumDeltas	The number of deltas to read.
pnError	Address of error return code. May be NULL.

Comments

The **ABF_ReadDeltas** function reads a Delta array (pDeltaArray) from the DeltaArray section of the ABF file.

Possible Error Codes

One of the following error codes may be returned on error (defined in ABFFILES.H).

Constant	Meaning
ABF_EREADDELTA	Error reading delta from file
ABF_ENODELTAS	File does not contain any delta information.
ABF_EREADTAG	Error reading tag from file

Example

```
static void ShowDeltas(char *pszDataFile, ABFFileHeader *pFH)
{
    int    nFile;
    int    nErrorNum   = 0;
    UINT   uMaxSamples = 0;
    DWORD  dwMaxEpi    = 0;

    if (!ABF_ReadOpen(pszDataFile, &nFile, ABF_DATAFILE, pFH, &uMaxSamples,
           &dwMaxEpi, &nErrorNum))
    {
          ShowABFError(nErrorNum, g_szDataFile);
         return;
    }

    if ((pFH->lDeltaArrayPtr <= 4) || (pFH->lNumDeltas < 1))
    {
    ABF_Close(nFile, NULL);
    Pause_printf( "Data file does not contain any deltas.\n");
    return;
    }

    ABFDelta Delta;
    char szText[80];
    for (DWORD i=0; i<(DWORD)pFH->lNumDeltas; i++)
    {
        if (!ABF_ReadDeltas(nFile, pFH, i, &Delta, 1, &nErrorNum))
        {
    ABF_Close(nFile, NULL);
    ShowABFError(nErrorNum, g_szDataFile);
    return;
```

```
        }
    Pause_printf( "%7lu %8ld    ", i+1, Delta.lDeltaTime);
    if( ABF_FormatDelta( pFH, &Delta, &szText[0], sizeof(szText), &nErrorNum )
    )
        Pause_printf( " %s \n", szText);
    else
        {
    ABF_Close(nFile, NULL);
    ShowABFError(nErrorNum, g_szDataFile);
    return;
        }
    }
    ABF_Close(nFile, NULL);
}
```

ABF_WriteDelta

BOOL ABF_WriteDelta(int nFile, **ABFFileHeader** *pFH, **const ABFDelta** *pDelta, **int** *pnError)

Writes a delta (a parameter which is changed during a recording) to a temporary file.

Parameter	Description
nFile	ABF file handle.
pFH	ABF File Header.
pDelta	ABFDelta structure.
pnError	Address of error return code. May be NULL.

Comments

The **ABF_WriteDelta** function writes the details af a parameter which is changed during a recording, to a temporary file. The deltas are written to the ABF file by ABF_Update.

Possible Error Codes

One of the following error codes may be returned on error (defined in ABFFILES.H).

Constant	Meaning
ABF_EREADONLYFILE	The file is read only.

ABF_FormatDelta

BOOL ABF_FormatDelta(const ABFFileHeader *pFH, **const ABFDelta** *pDelta, **char** *pszText,

UINT uTextLen, **int** *pnError)

This function builds an ASCII string to describe a delta.

Parameter	Description
pFH	ABF File Header.
pDelta	ABFDelta structure.

Parameter	Description
pszText	The text buffer.
uTextLen	Length of the text buffer.
pnError	Address of error return code. May be NULL.

Comments

The **ABF_FormatDelta** function builds an ASCII string (pszText) to describe a delta (pDelta).

Possible Error Codes

One of the following error codes may be returned on error (defined in ABFFILES.H).

Constant	Meaning
ABF_EBADDELTAID	The Delta has an unknown parameter ID.

Example

```
static void ShowDeltas(char *pszDataFile, ABFFileHeader *pFH)
{
    int    nFile;
    int    nErrorNum   = 0;
    UINT   uMaxSamples = 0;
    DWORD  dwMaxEpi    = 0;

    if (!ABF_ReadOpen(pszDataFile, &nFile, ABF_DATAFILE, pFH, &uMaxSamples,
              &dwMaxEpi, &nErrorNum))
    {
        ShowABFError(nErrorNum, g_szDataFile);
        return;
     }

    if ((pFH->lDeltaArrayPtr <= 4) || (pFH->lNumDeltas < 1))
    {
        ABF_Close(nFile, NULL);
        Pause_printf( "Data file does not contain any deltas.\n");
        return;
    }

    ABFDelta Delta;
    char szText[80];

    for (DWORD i=0; i<(DWORD)pFH->lNumDeltas; i++)
    {
        if (!ABF_ReadDeltas(nFile, pFH, i, &Delta, 1, &nErrorNum))
        {
```

```
            ABF_Close(nFile, NULL);
            ShowABFError(nErrorNum, g_szDataFile);
            return;
        }
        Pause_printf( "%7lu %8ld   ", i+1, Delta.lDeltaTime);
        if( ABF_FormatDelta( pFH, &Delta, &szText[0], sizeof(szText), &nErrorNum
        ) )
            Pause_printf( " %s \n", szText);
        else
        {
            ABF_Close(nFile, NULL);
            ShowABFError(nErrorNum, g_szDataFile);
            return;
        }
    }
    ABF_Close(nFile, NULL);
}
```

ABF_ReadScopeConfig

BOOL ABF_ReadScopeConfig(int *nFile*, **ABFFileHeader** **pFH*, **ABFScopeConfig** **pCfg*,

UINT *uMaxScopes*, **int** **pnError*)

Retrieves the scope configuration info from the data file.

ABF_WriteScopeConfig

BOOL ABF_WriteScopeConfig(int nFile, **ABFFileHeader** *pFH, **int** nScopes,

ABFScopeConfig *pCfg, **int** *pnError)

Saves the current scope configuration info to the data file.

ABF_ReadStatisticsConfig

#include "abffiles.h"

BOOL ABF_WriteStatisticsConfig(int nFile, **ABFFileHeader** *pFH,

const ABFScopeConfig *pCfg, **int** *pnError);

Read the scope configuration structure for the statistics window from the ABF file.

Parameter	Description
nFile	ABF file handle.
pFH	ABFFileHeader.
pCfg	ABFScopeConfig.
pnError	Address of error return code. May be NULL.

Comments

The **ABF_WriteStatisticsConfig** function writes the ABFScopeConfig structure to the ABF file.

Possible Error Codes

One of the following error codes may be returned on error (defined in ABFFILES.H).

Constant	Meaning
ABF_ENOSTATISTICSCONFIG	The file has no statistics window infomation.
ABF_EREADSTATISTICSCONFIG	There was an error reading the statistics window configuration.

Example

```
#include "abffiles.h"
BOOL CopyStatsConfig( ABFFileHeader *pFI, ABFFileHeader *pFO )
{
    if (pFI.lStatisticsConfigPtr)
    {
        static ABFScopeConfig StatsCfg;
    if (!ABF_ReadStatisticsConfig( nFileIn, pFI, &StatsCfg, &nErrorNum))
        ErrorReturn( nErrorNum );
    if (!ABF_WriteStatisticsConfig( nFileOut, pFO, &StatsCfg, &nErrorNum))
        ErrorReturn( nErrorNum );
    }
    return TRUE;
}
```

ABF_WriteStatisticsConfig

#include "abffiles.h"

BOOL ABF_WriteStatisticsConfig(int nFile, **ABFFileHeader** *pFH,
const ABFScopeConfig *pCfg, int *pnError);

Write the scope config structure for the statistics window out to the ABF file.

Parameter	Description
nFile	ABF file handle.
pFH	ABFFileHeader.
pCfg	ABFScopeConfig.
pnError	Address of error return code. May be NULL.

Comments

The **ABF_WriteStatisticsConfig** function writes the ABFScopeConfig structure to the ABF file.

Possible Error Codes

One of the following error codes may be returned on error (defined in ABFFILES.H).

Constant	Meaning
ABF_EREADONLYFILE	The file is read only.
ABF_EDISKFULL	The disk is full.

Example

```
#include "abffiles.h"
BOOL CopyStatsConfig( ABFFileHeader *pFI, ABFFileHeader *pFO )
{
    if (pFI.lStatisticsConfigPtr)
    {
        static ABFScopeConfig StatsCfg;
        if (!ABF_ReadStatisticsConfig( nFileIn, pFI, &StatsCfg, &nErrorNum))
            ErrorReturn( nErrorNum );
        if (!ABF_WriteStatisticsConfig( nFileOut, pFO, &StatsCfg, &nErrorNum))
            ErrorReturn( nErrorNum );
    }
    return TRUE;
}
```

ABF_WriteDACFileEpi

#include "abffiles.h"

BOOL ABF_WriteDACFileEpi(int *hFile*, **ABFFileHeader** **pFH*,

short **pnDACArray*, **int** **pnError*);

Writes a sweep to the DAC file section.

Parameter	Description
hFile	ABF file handle.
pFH	Pointer to acquisition parameters.
pnDACArray	Data buffer of the data.
pnError	Address of error return code. May be NULL.

Comments

The **ABF_WriteDACFileEpi** function writes a sweep from pnDACArray to the DAC file section of hFile.

Possible Error Codes

One of the following error codes may be returned on error (defined in ABFFILES.H).

Constant	Meaning
ABF_EWRITEDACEPISODE	Could not write data.

ABF_WriteRawData

#include "abffiles.h"

BOOL ABF_WriteRawData(int *hFile*, **void** **pvBuffer*, **DWORD** *dwSizeInBytes*, **int** **pnError*);

Writes a raw data buffer to the ABF file at the current file position.

Parameter	Description
hFile	ABF file handle.
pvBuffer	Pointer to the buffer of data to write.
dwSizeInBytes	The amount (in bytes) of data to write.
pnError	Address of error return code. May be NULL.

Comments

This routine writes a raw buffer of binary data to the current position of an ABF file previously opened with a call to ABF_WriteOpen. This routine is provided for acquisition programs that buffer up episodic data and then write it out in large chunks. This provides an alternative to retrieving the low-level file handle and acting on it, as this can be non-portable, and assumptions would have to be made regarding the type of file handle returned (DOS or "C" runtime).

Possible Error Codes

One of the following error codes may be returned on error (defined in ABFFILES.H).

Constant	Meaning
ABF_EDISKFULL	The destination drive is out of disk space.
ABF_EREADONLYFILE	File was opened with ABF_ReadOpen
ABF_EBADFILEINDEX	Bad ABF file handle passed in.

Miscellaneous Functions

Routine	Use
ABF_BuildErrorText on page 49	Build an error string from an error number and a file name.
ABF_EpisodeFromSynchCount on page 51	Find the sweep that contains a particular synch count.
ABF_FormatTag on page 52	This function reads a tag from the TagArray section and formats it as ASCII text.
ABF_GetEpisodeDuration on page 52	Get the duration of a given sweep in ms.
ABF_GetEpisodeFileOffset on page 53	Returns the sample point offset in the ABF file for the start of the given sweep number that is passed as an argument.
ABF_GetMissingSynchCount on page 53	Get the count of synch counts missing before the start of this sweep and the end of the previous sweep.

Routine	Use
ABF_GetNumSamples on page 55	Get the number of samples in this sweep.
ABF_GetStartTime on page 56	Gets the start time in ms for the specified sweep.
ABF_HasData on page 57	Checks whether an open ABF file has any data in it.
WINAPI ABF_HasOverlappedData on page 57	Determines if there is any overlapped data in the file.
ABF_IsABFFile on page 58	Checks the data format of a given file.
ABF_SetErrorCallback on page 59	This routine sets a callback function to be called in the event of an error occuring.
ABF_SynchCountFromEpisode on page 59	Find the synch count at which a particular sweep started.

ABF_BuildErrorText

#include "abffiles.h"

BOOL ABF_BuildErrorText(int *nError*, **const char** **szFileName*,

char **szTxtBuf*, **UINT** *uMaxLen*);

The ABF_BuildErrorText function builds an error message for the specified error number.

Parameter	Description
nError	Error number to create message from.
szFileName	Name of file.
szTxtBuf	Buffer for error text.
uMaxLen	Size of szTxtBuf.

Returns

If *nErrorNum* contains a valid error number, this function places the generated text into *szTxtBuf* and returns TRUE, otherwise it returns FALSE.

Comments

The ABF_BuildErrorText function builds an error message based on *nErrorNum* and *szFileName*.

Example

```
#include "abffiles.h"
BOOL ShowABFError( char *szFileName, int nError )
{
   char szTxt[80];

   if (!ABF_BuildErrorText( nError, szFileName, szTxt, sizeof(szTxt) ))
      sprintf( szTxt, "Unknown error number: %d\r\n", nError );
   printf( "ERROR: %s\n", szTxt );
   return FALSE;
}
```

ABF_EpisodeFromSynchCount

#include "abffiles.h"

BOOL ABF_EpisodeFromSynchCount(int hFile, **ABFFileHeader** *pFH,

DWORD *pdwSampleNumber, **DWORD** *pdwEpisode,
int *pnError);

Finds the sweep number that contains a specified synch count.

Parameter	Description
hFile	ABF file handle.
pdwSynchCount	Address of synch count to search for.
pdwEpisode	Address of sweep number that contains the requested synch count.
pnError	Address of error return code. May be NULL.

Comments

The **ABF_EpisodeFromSynchCount** function finds the sweep number for the specified synch count, and stores it in **pdwEpisode*.

Possible Error Codes

One of the following error codes may be returned on error (defined in ABFFILES.H).

Constant	Meaning
ABF_EBADFILEINDEX	Invalid ABF file handle specified.

Example

```
#include "abffiles.h"
BOOL FindAnEpisode( char *pszFileName, DWORD *pdwSample,
           DWORD *pdwEpisode )
{
    int hFile;
    int nError = 0;
    ABFFileHeader FH;

    DWORD dwMaxEpi = 0;
    UINT uMaxSamples = 16 * 1024;

    if (!ABF_ReadOpen( pszFileName, &hFile, ABF_DATAFILE,
           &FH, &uMaxSamples, &dwMaxEpi, &nError ))
        return ShowABFError(pszFileName, nError);

    if (!ABF_EpisodeFromSynchCount( hFile, &FH, pdwSynchCount, pdwEpisode,
    &nError ))
    {
        ABF_Close( hFile, NULL );
```

```
        return ShowABFError(pszFileName, nError);
    }
    if (!ABF_Close( hFile, &nError ))
        return ShowABFError(pszFileName, nError);
    return TRUE;
}
```

ABF_FormatTag

#include "abffiles.h"

BOOL ABF_FormatTag(int hFile, **ABFFileHeader** *pFH, **long** lTagNumber, **char** *pszBuffer,

UINT uSize, **int** *pnError)

This function reads a tag from the TagArray section and formats it as ASCII text.

Parameter	Description
hFile	ABF file handle.
pFH	File header for the file as returned by ABF_WriteOpenABF_WriteOpen .
lTagNumber	Number of the tag entry to format. (The first tag is tag 0)
pszBuffer	The buffer to receive the formatted text.
uSize	The size of the buffer pointed to by pszBuffer.
pnError	Address of error return code. May be NULL.

Comments

If tag number -1 is requested, the ASCII text returns column headings.

ABF_GetEpisodeDuration

BOOL ABF_GetEpisodeDuration(int nFile, **ABFFileHeader** *pFH, **DWORD** dwEpisode,

double *pdDuration, **int** *pnError)

Get the duration of a given sweep in ms.

Parameter	Description
nFile	ABF file handle.
pFH	File header for the file as returned by ABF_ReadOpen.
dwEpisode	Sweep number to return the start time of. (First sweep is sweep 1).
pdDuration	The location in which to return the start time of the sweep in ms.
pnError	Address of error return code. May be NULL.

ABF_GetEpisodeFileOffset

BOOL ABF_GetEpisodeFileOffset(int *nFile*, **ABFFileHeader** **pFH*, **DWORD** *dwEpisode*,

DWORD **pdwFileOffset*, **int** **pnError*)

Returns the sample point offset in the ABF file for the start of the given sweep number that is passed as an argument.

Parameter	Description
nFile	ABF file handle.
pFH	File header for the file as returned by ABF_ReadOpen.
dwEpisodel	Sweep number to return the start position of. (First sweep is sweep 1).
pdwFileOffset	Points to the location in which to return the sample offset of the start of the sweep (in samples per channel).
pnError	Address of error return code. May be NULL.

ABF_GetMissingSynchCount

#include "abffiles.h"

BOOL ABF_GetMissingSynchCount(int*hFile*, **DWORD***dwEpisode*,
DWORD **pdwMissingSamples*, **int** **pnError*);

Returns the number of synch counts missing before the specified sweep (event detected files only).

Parameter	Description
hFile	ABF file handle.
dwEpisode	Sweep number.
pdwMissingSamples	Number of synch count absent prior to this sweep.
pnError	Address of error return code. May be NULL.

Comments

The **ABF_GetMissingSynchCount** function finds the number of synch count missing for event detected data for the specified sweep, and stores it in **pdwMissingSynchCount*.

Possible Error Codes

One of the following error codes may be returned on error (defined in ABFFILES.H).

Constant	Meaning
ABF_EEPISODERANGE	Sweep number out of range.
ABF_EBADFILEINDEX	Invalid ABF file handle specified.

Example

```
#include "abffiles.h"
BOOL CopyDataFile(char *pszFileIn, int nFileIn, ABFFileHeader *pFI,
          char *pszFileOut, int nFileOut, ABFFileHeader *pFO)
{
```

```
    UINT uNumSamples = (UINT)pFI->lNumSamplesPerEpisode;
    DWORD dwEpiStart, dwMissingSamples;
    short *pnBuffer = (short *)malloc(uNumSamples * sizeof(short));
    if (!pnBuffer)
    {
       printf("Out of memory!\n");
       return FALSE;
    }
    for (DWORD i=1; i<=(DWORD)pFI->lActualEpisodes; i++)
    {
       UINT uFlag = 0;
       int nError = 0;
       if (!ABF_MultiplexRead( nFileIn, pFI, i, pnBuffer, &uNumSamples,
          &nError ))
          return ShowABFError(pszFileIn, nError);
       if (!ABF_SynchCountFromEpisode( nFileIn, pFI, i, &dwEpiStart,
          &nError ))
          return ShowABFError(pszFileIn, nError);
       if (pFI->nOperationMode == ABF_VARLENEVENTS)
       {
       if (!ABF_GetMissingSynchCount( nFileIn, pFI, I,
             &dwMissingSynchCount, &nError ))
          return ShowABFError(pszFileIn, nError);
       if (dwMissingSynchCount == 0)
          uFlag = ABF_APPEND;
    }
    if (!ABF_MultiplexWrite( nFileOut, pFO, uFlag, pnBuffer,
          dwEpiStart, uNumSamples, &nError ))
       return ShowABFError(pszFileOut, nError);
    }
    return TRUE;
}
```

ABF_GetNumSamples

#include "abffiles.h"

BOOL ABF_GetNumSamples(int *hFile*, **DWORD** *dwEpisode*,

UINT **puNumSamples*, **int** **pnError*);

Finds the number of sampleSamples in the specified sweep.

Parameter	Description
hFile	ABF file handle.
dwEpisode	Interesting sweep number.
puNumSamples	Number of data points in this sweep.
pnError	Address of error return code. May be NULL.

Comments

The **ABF_GetNumSamples** function finds the number of samples in the specified sweep, and returns it in **puNumSamples*.

Possible Error Codes

One of the following error codes may be returned on error (defined in ABFFILES.H).

Constant	Meaning
ABF_EEPISODERANGE	Sweep number out of range.
ABF_EBADFILEINDEX	Invalid ABF file handle specified.

Example

```
#include "abffiles.h"
BOOL HowManySamples( char *pszFileName, DWORD dwSweep)
{
    int hFile;
    int nError;
    ABFFileHeader FH;
    DWORD dwMaxEpi = 0;
    UINT uMaxSamples = 0;
    UINT uNumSamples;
    uMaxSamples = 16 * 1024;
    if (!ABF_ReadOpen( pszFileName, &hFile, ABF_DATAFILE, &FH,
              &uMaxSamples, &dwMaxEpi, &nError ))
        return ShowABFError(pszFileName, nError);
    if (!ABF_GetNumSamples( hFile, &FH, dwSweep, &uNumSamples,
              &nError ))
    {
        ABF_Close( hFile, NULL );
        return ShowABFError(pszFileName, nError);
    }
```

```
    ABF_Close( hFile, NULL );
    printf( "The number of samples is %u\n", uNumSamples );
    return TRUE;
}
```

ABF_GetStartTime

#include "abffiles.h"

BOOL ABF_GetStartTime(int *nFile*, **ABFFileHeader** **pFH*, **int** *nChannel*,

DWORD *dwSweep*, **float** **pfStartTime*, **int** **pnError*);

Gets the start time in ms for the specified sweep.

Parameter	Description
hFile	ABF file handle.
pFH	File header for the file as returned by ABF_ReadOpen.
nChannel	ADC channel of interest.
dwEpisode	Sweep number to return the start time for.
pfStartTime	Location in which to return the start time in ms.
pnError	Address of error return code. May be NULL.

Comments

The **ABF_GetStartTime** function returns the time at which the sweep of interest in the channel specified started.

Possible Error Codes

One of the following error codes may be returned on error (defined in ABFFILES.H).

Constant	Meaning
ABF_EEPISODERANGE	Sweep number out of range.
ABF_EBADFILEINDEX	Invalid ABF file handle specified.

Example

```
#include "abffiles.h"
BOOL GetStartEndTime( int nChannel, DWORD dwEpisode, float *pfTimeBase,
          float *pfStartTime, float *pfEndTime,
          int *pnError )
{
if ( ABF_GetStartTime( GetFileHandle(), &GetFileHeader(),
          nChannel, dwEpisode,
          &fStartTime, pnError ) == FALSE )
    return FALSE;
UINT uSamples = GetNumberSamples( GetMaximumEpisodes() - 1, NULL );
// Compensate for the length of the last trace
fStartTime += pfTimeBase;
```

```
// Add another trace to put space between us and the last trace
fStartTime += pfTimeBase;
if( pfStartTime != NULL )
    *pfStartTime = fStartTime;
if( pfEndTime != NULL )
{
    UINT uSamples = GetNumberSamples(dwEpisode, NULL);
        *pfEndTime = fStartTime + pfTimeBase;
}

return TRUE;
}
```

ABF_HasData

#include "abffiles.h"

void ABF_HasData(int *nFile*, **ABFFileHeader** **pFH*);

Checks whether an open ABF file has any data in it.

Parameter	Description
hFile	ABF file handle.
pFH	File header for the file as returned by ABF_ReadOpen or ABF_WriteOpen

Comments

The **ABF_HasData** function will examine an open ABF file and return TRUE if there is any data in the file, and FALSE if there is not.

Example

```
#include "abffiles.h"
```

WINAPI ABF_HasOverlappedData

#include “Abffiles.h”

BOOL WINAPI ABF_HasOverlappedData(int *nFile*, **BOOL** **pbHasOverlapped*, **int** **pnError*)

Returns true if the file contains overlapped data.

Parameter	Description
nFile	ABF file handle.
pbHasOverlapped	True if file contains overlapped data.
pnError	Address of error return code. May be NULL.

Comments

The **ABF_HasOverlappedData** determines if there is any overlapped data in the file. This can only occur in Fixed-length events detected mode when one sweep finishes after the following one starts.

Possible Error Codes

The following error code may be returned on error (defined in ABFFILES.H).

Constant	Meaning
ABF_ EWRITEONLYFILE	The file is write only.

Example

```
#include "abffiles.h"
BOOL OpenABFFile( char *pszFileName, BOOL *pbOverlappedData )
{
    int hFile;
    int nError = 0;
    ABFFileHeader FH;

    DWORD dwMaxEpi = 0;
    UINT uMaxSamples = 16 * 1024;

    if (!ABF_ReadOpen( pszFileName, &hFile, ABF_DATAFILE,
              &FH, &uMaxSamples, &dwMaxEpi, &nError ))
         return ShowABFError(pszFileName, nError);
    if( !ABFHasOverlappedData( &hFile, pbOverlappedData, &nError )
    return ShowABFError( pszFileName, nError );
return TRUE;
}
```

ABF_IsABFFile

#include "abffiles.h"

void ABF_IsABFFile(const char **pszFileName*, **int** **pnDataFormat*, **int** **pnError*);

Checks the data format of a given file.

Parameter	Description
pszFileName	Path name of the file to be tested.
pnDataFormat	Location to return the value of nDataFormat if it is an ABF file. May be NULL.
pnError	Address of error return code. May be NULL.

Comments

The **ABF_IsABFFile** function is used to determine firstly whether a file is an ABF file, and then if it is an ABF file, what type of ABF file it is. The value returned in the location pointed to by *pnDataFormat* will be the same value in the nDataFormat field in the header of the file if it is an ABF file.

Example

```
#include "abffiles.h"
```

ABF_SetErrorCallback

typedef BOOL (AXOAPI **ABFCallback*)**(void** **pvThisPointer*, **int** *nError*);

BOOL ABF_SetErrorCallback(int *nFile*, **ABFCallback** *fnCallback*, **void** **pvThisPointer*, **int** **pnError*)

This routine sets a callback function to be called in the event of an error occurring.

ABFCallback

typedef BOOL (AXOAPI *ABFCallback)**(void** **pvThisPointer*, **int** *nError*);

ABF_SynchCountFromEpisode

#include "abffiles.h"

BOOL ABF_SynchCountFromEpisode(int *hFile*, **const ABFFileHeader** **pFH*, **DWORD** *dwEpisode*,

DWORD **pdwSynchCount*, **int** **pnError*);

Finds the synch count for the start of the specified sweep number.

Parameter	Description
hFile	ABF file handle.
dwEpisode	Sweep number that is being searched for.
pdwSynchCount	Synch count of the first point in the sweep.
pnError	Address of error return code. May be NULL.

Comments

The **ABF_SynchCountFromEpisode** function finds the synch count point number for the start of the specified sweep number. It sets **pdwSynchCount* to the synch count of the first point in the sweep.

Possible Error Codes

One of the following error codes may be returned on error (defined in ABFFILES.H).

Constant	Meaning
ABF_EEPISODERANGE	Sweep number out of range.
ABF_EBADFILEINDEX	Invalid ABF file handle specified.

Example

```
#include "abffiles.h"
BOOL CopyDataFile(char *pszFileIn, int nFileIn, ABFFileHeader *pFI,
       char *pszFileOut, int nFileOut, ABFFileHeader *pFO)
{
    UINT uNumSamples = (UINT)pFI->lNumSamplesPerEpisode;
    DWORD dwEpiStart, dwMissingSamples;

    short *pnBuffer = (short *)malloc(uNumSamples * sizeof(short));
    if (!pnBuffer)
```

```
    {
       printf("Out of memory!\n");
       return FALSE;
    }

    for (DWORD i=1; i<=(DWORD)pFI->lActualEpisodes; i++)
    {
       UINT uFlag = 0;
       int nError = 0;
       if (!ABF_MultiplexRead( nFileIn, pFI, i, pnBuffer, &uNumSamples,
              &nError ))
          return ShowABFError(pszFileIn, nError);

       if (!ABF_SynchCountFromEpisode( nFileIn, pFI, i, &dwEpiStart,
              &nError ))
       return ShowABFError(pszFileIn, nError);
    if (pFI->nOperationMode == ABF_VARLENEVENTS)
    {
       if (!ABF_GetMissingSynchCount( nFileIn, pFI, I,
                                      &dwMissingSynchCount, &nError ))
          return ShowABFError(pszFileIn, nError);
       if (dwMissingSynchCount == 0)
          uFlag = ABF_APPEND;
    }
    if (!ABF_MultiplexWrite( nFileOut, pFO, uFlag, pnBuffer,
              dwEpiStart, uNumSamples, &nError ))
       return ShowABFError(pszFileOut, nError);
    }
    return TRUE;
}
```

Use With Care!

The following functions are strictly a violation of the design and modularization of the ABF file I/O routines, but they are provided for the use of time-critical acquisition programs that require maximum efficiency when doing file I/O during data acquisition.

Routine	Use
ABF_GetSynchArray on page 61	Returns a pointer to the CSynch object used to buffer the Synch array to disk.
ABF_GetFileHandle on page 61	Returns the DOS file handle associated with the specified file.
ABF_ UpdateAfterAcquisition on page 63	Update the ABF internal housekeeping after data has been written into a data file without using the ABF file I/O routines.

ABF_GetSynchArray

void *ABF_GetSynchArray(int *nFile*, **int** **pnError*)

Returns a pointer to the CSynch object used to buffer the Synch array to disk.

CAUTION! Use with care!

ABF_GetFileHandle

#include "abffiles.h"

BOOL ABF_GetFileHandle(int *hFile*, **HANDLE** **phHandle*, **int** **pnError*);

Returns the DOS file handle associated with the specified file. This function should not need to be called if all access to ABF files are performed through the ABF file routines. It is provided for debugging purposes and for acquisition programs that do their own file I/O for performance reasons.

Parameter	Description
hFile	ABF file handle.
phHandle	DOS file handle.
pnError	Address of error return code. May be NULL.

Comments

The **ABF_GetFileHandle** function sets **phHandle* to the DOS file handle associated with the file specified in *hFile*. If the file is written to through the handle obtained by this function, then **ABF_UpdateAfterAcquisition** must be called prior to **ABF_UpdateHeader**.

Possible Error Codes

One of the following error codes may be returned on error (defined in ABFFILES.H).

Constant	Meaning
ABF_EBADFILEINDEX	Invalid ABF file handle specified.

Example

```
#include "abffiles.h"
```

```
BOOL Acquisition( char *pszFileName, ABFFileHeader *pFH )
{
    int hFile;
    HANDLE hHandle;
    int nError = 0;
    DWORD dwEpisodes, dwSamples;

    if (!ABF_WriteOpen( pszFileName, &hFile, ABF_DATAFILE, pFH,
              &nError ) )
    return ShowABFError(pszFileName, nError);
    if (!ABF_GetFileHandle( hFile, &hHandle, &nError ))
    {
    ABF_Close( hFile, NULL );
    return ShowABFError(pszFileName, nError);
    }
    AcquireAndWriteData( hHandle, pFH, &dwEpisodes, &dwSamples );
    if (!ABF_UpdateAfterAcquisition( hFile, pFH, dwEpisodes, dwSamples,
              &nError ))
    {
    ABF_Close( hFile, NULL );
    return ShowABFError(pszFileName, nError);
    }
    if (!ABF_UpdateHeader( hFile, pFH, &nError ))
    {
      ABF_Close( hFile, NULL );
      return ShowABFError(pszFileName, nError);
    }
    if (!ABF_Close( hFile, &nError ))
       return ShowABFError(pszFileName, nError);
    return TRUE;
}
```

ABF_UpdateAfterAcquisition

#include "abffiles.h"

BOOL ABF_UpdateAfterAcquisition(ABFFileHeader *pFH,

DWORD *dwAcquiredEpisodes*, **DWORD** *dwAcquiredSamples*,
int **pnError*);

Update the ABF internal housekeeping after data has been written into a data file without using the ABF file I/O routines. This function should not need to be called if all access to ABF files are performed through the ABF file routines. It is provided for debugging purposes and for acquisition programs that do their own file I/O for performance reasons.

Parameter	Description
hFile	ABF file handle.
pFH	File header returned from the ABF_WriteOpen call for this file.
dwAcquiredEpisodes	Number of acquired sweeps.
dwAcquiredSamples	Number of acquired samples.
pnError	Address of error return code. May be NULL.

Comments

The **ABF_UpdateAfterAcquisition** function updates ABF internal housekeeping of acquired data. This function must be called before ABF_UpdateHeader if the file has been written to via the handle obtained by ABF_GetFileHandle on page 61

Possible Error Codes

One of the following error codes may be returned on error (defined in ABFFILES.H).

Constant	Meaning
ABF_EBADFILEINDEX	Invalid ABF file handle specified.

Example

```
#include "abffiles.h"
BOOL Acquisition( char *pszFileName, ABFFileHeader *pFH )
{
    int hFile;
    HANDLE hHandle;
    int nError = 0;
    DWORD dwEpisodes, dwSamples;
    if (!ABF_WriteOpen( pszFileName, &hFile, ABF_DATAFILE, pFH,
    &nError ) )
    return ShowABFError(pszFileName, nError);
    if (!ABF_GetFileHandle( hFile, &hHandle, &nError ))
    {
    ABF_Close( hFile, NULL );
    return ShowABFError(pszFileName, nError);
    }
```

```
    AcquireAndWriteData( hHandle, pFH, &dwEpisodes, &dwSamples );
    if (!ABF_UpdateAfterAcquisition( hFile, pFH, dwEpisodes, dwSamples,
    &nError ))
    {
    ABF_Close( hFile, NULL );
    return ShowABFError(pszFileName, nError);
    }
    if (!ABF_UpdateHeader( hFile, pFH, &nError ))
    {
    ABF_Close( hFile, NULL );
    return ShowABFError(pszFileName, nError);
    }
    if (!ABF_Close( hFile, &nError ))
    return ShowABFError(pszFileName, nError);
    return TRUE;
}
```

Appendix A: ABF Hardware and Storage Limits

Some of the more important limitations in the range of hardware supported and the size of components in ABF formatted files are listed here.

- Sixteen physical ADC channels, numbered 0 to 15.
- Up to sixteen bits per ADC word.
- Up to 1,032,258 multiplexed samples per sweep in **High-speed oscilloscope** mode, **Fixed-length events** mode and **Episodic stimulation** mode.
- Up to 2 G multiplexed samples per segment in **Variable-length events** mode and **Gap-free** mode.
- **Stimulus** waveform can be generated on up to eight DAC channels simultaneously (digitizer dependent).
- **Pre-sweep Train** (previously called **Conditioning Train**) can be generated on all DAC channels simultaneously.
- One **Math** channel.
- Up to 16 telegraphed instruments (digitizer dependent)
- **P/N Leak Subtraction** can be applied to ADC channels simultaneously.
- One set of display amplifications and offsets.
- Only one averaged run per file!

File ID and Size Information

Field Name	Type	Description
float	fFileVersionNumber	File format version stored in the data file during acquisition. Present version is 2.0
short	nOperationMode	Operation mode: 1 = Event-driven, variable length; 2 = Event-driven, fixed length; 3 = Gap-free; 4 = High Speed Oscilloscope; 5 = Episodic stimulation (Clampex software only).
long	lActualAcqLength	Actual number of ADC samples (aggregate) in data file. See lAcqLength. Averaged sweeps are included.
short	nNumPointsIgnored	Number of points ignored at data start. Normally zero, but non-zero for gap-free acquisition using AXOLAB configurations with one or more ADS boards.
long	lActualEpisodes	Actual number of sweeps in the file including averaged sweeps. See lEpisodesPerRun. If nOperationMode = 3 (gap-free) the value of this parameter is 1.
UINT	uFileStartDate	Date when the data portion of the file was first written. Stored as YYYYMMDD
UINT	uFileStartTimeMS	Time of day in milliseconds past midnight when data portion of this file was first written to.

Field Name	Type	Description
long	lStopwatchTime	Time since the stopwatch was zeroed that the data portion of this file was first written to. Not supported by all programs. Default = 0.
float	fHeaderVersionNumber	Version number of the header structure returned by the ABF_ReadOpen function. Currently 2.0. This parameter does not identify the data file format. See fFileVersionNumber.
short	nFileType	Type of file. 1 = ABF file; 2 = Old FETCHEX file (FTCX); 3 = Old Clampex software file (CLPX). See sFileType.

Glossary

ADC, A/D

Analog-to-Digital converter.

char

String containing a fixed number of one-byte characters. (Not NULL terminated.)

DAC, D/A

Digital-to-Analog converter.

DWORD

32-bit unsigned integer

Episode

Synonym for “Sweeps”. Used by pClamp 6 and earlier versions.

Episodic Stimulation

In this mode, a number of equal-length sweeps (also known as episodes) are acquired. A set of parametrically related sweeps is called a run. Runs can be repeated a specified number of times to form a trial. If runs are repeated, the corresponding sweeps in each run are automatically averaged and the trial contains only the average. The trial is stored in a file. Only one trial can be stored in an ABF file.

File

Each ABF data file contains one trial.

Fixed-Length Event-Driven

Data acquisition is initiated in segments whenever a threshold-crossing event is detected. A pre-trigger portion below threshold is acquired. Unlike variable-length event-driven acquisition, the length of each segment of data is a pre-specified constant for all segments. For this reason, the segments are often referred to as sweeps. In this mode, every threshold crossing triggers a sweep, therefore fixed-length event-driven mode is also sometimes referred to as loss-free oscilloscope mode. If a second event occurs before the current sweep is finished, a second sweep is acquired triggered from the second event. This occurrence is referred to as overlap. In this case, consecutive sweeps in the data file contain redundant data.

The precise start time and length of each sweep is stored in the Synch Array. Although the length of each sweep is redundant in this mode, it is stored in order to simplify reading and writing of the Synch Array. Similarly, the storage of redundant data during overlap is not strictly necessary, but it simplifies analysis and display for each sweep to be returned as a fixed-length sweep with a known and constant trigger time. Since no triggers are lost, fixed-length event-driven acquisition is ideal for the statistical analysis of constant-width events such as action potentials.

float

IEEE floating point format, 4 bytes long.

Gap-Free

Gap-free ABF files contain a single sweep of up to 4 GB of multiplexed data. A uniform sampling interval is used throughout. There is no stimulus waveform associated with gap-free data.

Gap-free mode is usually used for the continuous acquisition of data in which there is fairly uniform activity over time.

Hierarchy

A File contains one Trial. A Trial contains one or more Runs. A Run contains one or more Sweeps. A Sweep contains one or more ADC channels.

High-Speed Oscilloscope

In high-speed oscilloscope mode a pre-trigger portion before the threshold is acquired. Unlike fixed-length event-driven acquisition, in high-speed oscilloscope mode not every threshold crossing triggers a sweep. The emphasis is on allowing the digitizer to be used at the highest possible sampling rate. Like a real high-speed oscilloscope, there is a "dead time" at the end each sweep during which the display is updated and the trigger circuit is re-armed. Threshold crossings that arrive during this dead time are simply ignored. Similarly, second and subsequent threshold crossings during a sweep do not start a new sweep. Thus there is no storage of overlapping (redundant) data.

Instrument

Refers to the external measurement equipment. For example, an Axopatch, an Axoclamp, or a SmartProbe.

int

Signed integer of the native size of the CPU

long

Four byte signed integer.

Run

A group of related sweeps. ABF files contain only one Run per file, which is the averaged run for all sweeps. Currently, the ABF routines only support one Run per file.

Sample

The datum produced by one A/D conversion or the datum describing one D/A output.

Sequence

A set containing one sample from each of the actively sampled input channels and one sample for each of the actively generated output channels.

short

16-bit signed integer

Signal Conditioner

A signal conditioner is a programmable analog device for applying filtering, gain and offset to the signal before digitization. ABF formatted files store signal conditioning information for each channel in the following arrays: fSignalGain, fSignalOffset, fSignalLowpassFilter, fSignalHighpassFilter.

Sweep

A continuous set of data samples multiplexed from all A/D channels. pCLAMP version 6 and earlier used the term “episode”.

Trace

A continuous set of data samples from a single A/D channel.

Trial

Non episodic files: A group of one or more sweeps acquired at one time. The start time and length of each sweep are described in the SYNCH array.

Episodic files: If there was no averaging, a trial is the same as the single acquired run. If there was averaging, the trial contains the average of the two or more acquired runs.

UINT

Unsigned integer of the native size of the CPU

User Units

ADC / DAC data is scaled in User Units (e.g. nA or mV) to take into account any scaling performed in either hardware or software.

Variable-Length Event-Driven

Data acquisition is initiated in segments whenever a threshold-crossing event is detected. Pre-trigger and trailing portions are also acquired. The length of the segment of data is determined by the nature of the data, being automatically extended according to the amount of time that the data exceeds the threshold. If the pre-trigger portion of the next event would overlap the trailing portion of the current event, the current segment is extended. There is no storage of overlapping data. The precise start time and length of each segment is stored in the Synch Array. Variable-length event-driven acquisition is usually used for the continuous recording of "bursting" data in which there are bursts of activity separated by long quiescent periods.

WORD

16-bit unsigned integer

Contact Us

Phone: +1-800-635-5577
Web: moleculardevices.com
Email: info@moldev.com

Visit our website for a current listing of worldwide distributors.