CFS
The CED Filing System

October 2006

First edition: August 1988
Revised: May 1989
Revised: February 1990
Revised: February 1991
Revised: February 1992
Revised: March 1993
Revised: June 1993
Revised: September 1993
Revised May 1998
Revised January 2003
Revised October 2006

Manual written by:
Armand Cachelin, Gareth Gaskell, Greg Smith, Tom Ward, Paul Cox, Tim Bergel, Joan Grayer, Simon Parker

Published by:

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

Telephone: Cambridge +44 (0)1223 420186
Fax: Cambridge +44 (0)1223 420488
Email: info@ced.co.uk
Web: http://www.ced.co.uk

Table of contents

Introduction to CFS

This manual

CFS (CED Filing System or sometimes the Cambridge Filing System) is a disk data structure designed to hold the types of data commonly encountered by experimental scientists. It can be likened to a database structure where each record holds the same number of data items, but the items can each be large arrays of data as well as single numbers or character strings. CFS is designed to handle files of many megabytes in size as efficiently as small files through the use of a table of pointers (like a database index). There is no anonymous data in a CFS file; every piece of data has units and a description.

This manual defines the CFS disk data structures, and describes a library of routines written in C to create and manipulate CFS files within the MS-DOS, Windows and Macintosh environments. This manual would also be the starting point to a programmer implementing the CFS under a new operating system or writing a library for use from another language.

CED also provides the same routines written in Borland Turbo Pascal which are designed to work with the DOS operating system. They are described in a separate chapter of the manual.

This manual can be read at several levels. If you will be using the CFS to move data between different programs which use CFS as a data storage and retrieval tool you will find this chapter of use as background information.

Why do we need CFS

In 1985 we realised that as CED wrote more application software, each programmer was designing file data formats which, while no doubt optimal for the project, were incompatible with software written for other projects. Not only were programmers spending time designing and debugging these filing systems but they were also building barriers between the software systems. If we wanted to import some data from one software system into another we had to write an intermediate translation program. If nothing else, this was a lot of work, but worse, it stopped systems becoming greater than the sum of their parts.

What we needed was a set of tools to let each program store and retrieve data in a manner which was natural and sensible for the program, but which followed rules that would allow other programs to identify the contents of the file. We decided that there should be no "anonymous" variables in the file; all stored items should have a description and units. We also stipulated that the system should not compromise the speed of continuous data recording to disk.

CFS version 2.00

The CED User Group emphasised the strategic importance of such a library, and set up a committee to advise on future developments of the CFS. The committee pointed out defects in the original system, and made suggestions for improvements. CED have now completely re-written the CFS keeping the original design concepts, but adding and amending as suggested by the committee. This document describes the API (Application Programmer Interface) of the new CFS.

The new system has been changed in ways that make it non-compatible with the original version. This will cause work modifying programs, but it was felt that the advantages far outweighed the extra work required. The longer changes are delayed, the larger the number of programs to be converted. There is a translation program which will convert an old format data file to the new format so users with data recorded with old format programs will not be too greatly inconvenienced. The changes from the original version are described in an appendix to this manual. They will be of interest to programmers converting to the new version.

Experimental models

When the CFS was designed, we had two experimental models in mind. The simpler model can be thought of as the continuous model. The more complex model can be described as the repetitive model. In both models there are two kinds of data stored: discrete data like the sex of a subject, and array data like the vibration of a beam which has been struck with a hammer. In the CFS we term the discrete data as variables and the array data as channel data (because it is usually obtained through an analogue input channel of a data acquisition device).

Continuous model

A continuous model experiment has a set of variables which describe the data and the conditions under which the data is collected plus channels of data. This data may be broken up into sections with gaps between them, but the sections form a contiguous whole nevertheless.

Repetitive model

A repetitive model experiment has a set of variables which describe the experiment, followed by repeated similar data acquisition episodes. Each of these episodes has a set of variables which describe the episode and channels of data.

Data types

We recognise that experimental situations require many different types of data to be stored. To accommodate this we support 1, 2 and 4 byte signed integer types and 1 and 2 byte unsigned types, IEEE standard 4 byte and 8 byte reals, plus character strings and text.

The channel data consists of arrays of these types (text is stored as an array of characters). A channel can be designated as `EqualSpaced`, `Matrix` or `Subsidiry`.

EqualSpaced channel data

There are many occasions where data is stored at fixed intervals in some parameter. This other parameter is most often time (for example data sampled at 1 kHz) but could just as easily be position or any other measure. It would be wasteful of space and time to write this other parameter with each data point, so we mark such data as EqualSpaced and save the increment between data points, and the value at the first data point.

Matrix channel data

Matrix channel data is modelled on a two dimensional array. Think of each channel as a column of numbers in a rectangular grid. There may be only one column, or up to 99. The data in the rows is considered to be linked, but the order of the columns may not be important. Some examples may help:

1 Imagine a Hooke's Law experiment where weights are hung on a spring and the extension is measured. The two channels would be the weight and the extension. The pairing of the values is important, but the order in which the weights were hung on the spring is not important if the experiment is conducted without over-stretching the spring. If the spring is over-extended, then the order of points will matter.

2 A firefly flies within a cage, flashing at irregular intervals. The x, y and z position of the insect, plus the time of each flash are recorded and stored as a matrix. In this case, the order of the points is important as it holds information on the insect's path.

Subsidiary channel data

Sometimes Equalspaced data may have additional information associated with each data point. For example a waveform representing an average of several other waveforms may have the error information of each point stored in addition to the average value itself. This additional information should be stored in a Subsidiary channel with a reference to the Equalspaced channel to which it belongs.

Channel data storage

It is important that the channel data can be copied quickly from the computer memory to the disk file without time-consuming formatting. We have devised a method of recording the data storage format so that data can be written in most cases in the same format in which it was sampled. It is up to the user to write the data into a data section in the most efficient format (see the function WriteData for details). For each data channel, the user must also supply the offset to the first byte of data in the data section, the number of bytes between each data point in the data section and the number of data points.

The stored channel information allows a general purpose program to collect data from any channel without needing to know details of how the data was originally written. A routine, GetChanData is provided to collect a single channel of data from a specified data section.

Revision History

New versions of the CFS library usually indicate internal improvements to increase the speed and reliability of the filing system. The details of these changes are given in the source code files. The following versions also included visible additions to the library.

CFS version 2.52 A new channel kind of "subsidiary" has been added.

CFS version 2.51 SetDSChan and WriteData were changed in the C version of the library. Both can be used on all data sections of a new file, with DS set to 0 or n.

CFS version 2.50 The `GetVarVal`, `SetVarVal`, `SetDSChan`, `WriteData` and `DSFlags` functions in the C library were modified so that they can be used on any data section in a new file, not just the last section. The Pascal version was left alone for compatibility.

CFS version 2.40 The `CFSFileSize` function was added for the C library only. The type `TDataKind` was changed to `TCFSKind` to avoid clashes with the `SON` library. Pointer parameters are `const` whenever possible. The library can now be built as a 32-bit Windows DLL as well as the 16-bit version.

CFS version 2.30 The `CommitCFSFile` function was added. Error handling was adjusted so that all errors are logged so that they can be retrieved using `FileError()`.

CFS version 2.20 The Microsoft C library functions were modified to work with DOS, Windows and the Macintosh operating systems.

CFS version 2.17 The `ClearDS` routine was added for all three languages, and to the Microsoft Pascal interface in version 2.18 (September 1992).

CFS version 2.10 A Microsoft C implementation of the CFS was added to the original Microsoft Pascal and Turbo Pascal implementations.

CFS version 2.04 The `SetComment` routine was added.

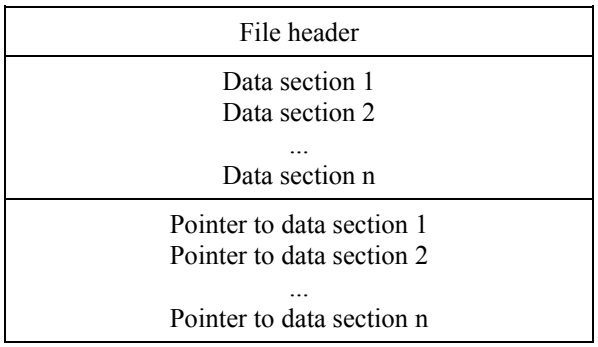
CFS version 2.03 A Turbo Pascal implementation was added to the original Microsoft Pascal implementation.

Structure of a CFS file

File structure

In the descriptions which follow, various terms such as file variable, data section (DS), data section variable and file header are used. You will have to take some of these terms on trust the first time you read through this section. They are all defined by the end.

CFS File Structure



A CFS file has a head, a body and a tail. The head holds information which describes the entire file. The body is composed of one or more data sections. The data sections all have the same structure, but need not be identical in length. The tail holds a list of pointers to the data sections to allow fast random access. Like some lizards, CFS can re-grow a tail if it gets lost by some accident!

Most programmers will not need to know anything about the details of the internal structure of the file, but a general understanding will help you to get the best out of the system. Please note that the information given below is correct at the time of writing, but should not be used directly. You should use the function and procedure calls in the CFS library to manipulate the files.

Storing data

There are three ways of storing data in a CFS file:

1. File variables. These store user defined single data items which will apply to all the data in the file, for example the age, name and sex of a subject, or the weight of some object.

2. Data section variables. These store single data items which are expected to change with each data section. Data sections are typically repeats of some protocol. For example, the maximum response to a stimulus, the temperature or a comment on each data section might be stored here.

3. Channel data. This is data which requires many values to describe. This would normally be an analogue waveform, or a sequence of digital values, or a list of times. Channel data is stored in a data section. Each data section in a file has the same number of channels of data, but the scaling of the data and the number of elements is allowed to change between data sections. For example, channel 1 in each data section would always hold the same kind of data (for example the vibration of a turbine) but the sampling rate for channel 1, and the number of points need not be the same in each data section.

Data types

There are 8 types of data which can be stored by any of the above three methods:

`INT1` Data in the range -128 to 127 stored in a single byte of memory.

`WRD1` Data in the range 0 to 255 stored in a single byte of memory.

`INT2` Data in the range -32768 to 32767 stored in two bytes of memory (for example 1401 analogue input data).

`WRD2` Data in the range 0 to 65535 stored in two bytes of memory.

`INT4` Data stored as 4 byte integer data in the range -2147483648 to 2147483647.

`RL4` IEEE format 4 byte floating point numbers.

`RL8` IEEE format 8 byte floating point numbers.

`LSTR` Character data. For file variables and data section variables these are stored as Pascal Strings, with the first byte holds the number of characters following. This means that these have a maximum size of 255 characters. For channel data this type refers to text, of any length. Lines of text are expected to be separated by CR LF character pairs. This might be used to store text entered from a pop-up note pad.

In the C version upper case letters are used for the data types in line with the C convention for naming constants. The Pascal constants are case-insensitive.

We recognise that programs will not, in general, use or support all these data types. CED currently uses the `INT2` and `RL8` data types in channel data, and the `INT2`, `INT4`, `RL4`, `RL8` and `LSTR` type in variable data, but not all in the same application!

File header

The file starts with a header which identifies the file as a CFS file, describes the file, and holds user data which doesn't change between data sections. The header is composed of five frames. A file header with n file variables, c data channels and d data section variables can be shown as:

General header
Channel 0 information Channel 1 information ... Channel c-1 information
File variable 0 information File variable 1 information ... File variable n-1 information A system file var to calculate the size of var n-1
DS variable 0 information DS variable 1 information ... DS variable d-1 information A system DS var to calculate the size of var d-1
File variable 0 File variable 1 ... File variable n-1

General header

This contains the file identifier and the parameters which determine the underlying structure of the file. The file marker holds CEDFILE" which is the same for version 1, except that the last character has been incremented from ! to ". The last character is the revision level of the CFS (!=1, "=2 and so on in ASCII sequence).

	Description	Bytes	Offset	Type
1	Marker to identify the file	8	0x00	`char[8]`
2	File name (up to 12 characters)	14	0x08	`string[13]`
3	File size (in bytes)	4	0x16	`long`
4	Time when file was created	8	0x1a	`char[8]`
5	Date when file was created	8	0x22	`char[8]`
6	Channels in data section (0-99)	2	0x2a	`short`
7	Number of file variables (0-99)	2	0x2c	`short`
8	Number of data section variables (0-99)	2	0x2e	`short`
9	Byte size of file header	2	0x30	`short`
10	Byte size of data section header	2	0x32	`short`
11	Last data section header offset	4	0x34	`long`
12	Number of data sections	2	0x38	`WORD`
13	Disk block size rounding (1=none)	2	0x3a	`WORD`
14	File comment (up to 72 characters)	74	0x3c	`string[73]`
15	Pointer table offset	4	0x86	`long`
16	Reserved space for future expansion	40	0x8a	Zero filled

Channel information

Each data section can hold up to 99 channels. The channel information section of the file header describes the characteristics of each channel which do not change from data section to data section. The following information is held for each channel:

	Description	Bytes	Offset	Type
1	Channel name (up to 20 characters)	22	0x00	`string[21]`
2	Y axis units (up to 8 characters)	10	0x16	`string[9]`
3	X axis units (up to 8 characters)	10	0x20	`string[9]`
4	Data type e.g. int2, rl4	1	0x2a	`TDataType`
5	Data kind (equalSpaced, matrix or subsidiary)	1	0x2b	`TCFSKind`
6	Byte space between elements	2	0x2c	`short`
7	Next (matrix) or master (subsidiary) channel	2	0x2e	`short`

There are three types of channel data: `equalSpaced`, `subsidiary` and `matrix`. `equalSpaced` data is typically analogue data where the time interval between each data point is fixed and the sequential position of the data in the channel is important. `subsidiary` data is like `equalspaced` data, but is extra data associated with an `equalspaced` channel such as errors. `matrix` data implies an N dimensional array (where N can be 1 for a simple list of values). This is used for a set of (x,y) positions, for example. The next channel description holds the channel number of the next channel in the matrix, or the current channel number for a one dimensional matrix. The last channel in a matrix points back at the first channel.

It is entirely up to the user how the channel data is organised within a data section, but the gap between consecutive elements of a channel must be fixed. This allows us to store either interleaved or non-interleaved data.

File variable information

The file variable information frame is used to describe each of the file variables. The following is stored for each file variable.

	Description	Bytes	Offset	Type
1	Description (up to 20 characters)	22	0x00	`string[21]`
2	Variable type (e.g. int2 or rl4)	2	0x16	`TDataType`

3	Variable units (up to 8 characters)	10	0x18	`string[9]`
4	Byte offset of variable	2	0x22	short

By convention the first file variable should be used to store a dummy value, with the description and unit fields used to store identification for where the file was produced.

The byte offset of the variable is the offset from the start of the file variable area to the first byte of the variable.

Data section variable information

The data section variable information frame is used to describe each of the data section variables. The format is exactly the same as for the file variable information, except that offset to the variable is from the start of the data section variable area.

IMPORTANT

It should be noted that the way in which Strings are described for internal use, ie Channel name, is NOT the same as the way the programmer should define their own string variables. The programmer would declare a string of length n and could use all n characters. The CFS internal string variables are declared as length m but can only be used to store m-1 characters. The reasons for this are explained further (see page 8).

The file variables

These are stored as a continuous list and are accessed using the offsets stored in the file variable information frame.

Data sections

A data section is an optionally repeated section of a CFS file which holds all the user data except the file variables. Each data section has two parts: the data section header and the channels of data, referred to as the data section data.

Data section header

This header contains the information which is allowed to vary from data section to data section. It is divided into three frames as shown below:

General data section header
Channel 0 information Channel 1 information ... Channel c information
Data section variable 0 Data section variable 1 ... Data section variable d

The general data section header

This frame is used to link the data sections together, to record the size of the channel data and to save 16 flags which are used to mark data blocks.

	Description	Bytes	Offset	Type
1	Pointer to previous data section header	4	0x00	`long`
2	Pointer to start of channel data for this DS	4	0x04	`long`
3	Size of this channel data area	4	0x08	`long`
4	Flags for marking data sections	2	0x0c	`TSFlags`
5	Reserved space for future expansion	16	0x0e	Zero filled

The data sections are linked together by a list of pointers which point to the previous section. We do not use a doubly linked list (with forward and backward links) because of the speed penalty imposed when writing data (you would have to move to the previous block to fill in the forward link). You should make no assumptions about the relative

positions of the data section header and the data section channel data. In this implementation the channel data comes first (for reasons of speed when writing) but this may change in future versions.

Channel information

The data section channel information describes the channel parameters which can change between data sections.

	Description	Bytes	Offset	Type
1	Offset in data section to first byte	4	0x00	`long`
2	Data points (not bytes)	4	0x04	`long`
3	Y scale (integer/word data)	4	0x08	`float`
4	Y offset (integer/word data)	4	0x0c	`float`
5	X increment (equalSpaced and subsidiary data)	4	0x10	`float`
6	X offset (equalSpaced and subsidiary data)	4	0x14	`float`

The Y scale and offset are used to scale the integer and word data types to real values. They are currently not used for real data or text.

The X increment is the change in the "X" parameter between each channel point. The X offset is the value of the "X" parameter for the first point on the channel.

CED suggest that the X and Y scale and offsets be set to 0.0 if they are not used, for future compatibility. The limits of the `Real` type used are 8.43×10^{-37} to 3.3×10^{38}.

Pointer table section

The pointer table is simply a list of the positions in the file of each of the data section headers. This list is used to speed up random access to the file. The list can (optionally) be read into memory for a further speed improvement. Each pointer is an `int4` holding the byte position in the file of the first byte of each data section header.

It is possible for this list to become detached from the file if a CFS file is opened in read/write mode and not closed. The next time the file is opened the list will be rebuilt. (The rebuilt list will be added to the damaged file even if it is opened in read only mode.)

Notes on data types

The ordinal data types, and the `TSFlags` type are defined in the following chapter on using the CFS. The `String[n]` types used to describe strings need a little more explanation to avoid confusion.

C version of library

User strings should be declared in the normal way:

```
char string[n+1];
```

where `n` is the maximum number of characters you want to store and the extra `1` is for the terminating `NULL` character. When a file or DS variable, which is to be a string is defined it is defined with `vType=LSTR` and `vSize=n+1`.

Internally, the CFS library will allow an extra byte in the file and store in it the actual number of characters in the stored string. Thus the string will be stored in the format described above. When the string is read back using the library function the length byte will not be included.

Some string types have been defined for your use (see type definitions in Using CFS from C). These are dimensioned to take the maximum number of characters for the various CFS storage types.

Pascal version of library

User strings should be declared in the normal way, ie `STRING[20]` to hold 20 characters. When a file or DS variable is defined which is to be a string, it is defined as `vType=lstr` and `vSize=n+1`, where `n` is the maximum number of characters it may hold and the `+1` is for the length byte at the start.

Internally, the CFS library will allow an extra byte in the file, so that a null character may be placed at the end of the string. The result is a string which consists of a length byte, followed by a stream of characters, followed by a null character and then maybe some spare space. This allows languages which do not use character counts, such as C to read the string by simply incrementing the pointer to the variable. When a string is read back by Pascal, the null character should not be used.

CFS internal strings are NOT defined in the same way, as they allow space for the null character directly by defining a string one character longer than required. The way in which these are defined should not affect the way in which you define your own strings. The system must define its strings in this way so that every byte in the header and other structures can be accounted for properly.

If you are not using C or Pascal

The information given below, together with a source of the CFS should provide all the information you need to implement the CFS routines in other languages, or on other systems apart from MS-DOS (or Windows). Most of the data types used translate very easily into other languages than Pascal.

To make life easier for languages which terminate strings with a zero character and omit character counts, all the strings written to a CFS file have a zero byte added by the CFS library routines. This zero character does not concern the programmer as it is added to the string by the CFS library immediately prior to storage and removed when the string is read back. The zero is not included in the character count. Thus if you have a pointer to a `STRING` variable copied from a CFS file in C, you can have a pointer to a C string by adding 1 to the pointer.

To help software migrate between systems and languages, and to minimise the documentation problem, please implement the same routines for accessing CFS files as we have. You may wish to produce a super-set of our routines, we would be interested to know what we have missed out.

The CFS is written with the "least significant byte at the lowest address" model of memory. This may cause problems if data is to be transferred to a different machine (for example 68000 based machines like the Macintosh which put the most significant byte at the lowest address). We believe that there is too great a speed penalty to insist that data should be stored in the format most convenient for MS-DOS on other machines. We suggest that CFS is implemented using the defined structure, but with the byte sequencing within variables and array elements in the natural order for the machine. However, a utility should be written to convert the files to the "least significant byte at the lowest address" model to allow image transfer of files via networks or other methods. If your machine does not support IEEE format reals you will also need to convert these.

CFS File conventions

Why have conventions

The CFS file structure has been designed to be flexible enough to allow almost any arrangement of experimental data. The usefulness of CFS depends upon this flexibility, but it can cause difficulties. A CFS file produced by one program may not be useful when read into another program; so much depends upon the way in which the data is arranged within the file. Because of this, CED is developing a set of conventions in CFS file design that will help to allow CFS files to be used in an intelligent manner by a properly designed program.

These conventions are not mandatory, but CED urges that you adhere to them wherever possible. On the other hand, CFS was designed to be flexible and if a convention is restricting your use of CFS overmuch then it should be ignored, particularly if you do not expect the CFS files to be used by other programs.

File arrangement

If you are designing a CFS file structure, try to ensure that the data is held in a general-purpose manner. The arrangement of data that most current CFS programs deal with is channels of INT2 equal-spaced data and time markers, if available. If some extra data is required, try using the markers to hold it, if this is not possible use data section variables. Most programs will ignore this extra information, particularly if you add channels to the marker matrix.

As a general point we would urge programmers not to be too creative in their use of the CFS facilities. If you must encrypt some special information in the data section flags or the description of a file variable then do so, but other programs may not be able to access this information in a useful manner.

Comments and descriptions

A CFS file contains a number of strings whose precise use is determined by the applications programmer. These are the file comment and the file and data section variable descriptions. CED would urge that these strings hold only printing characters forming human-readable descriptions of the file and variables and not left blank or used to hold complex information to be interpreted by an application.

One can imagine a file comment having the first 8 chars holding 8 bytes of user ID, the next six having a code for the type of experiment and so forth. This will work fine for a specific program but might look ridiculous if another program tried to display the comment for a user. It would be better to put this sort of information in separate file variables.

Channel naming

Every channel in a CFS file has a name associated with it, the use of this name is up to the applications programmer. CED would urge that programs that create CFS files do not allow blank channel names and that, by default, the channel name should indicate the source of the data such as 'ADC 0'. For many applications the channel name should indicate what the values are, as in 'Resp' or 'ECG'. Again, please ensure that the names are human-readable and contain only printing characters.

X and Y units

Please use SI units inside CFS files wherever possible. This allows programs to recognise the data they are dealing with. It also allows them to adjust the units and internal scaling in steps of 103 to keep the numbers on graph axes decent.

A special case of units information occurs when the X units of equal-spaced data is time, which can often be treated in a specialised manner. An excellent example of this is a file where data is sampled in a number of sections with gaps. Careful programming can treat this file as one long data section with holes in it, which is a particularly useful model. Channels which hold equal-spaced data where the spacing is over time should have an X units of 's' in order for them to be correctly recognised.

Once again, unit strings should not be blank, contain only printing characters, and should be suitable for labelling graph axes.

File variable zero

This file variable should always be used to hold information about the program that produced the file. The type of the variable should be `INT2`, the variable units should be the name of the program in upper case (for example PATCH or SIGAVG) and the variable description a longer description of the source program. The variable value should be set to 100 times the program revision level (version 2.03 would be stored as 203).

All programs creating CFS files are expected to adhere to this convention. Programs should look at file variable zero to determine if a given file is one produced by itself. If a file is known to be self-produced, you can take advantage of this knowledge to make maximal use of the file, while the operations possible on foreign files may be limited.

File and data section variables

As a matter of general policy CED suggests that all file and data section variables have a readable description, and that programs access variables by description rather than by number. By this we mean that you would use a procedure to find the value of the data section variable with the description 'Sweeps' rather than relying on this being a specific variable number. This means that if a later version of a program uses a different arrangement of variables, it will still work with the early model files.

It is also a good idea to include extra, unused, rl4 variables in the file. These can hold data extracted from the file, or allow for simple extensions.

Flags

Data section flags provide a convenient, quick and compact way of indicating conditions. A given flag will have the same meaning in all data sections in a file, but all other aspects of flag usage are application specific.

Marker information

A marker, in CED terminology, is a flag that indicates a specific time that is of interest and provides some information about it. An obvious example of this is a chart recorder program, where the user can press a key to flag a time, the key pressed is stored with the time to allow sorting of different types of markers.

For the CFS, markers are implemented as a form of `matrix` data. A channel matrix containing marker information is indicated by a defined pattern of matrix channels and channel names and units, the matrix can be expanded to include other, optional, marker information. As far as is possible, programs should treat a marker matrix as a single channel of structured information, whose time values correspond to the times in other, equal-spaced, channels of data in the same file.

The basic CFS marker matrix will consist of two channels as follows:

1. A channel of `INT4` data, with a name starting with the text "`Marker`" and a Y units name starting with the character "`s`". By default, the name would be "`Marker time`", but this could be altered to allow multiple sets of markers in a CFS file. The channel data will hold the time to which the markers are attached, these time values will be in ascending order throughout each data section. The `scale` and `offset` values would be used to calculate the actual time, in seconds, associated with the marker. Upper or lower case characters can be used in the name and units identifiers.
2. A channel of `INT4` data holding information about the marker. The name for this channel would be used as the name for the 'marker channel' as a whole. The units and use of this data is application dependent - but see below.

The presence of a marker matrix channel in a CFS file will be recognised by the presence of a matrix of channels including two channels containing `INT4` data, where an `INT4` channel can be found with the appropriate channel name and Y units. If these criteria are matched, the entire matrix of channels can be treated as a single channel of markers by programs which assume nothing about the source of the file.

There is a general purpose arrangement of marker information used at CED which matches the information used by Spike2. This treats the marker data as four bytes. The first byte holds information about the type of the marker - currently we use 0 to indicate the start of a section of data, 1 to indicate the end of a section, and the ASCII value of the key pressed to indicate a user keypress. The other three bytes hold other, unspecified, information.

There is no limit to the number of different marker matrices that can be included in a CFS file, nor are there any restrictions on the other data that might be included in the matrix. Note however that extra marker information may not necessarily be used by general purpose programs. If there are multiple `INT4` channels in the matrix, then the channels used to supply the marker information will be the first such channel encountered in the channel matrix.

Error information

Error information can be stored in a subsidiary CFS channel. This information will probably need to be stored in an application-specific manner, but we do have the current convention that an error channel is a channel of subsidiary type whose name starts with “Error” and we encourage other software to use the same convention so that error data can be generally recognised. In order for error data to be recognised and used by the CED Signal software, the subsidiary channel units should be “SSD” and the actual data values stored should be the sum of the squares of the differences from the mean value. This will allow Signal to calculate SD and SEM values as required and is a generally useful arrangement – we encourage others to use it.

There are no other conventions currently in use for defining other types of subsidiary data.

The CFS library

The C version of the CFS library works with DOS, Windows and the Macintosh operating systems. To use the CFS functions you need to include the CFS header file in your program:

```
#include "cfs.h"
```

The files CFS.H and MACHINE.H should also be in your current directory. These will include all the definitions you need to use CFS in your program.

Linking programs

DOS — Program should be linked with the object module CFS.OBJ, which should be in your current directory.

Windows — 16-bit applications should be linked with the CFS16.LIB library, which should be in your current directory or a location accessible to your linker. Your application should also have access to the CFS16.DLL dynamic-link library.

32-bit applications should be linked with the CFS32.LIB library, which should be in your current directory or a location accessible to your linker. Your application should also have access to the CFS32.DLL dynamic-link library.

Windows applications that are not written in C can still use the CFS DLLs, use CFS16.DLL for 16-bit programs and CFS32.DLL for 32-bit. You will need an equivalent to CFS.H for your programming language, CED distributes some of these with the CFS software, or you can develop your own. Depending upon your programming environment, you will need to link with the CFSXX.LIB library, generate your own library, or link to the DLL in some other fashion. Contact CED for assistance if you have problems with a specific programming environment.

Mac — Program should be linked with the CFS object module.

Constants defined

The header file CFS.H contains definitions of constants to be used in programs using the CFS functions.

Two constants are defined which are used to select between file variables and data section variables in some of the variable related functions:

```
#define FILEVAR     0
#define DSVAR       1
```

A set of 16 constants are used to define the flag bits. These flags can be stored in CFS data section header:

```
#define FLAG7   1
#define FLAG6   2
#define FLAG5   4
#define FLAG4   8
#define FLAG3   16
#define FLAG2   32
#define FLAG1   64
#define FLAG0   128
#define FLAG15  256
#define FLAG14  512
#define FLAG13  1024
#define FLAG12  2048
#define FLAG11  4096
#define FLAG10  8192
#define FLAG9   16384
#define FLAG8   32768
```

This constant defines a flag value with no flags set

```
#define noFlags 0
```

A set of 8 constants are defined which are used to describe a data type. These are used in describing file or data section variables and channel data:

```
#define INT1  0            /* single byte data */
#define WRD1  1            /* single byte data */
#define INT2  2            /* 2 byte integer data */
#define WRD2  3            /* 2 byte integer data */
#define INT4  4            /* 4 byte integer data */
#define RL4   5            /* 4 byte floating point */
#define RL8   6            /* 8 byte floating point */
#define LSTR  7
```

Three constants are defined which describe the storage method for channel data:

```
#define EQUALSPACED   0
#define MATRIX        1
#define SUBSIDIARY    2
```

Some constants are defined which represent the maximum number of characters which can be stored in the various string formats of the CFS file:

```
#define DESCCHARS    20    /* description 20 chars */
#define FNAMECHARS   12    /* filename 12 chars */
#define COMMENTCHARS 72    /* comment 72 chars */
#define UNITCHARS     8    /* units 8 chars */
```

Types defined

The header file CFS.H defines variable types for use in your program.

The type TSFlags is a 16 bit unsigned short to correspond to the SET used for flags in the Pascal version. Each flag occupies 1 bit and they are arranged so they will be stored in the same order as the Pascal SET

```
typedef unsigned short TSFlags;   /* DS flag definition */
```

The type TDataType is used to set the type of data in a file variable, data section variable or data section channel. This type occupies 1 byte of memory. Its value should always be set to one of the type constants INT2,WRD1 etc. supplied.

```
typedef char TDataType;           /* INT1, WRD1, etc. */
```

The type TCFSKind sets the style of channel data. It's value should always be one of the type constants EQUALSPACED, MATRIX or SUBSIDIARY supplied.

```
typedef char TCFSKind;            /* EQUALSPACED, MATRIX, etc. */
```

There are 4 string types defined which can be used to declare strings of sufficient length to hold strings of the various CFS storage types:

```
typedef char TDesc[DESCCHARS+2];        /* descriptions */
typedef char TFileName[FNAMECHARS+2];   /* filenames */
typedef char TComment[COMMENTCHARS+2];  /* comments */
typedef char TUnits[UNITCHARS+2];       /* units */
```

The type TVarDesc is used to store the description of either a file variable or a data section variable. Each variable stored must have such a description:

```
typedef struct
{
    TDesc     varDesc;   /* description of variable */
    TDataType vType;     /* one of INT1,WRD1 etc. */
    char      zerobyte;  /* for MS Pascal compatibility */
    TUnits    varUnits;  /* variable units */
```

```
    short     vSize;     /* no. of characters if LSTR */
}TVarDesc;
```

This type is packed so there are no spare bytes between its elements.

Not all fields are relevant for all variables. In particular the field `vSize` is only set for string variables (type `LSTR`) where it is set to the maximum number of characters for the string. All other variables types have known, fixed sizes.

The type `TSuperDesc` which is a pointer to `TVarDesc` is provided for handling arrays of variables descriptions:

```
typedef TVarDesc *TSuperDesc;
```

Simple data types

These types provide formal definitions of data types and pointers to various types of data, both for use as function parameters and to ensure that `FAR` pointers are used where necessary.

```
typedef unsigned short     WORD;
typedef char        FAR * TpStr;
typedef short       FAR * TpShort;
typedef float       FAR * TpFloat;
typedef long        FAR * TpLong;
typedef void        FAR * TpVoid;
typedef TSFlags     FAR * TpFlags;
typedef TDataType   FAR * TpDType;
typedef TCFSKind    FAR * TpDKind;
typedef WORD        FAR * TpUShort;
```

Global variable

The variable `noFlags` of type `TSFlags` is provided. All bit fields in `noFlags` are initialised to zero. It can be used by those not interested in flags when using the function `InsertDS`.

Routines in the CFS library

Overview

CFS files are opened in one of three ways: as a new file, as a read-only old file, and as a read-write old file. Some procedures and functions operate differently in the different modes. In the table below the routines are coded to point out these differences:

R Can be used with any file opened for reading.

R* Can be used with a write enabled old file.

W Can be used with a new file.

W* Can be used with a new file, but the `dataSect` parameter is ignored and the current data section is used.

Routine	Modes	Purpose
`CreateCFSFile`	W	Create a new CFS file.
`SetFileChan`	W R*	Set the unchanging channel description.
`SetDSChan`	W R*	Set the changing part of the channel description.
`WriteData`	W R*	Write to channel data area.
`SetWriteData`	W	Used directly before `WriteData` to increase speed.
`InsertDS`	W	Set the position of the current data section in the file.
`RemoveDS`	W R*	Unlink a data section from the file, effectively delete it.
`ClearDS`	W	Clear any data already written to the current DS.
`AppendDS`	W R	Add a new DS to the end of an old file.
`SetVarVal`	W R*	Set the value of a file or data section variable.
`SetComment`	W R*	Update the file comment information.
`CloseCFSFile`	W R	Close a CFS file, release memory used and tidy up.
`OpenCFSFile`	R	Open an old CFS file in read-only or read-write mode.
`GetGenInfo`	W R	Read file date, time and comment.
`GetFileInfo`	W R	Read file structure parameters.
`GetVarDesc`	W R	Read file or data section variable description.
`GetVarVal`	W R	Read the value of file or data section variable.
`GetFileChan`	W R	Read the unchanging part of a channel description.
`GetDSChan`	W R	Read the changing part of the channel description.
`GetChanData`	W R	Read specific channel of specific data section.
`CFSFileSize`	W R	Read the size of the CFS file.
`GetDSSize`	W* R	Read the size of a data section.
`ReadData`	W R	Read channel data area from any data section.
`DSFlags`	W R	Read or change a data section block flags.
`FileError`	W R	Collect an error code and the routine where it happened.
`CommitCFSFile`	W	Commits a new CFS file to disk.

Differences between new and old files

When you read or modify an existing file you are allowed random access to all the data sections of a file. While you are creating a new file your access is more limited. This is because each data section inherits its data section variables from the previous data section written. This inheritance runs in order of writing, not in the order the data sections are linked together (as you can insert data sections in any order). This is intended to mirror an experiment where most variables stay the same between runs. The calls marked W* in the list above will ignore the data section parameter and will transfer information to the current data section when you are working with a new file. The two exceptions to this are `ReadData` and `GetChanData` which take special precautions not to disturb the inheritance process.

In the C library, many of these restrictions have been relaxed. It is possible that future revisions of the CFS API will allow complete random access with new files. To avoid the need to change your code you should set the data section parameter to 0 for the affected calls which we take to mean the current data section.

To create a new file

The sequence of operations required to create and write data to a new CFS file is:

1 Build the variable description arrays for all the variables you intend to use. It is often a good idea to include extra `RL4` variables for later use to hold results calculated from data at the analysis stage.

2 Call `CreateCFSFile` to allocate space for and initialise the file header, to set-up the file and data section variables, to allocate space for and initialise the channel descriptions and to allocate space for and to initialise the current data section header (which is kept in memory).

3 For each data channel you must call `SetFileChan` to set the fixed channel descriptions. If your channels will not change between data sections you may also call `SetDSChan` here to avoid calling it for each channel every time you output a new data section.

4 At any point after step 2 you can set the value of file and data section variables. If you set a file variable more than once the value saved in the file is the last value set before `CloseCFSFile` is used. All data section values remain in force until changed, they are remembered from one data section to the next.

5 For each data section you would use `WriteData`, `SetVarVal`, `GetVarVal` and `SetDSChan` (if your channel parameters change) in any order you wish. The CFS writes channel data through the `WriteData` routine. The current data section is written at the end of the file and CFS records the maximum size of the data section implied by the `WriteData` routines. When you have finished writing data to the current data section move on to step 6.

6 It is possible to use `WriteData`, `SetVarVal`, `DSFlags` and `SetDSChan` on previous data sections if you wish, but this is less efficient than operating on the current data section..

7 Use `InsertDS` to close the current data section and write the data section header (which holds the data section variables) to the physical end of the file. The data section is logically inserted anywhere you like in the list of data sections written so far. This is done by breaking the linked list of data section headers at the requested position and inserting the new header. It is most efficient (fastest) to insert data sections at the end of the file. Once you have called `InsertDS` all operations now apply to the next data section.

8 When you have written all your data sections use `CloseCFSFile` to close the file and tidy up memory. `SetComment` can be used at any point before this to update the comment entered by `CreateCFSFile`.

To use an old file

Extracting information from a CFS file is much simpler than creating the file. The typical operations required to open an existing file and to extract information are:

1 Use `OpenCFSFile` to get a file handle to the required data file. If you only want to extract data, set `enableWr False` to avoid inadvertent changes to the file. To edit the file you must set this `True`.

2 Use `GetFileInfo` to read the basic file parameters (number of file variables, data section variables, channels and data sections).

3 Optional step for programs which require a specific source of data. Use `GetVarVal` to check file variable 0 to determine if this is a file from the correct source.

4 You can then use any of the procedures listed above which are allowed in read mode. General purpose programs will use the `Get`... family of calls to extract data from the file and the `Set`... family (not `SetWriteData`) to edit values. More specialised programs will use the `ReadData` and `WriteData` routines.

5 Use `CloseCFSFile` when you have finished with the file.

Reading from a newly created file

Usually you will close a data section using `InsertDS` before attempting to read data back from it. This ensures that the CFS file contains both the data and its associated header. There are circumstances however in which you may wish to read back data written to the current data section. (ie. before closing it). To do this you must keep the program record of the data header up to date by calling `SetDSChan` each time data is written to the data section. Thus the number of data points for each channel will be known at the point at which you wish to read back the data.

An important note when using C

When a function returns a string or value via a pointer it is up to you to make sure that sufficient space is allocated to the pointer for the return string or value. If you do not then it is possible to overwrite memory and your program will fail in an unpredictable and inconsistent way.

CreateCFSFile

This function is used to create a CFS file for writing. If a file exists of the same name it is truncated to zero length and then rebuilt. This is the first step in writing a CFS file.

```
short CreateCFSFile(TpStr fName,TpStr comment,
            WORD blockSize,short channels, TSuperDesc fileArray,
            TSuperDesc DSArray, short fileVars, short DSVars);
```

`fName` Pointer to the string holding the name of the file to be created for writing. This name can include a MS-DOS path. The file name will be stripped from this and written into the file header.

`comment` Pointer to the string holding the file comment, can be up to 72 characters long. If the comment is longer than this it is truncated to 72 characters.

`blockSize` This parameter optimises disk usage for your particular application. Disk systems store data physically in sectors of a fixed size. Data is always read and written in integral units of disk sectors. Operating systems (like MS-DOS) hide this from the user and let you use the disk as though you had access to individual bytes. Disk performance can be enhanced, at the cost of using more disk space, by rounding up the sizes of the file header, the data section header and the data section to a multiple of the sector size. Under MS-DOS, set this to 512 to round up to sectors (this means a minimum data section size of 1024 bytes) or to 1 for the most compact, but slower, file.

`channels` The number of channels of data in each data section.

`fileArray` Pointer to an array of variable description records for the file variables. You must set this up before you call this function by filling in the variable description, type, units and size for every variable you have set.

`DSArray` Pointer to an array of data section variable descriptions. You must set this up in the same way as `fileArray` before you call this function.

`fileVars` The number of file variables described in `fileArray` in the range 0-100.

`DSVars` The number of data section variables described in `DSArray` in the range 0-100.

This function allocates space for the file header and for a data section header in memory, and initialises all the variables to default values. All `INT1`, `WRD1`, `INT2`, `WRD2` and `INT4` variables are set to 0. `Real` variables are set to 0.0 and strings are set to `NULL`.

The data channels defined are all set to a default state (the default values are given in `SetFileChan` and `SetDSChan` below).

The function returns either a positive file handle, or a negative error code. The CFS system can have up to 16 data files open at any one time. This is an arbitrary limit built into the CFS library. It is a simple matter to rebuild the library for more files, at the cost of more memory used to hold file information.

Error codes are listed in the description of the `FileError` function.

SetFileChan

This function is used to set the fixed channel information for a single channel. This function must be used to describe each data channel before any data is written to the file.

```
void SetFileChan(short handle, short channel, TpStr chanName,
              TpStr yUnits, TpStr xUnits, TDataType dataType,
              TCFSKind dataKind, short spacing, short other);
```

`handle` The file handle returned from `CreateCFSFile` or `OpenCFSFile`, used to identify the file. You can have up to three files open at a time.

`channel` The channel number to be set-up. You must use this routine to set the fixed information for all the channel numbers that you requested in `CreateCFSFile`. NB 1st channel is number 0.

`chanName` This is a pointer to a string describing or naming the channel. Only the first 20 characters will be saved.

`yUnits` Pointer to a string holding the y axis units for the channel. If the data is scaled then these units should be the units to use after scaling. Only the first 8 characters that you supply are saved.

`xUnits` Pointer to a string holding the x axis units for equal spaced data channels. If the data is scaled then these units should be the units to use after scaling. This is saved, but not assigned special meaning for matrix data. Only the first 8 characters are saved.

`dataType` The type of each element of the data saved on this channel. This parameter would be `INT1`, `WRD1`, `INT2`, `WRD2`, `INT4`, `RL4`, `RL8` or `LSTR`. The `LSTR` type indicates text, not a character string.

`dataKind` Set to either `EQUALSPACED`, `MATRIX` or `SUBSIDIARY`. (The `LSTR` data type would normally be a 1 dimensional matrix).

`spacing` The number of bytes between the start of the first data element and the second data element in the data section on disk. This is telling the system how you intend to store your data. For `INT2` data stored with no other intervening data, this would be 2. If you had 2 channels of interleaved `INT2` data, this would be 4. If you had 1 channel of `INT2` Data and one channel of `INT4` data interleaved this would be 6.

`other` The next channel number, for `MATRIX` data. For `SUBSIDIARY` data, the master channel to which the subsidiary data refers. For `EQUALSPACED` data, the subsidiary channel associated with it if it has one, or zero. If the channel is a one dimensional matrix, set this to the current channel number.

Any errors detected in this function are returned via the `FileError` function, described below. You should use this function with great care if your file was opened with `OpenCFSFile`, preferably avoid it completely.

A note on units

It is advisable always to use SI units for data in a CFS file, as this enables generalised programs to use the data more easily.

Default state after CreateCFSFile

If this routine is not called for a channel all the string variables are set to `NULL`, `dataType` is set to `INT2`, `dataKind` to `EQUALSPACED`, spacing to 2 and other to 0.

SetDSChan

This function is used to set the channel information which can change between data sections. We anticipate that most uses of the CFS will have invariant channel information (i.e. each data section will have the same parameters), so there is no need to call this routine for every data section. Once a channel is set-up the values are remembered and this routine need only be called for changes to the parameters.

When writing a new file you must call `SetFileChan` before you use this routine for each channel. You should call `SetDSChan` for each channel before the first time you use `InsertDS`. When reading this routine can be used (with care) to change the fixed channel parameters for a specific data section. Remember you are responsible for setting sensible parameters with which to interpret your data.

```
void SetDSChan(short handle, short channel, WORD dataSect,
               long chOffset, long points, float yScale,
               float yOffset, float xScale, float xOffset);
```

`handle` The file handle returned by `CreateCFSFile` or `OpenCFSFile`.

`channel` The data section channel number to use. Channel numbers start at 0.

`dataSect` This is the data section in which to change the channel information. For files opened with `CreateCFSFile` set this to zero to use the current, unclosed, data section.

`chOffset` The byte offset from the start of the data section data to the first byte of this channel. Data is written with the `WriteData` function described below. It is entirely your responsibility to set the format of the data you write. All you are doing here is telling the CFS where to find the data for a particular channel.

`points` The number of data points for this channel.

`yScale` The `yScale` and `yOffset` parameters are used to convert data stored in integer and word formats (`INT1`, `WRD1`, `INT2`, `WRD2`, `INT4`) to the real world units defined for this channel in `SetFileChan`. This means that we are allowing the user to change gain and offset between successive data blocks. For `RL4`, `RL8` and `LSTR` please set this to 1.0.

`yOffset` Used with `yScale` to convert data to real world units. The calculation implied is that real units=data*`yScale` + `yOffset`. We ignore the offset for `RL4`, `RL8` and `LSTR` channels.

`xScale` Used only for `EQUALSPACED` and `SUBSIDIARY` channels. This is the increment in the "x" position between consecutive data points. Please set to 0.0 for `MATRIX` channels.

`xOffset` Used only for `EQUALSPACED` and `SUBSIDIARY` channels. This is the "x" position of the first point on the channel. Please set to 0.0 for `MATRIX` channels.

Any errors detected in this function are returned via the `FileError` function, described below. If your file was opened with `OpenCFSFile` you should use this function with care.

Default state after CreateCFSFile

If you do not call `SetDSChan` for a particular channel then `chOffset`, `points`, `yOffset` and `xOffset` are set to 0 and `yScale` and `xScale` are set to 1.

WriteData

Write the channel data into the current data section. This can be used on files opened with both `CreateCFSFile` (for writing) and `OpenCFSFile` (as long as the `enableWr` flag is set). If you are writing to a file opened with `OpenCFSFile` you will get an error if your write would extend beyond the data area originally written.

```
short WriteData(short handle, WORD dataSect, long byteOff,
             WORD bytes, TpVoid dataADS);
```

`handle` The handle of the file to which to write the data.

`dataSect` The data section to be updated, which must already exist. For files opened with `CreateCFSFile` set this to zero to use the current, unclosed, data section.

`byteOff` The byte offset within the data section data at which to start writing. You can write to any byte offset, but you should be aware that writes of integral numbers of disk sectors starting at a disk sector boundary are faster than random writes. See the `blockSize` parameter to `CreateCFSFile`.

`bytes` The number of bytes of data to write.

`dataADS` A pointer to the data buffer to write to the file.

`WriteData` returns either 0 if no error was detected or a negative error code. Error codes are listed in the description of the `FileError` function.

SetWriteData

This is an optional routine which can be used to speed up long disk writes. Because MS-DOS uses a sophisticated disk allocation strategy with tables of free disk space kept on disk and only subsets of these tables in memory, long disk writes are likely to exhaust the tables in memory and cause unnecessary disk head movement to read in the new table sections. These disk head movements take time, and can cause time critical operations, such as continuous high-speed ADC to disk to fail.

The `SetWriteData` function writes one byte of data at the very end of the area you are about to write to and moves the disk head back to the start of the area, ready for your write. The act of writing at the end of the area causes the file to be extended to this new length (if it was shorter) and will load the disk tables into MS-DOS internal buffers. There is no guarantee that there will be room in memory for all the required buffers but experience shows that this speeds up subsequent writes, at the expense of a delay while memory is allocated. Please make sure that the `BUFFERS=` parameter in the MS-DOS system configuration file `CONFIG.SYS` is set to at least 20.

```
void SetWriteData(short handle, long byteOff, long bytes);
```

`handle` The file handle returned by `CreateCFSFile`.

`byteOff` The byte offset within the data section data at which to start writing.

`bytes` The number of bytes you intend to write.

Any errors are returned through the `FileError` function.

InsertDS

This function is used to close the current data section, and writes the data section header to the file. It can only be used on a file opened with `CreateCFSFile`. You should have written all the channel data you require, and set all the data section variables you want to change before you use this function.

```
short InsertDS(short handle, WORD dataSect, TSFlags flagSet);
```

`handle` The file handle returned by `CreateCFSFile`.

`dataSect` The data section you wish this data to become, or 0 to add this data section to the end of the file. Data sections are numbered from 1. It is an error to give a data section number which would result in undefined data sections. For example, if there were 27 data sections already in a file, it would be legal to request that the new data section be given the number 1 to 28, or 0 (equivalent to 28), but 29 or above would be illegal.

`flagSet` Flag variable in which each bit can be used as a flag. It is up to you to set the bits representing the flags you wish to be associated with the data section. If you have no interest in flags set this to `noFlags`. (A global variable of the right type provided for this purpose).

The function returns 0 if no error was detected, or a negative error code. Error codes are listed in the description of the `FileError` function..

RemoveDS

You can delete a data section from a file opened by `CreateCFSFile`, or by `OpenCFSFile` with `enableWr` set to 1. The deletion process simply unlinks the data section from the linked list of data sections, the disk space is not recovered.

```
void RemoveDS(short handle, WORD dataSect);
```

`handle` The file handle returned by `CreateCFSFile` or `OpenCFSFile`.

`dataSect` The data section to be removed.

If an error occurs, it is reported through the `FileError` function.

ClearDS

This deletes any data already written in the current data section in a file opened by `CreateCFSFile`. The deletion process resets the size of the unclosed data section to zero and resets the file size accordingly. The disk space is recovered.

```
short ClearDS(short handle);
```

`handle` The file handle returned by `CreateCFSFile`.

The function returns 0 if no error was detected, or a negative error code. See the `FileError` function (below) for a list of error codes.

SetVarVal

This function is used to set the value of specific file and data section variables. This function can be used with files opened by `CreateCFSFile`, or with files opened by `OpenCFSFile`.

When used with `CreateCFSFile` files, data section variables changed in the current data section stay changed for future data sections.

When used with `OpenCFSFile` files (which must be write enabled) the system moves to the data section header requested and changes the variable. This does not affect future data sections. The data section header and file header are NOT written back to disk immediately. The file header is only written back when the file is closed. The data section header is written back either when the file is closed or when a request is made to read a different data section. These delayed writes are for performance reasons to minimise disk accesses.

```
void SetVarVal(short handle, short varNo, short varKind,
               WORD dataSect, TpVoid varADS);
```

`handle` The file handle returned by `CreateCFSFile` or `OpenCFSFile` with `enableWr` set to 1.

`varNo` The file variable or data section variable number. Numbering starts at 0.

`varKind` Set to `FILEVAR` for a file variable or to `DSVAR` for a data section variable.

`dataSect` This parameter sets the data section to use. For files opened with `CreateCFSFile` set this to zero to use the current, unclosed, data section.

`varADS` A pointer to the variable holding the data to be written to the file. We use a void pointer to allow us to use a single function to deal with all the various data types we wish to pass. It is up to you to make sure you pass the correct data type.

If an error occurs it is reported through the `FileError` function.

SetComment

This function updates the file comment information in the file header. The file comment is normally set by `CreateCFSFile`, the `SetComment` function makes it possible to change the comment to reflect the information actually written, or any changes made.

```
void SetComment(short handle, TpStr comment);
```

`handle` The file handle returned by CreateCFSFile or `OpenCFSFile` with `enableWr` set to 1.

`comment` Pointer to the string containing the file comment. It can be up to 72 characters long. If the comment is longer than this it is truncated to 72 characters.

If an error occurs, it is reported through the `FileError` function.

CloseCFSFile

This function closes newly created files, and files opened for reading. If the file header or data section header of the current data section need to be saved, they are written to disk before the file is closed. Once the file is closed, the file handle becomes invalid.

```
short CloseCFSFile(short handle);
```

`handle` The file handle of the file to be closed.

The function returns 0 if no errors were detected or a negative error code. Error codes are listed in the description of the `FileError` function.

OpenCFSFile

This function is used to open an existing CFS file for reading or editing. If a file is opened in read only mode, any attempt to change the file will be regarded as an error and disallowed.

```
short OpenCFSFile(TpStr fName, short enableWr, short memTable);
```

`fName` A pointer to a string holding the name of the file to be opened. This can include a full drive and path.

`enableWr` Set this 1 to allow modifications to be made to the file with `SetVarVal`, `SetFileChan`, `SetDSChan`, `SetComment`, `RemoveDS`, `WriteData` and `DSFlags`. If this is set 0, no changes will be allowed.

`memTable` The CFS can speed up access to data sections, at the expense of 4 bytes of memory per data section, by holding the list of start positions of the data section headers in memory. Set this parameter to 1 to enable this feature otherwise to 0. If the table cannot fit in memory, a -8 error will be returned via `FileError` (out of memory) but processing will continue using the table on disk.

The function returns a positive file handle if no error was detected (or only error is inability to allocate memory for the table) or a negative error code if the file could not be opened. Error codes are listed in he description of the `FileError` function

GetGenInfo

After a file is opened with CreateCFSFile or OpenCFSFile, you can use this function to read back the file creation date and time, and the file comment.

```
void GetGenInfo(short handle, TpStr time, TpStr date,
                                                TpStr comment);
```

`handle` The file handle returned by `CreateCFSFile` or `OpenCFSFile`

`time` Pointer to a character array of at least 8 characters to receive the time of the file creation as hh:mm:ss. On return there is no `NULL` terminating character.

`date` Pointer to a character array of size at least 8 characters to receive the file creation date in the format dd/mm/yy. On return there is no `NULL` terminating character.

`comment` Pointer to a string to hold at least 72 characters (eg. of type `TComment`) to receive the file comment.

It is up to you to allocate sufficient space for the string returned. If your character arrays are too short memory will be overwritten.

If an error occurs it is reported through the `FileError` function.

GetFileInfo

This function is used to read back the characteristics of a file opened with `OpenCFSFile` or `CreateCFSFile`.

```
void GetFileInfo(short handle, TpShort channels, TpShort fileVars,
            TpShort DSVars, TpUShort dataSects);
```

`handle` The file handle returned by `OpenCFSFile` or `CreateCFSFile`.

`channels` Pointer to a short variable to which the number of data channels held in each data section of the file will be written.

`fileVars` Pointer to a short variable to which the number of file variables in the file will be written.

`DSVars` Pointer to a short variable to which the number of data section variables in the file will be written.

`dataSects` Pointer to a `WORD` variable for return of the number of data sections stored in the file.

If an error occurs it is reported through the `FileError` function.

GetVarDesc

This function returns the description of a particular file or data section variable in a CFS file opened with `OpenCFSFile` or `CreateCFSFile`.

```
void GetVarDesc(short handle, short varNo, short varKind,
       TpShort varSize, TpDType varType,TpStr units, TpStr about);
```

`handle` The file handle returned by `OpenCFSFile` or `CreateCFSFile`.

`varNo` The file or data section variable number.

`varKind` Set to `FILEVAR` for a file variable or to `DSVAR` for a data section variable.

`varSize` Pointer to a short variable for return of the maximum size, in bytes, of the data item. This is a fixed size for everything except `LSTR` data. For `LSTR` data this is returned as the length of the string which will hold the maximum number of characters, `n`, that can be saved in the variable. This will be `n+1` bytes. The CFS internally allows `n+2` bytes (1 for the string length, `n` for the maximum string and 1 for the `NULL` character on the end).

`varType` Pointer to a `TDataType` type variable to return the type of the data (`INT1`, `WRD1`, `INT2`, `WRD2`, `INT4`, `RL4`, `RL8`, `LSTR`).

`units` Pointer to a string to return the units. The string should be able to hold at least 8 characters to guarantee space for the longest allowed unit string (variable type `TUnits` is recommended.)

`about` Pointer to a string to return the variable description. The string should be able to hold at least 20 characters (variable type `TDesc` is recommended.)

If an error occurs it is reported through the `FileError` function.

GetVarVal

This function is used to get the value of a particular file or data section variable. It is up to the user to make sure that the variable into which the data is returned is large enough to accommodate the data passed back. This is particularly important with character strings.

```
void GetVarVal(short handle, short varNo, short varKind,
               WORD dataSect, TpVoid varADS);
```

`handle` The file handle returned by `OpenCFSFile` or `CreateCFSFile`.

`varNo` The file or data section variable number.

`varKind` Set to `FILEVAR` for a file variable or to `DSVAR` for a data section variable.

`dataSect` This sets the data section for which the data is required. For files opened with `CreateCFSFile` set this to zero to use the current, unclosed, data section.

`varADS` A pointer to a variable into which the data is returned. It is your responsibility to ensure that the data variable is large enough to accommodate the returned data.

Please note that the data can be returned into any type of variable. We have used a void pointer for simplicity. The effect of sidestepping the type checking is that it is your responsibility, rather than the compilers, to check that the destination of the returned data is large enough.

If your program misbehaves badly after a call to `GetVarVal` please check that you have not accidentally passed a pointer to a variable which is too small to hold the returned value.

If an error is detected it is reported through the `FileError` function.

GetFileChan

This function returns the constant information for a particular data channel. This routine may only be used with a file opened with `OpenCFSFile`. This routine is the logical inverse of `SetFileChan`.

```
void GetFileChan(short handle, short channel, TpStr chanName,
                 TpStr yUnits, TpStr xUnits, TpDType dataType,
                 TpDKind dataKind, TpShort spacing, TpShort other);
```

`handle` The file handle returned by `OpenCFSFile`.

`channel` The data channel within the data section for which data is required.

`chanName` Pointer to a string for return of the channel description string set in the `SetFileChan` call. String should be at least 21 characters long, use type `TDesc`.

`yUnits` Pointer to a string for return of the units in which this data is measured. This would be the y axis units label if the data were plotted. The string should be at least 9 characters long, use type `TUnits`

`xUnits` For `EQUALSPACED` and `SUBSIDIARY` data only. This is a pointer to a string for return of the units of the equal spaces between the data points. The string should be at least 9 characters long, use type `TUnits`.

`dataType` Pointer to the variable for return of the type of data stored in the channel. (`INT1`, `WRD1`, `INT2`, `WRD2`, `INT4`, `RL4`, `RL8` or `LSTR`).

`dataKind` Pointer to the variable for return of the way the data is stored on this channel (`EQUALSPACED`, `MATRIX` or `SUBSIDIARY`).

`spacing` Pointer to the variable for return of the number of bytes between successive data values, on this channel, on the disk.

`other` Pointer to the variable for return of the next channel in the matrix for matrix data or associated channel for `SUBSIDIARY` data. For a one dimensional matrix this returns the channel number. If there is no other channel this is returned as 0.

Please note that it is important that the strings to which the pointers point are of sufficient length for the data returned.

If an error is detected it is reported through the `FileError` function.

GetDSChan

This function reads the channel information for a particular data section for a file opened with `OpenCFSFile` and the channel information from the current data section for a file opened with `CreateCFSFile`. The values returned are those which are allowed to change between data sections. The first data section number is 1.

```
void GetDSChan(short handle, short channel, WORD dataSect,
               TpLong chOffset, TpLong points, TpFloat yScale,
               TpFloat yOffset, TpFloat xScale, TpFloat xOffset);
```

`handle` The file handle returned by `OpenCFSFile` or `CreateCFSFile`.

`channel` The data section data channel for which information is required.

`dataSect` The data section within the file for which information is required. If the file was opened with `CreateCFSFile` this parameter can be set to 0 for access to the current section.

`chOffset` Pointer to the variable for return of the offset in bytes from the start of the data section data to the first byte of data for this channel.

`points` Pointer to the variable for return of the number of data points stored on this channel.

`yScale` Pointer to he variable for return of the scaling factor to convert `INT1`, `WRD1`, `INT2`, `WRD2` and `INT4` data to the real `yUnits` for this channel.

`yOffset` Pointer to the variable for return of the offset to convert `INT1`, `WRD1`, `INT2`, `WRD2` and `INT4` data to the real yUnits for this channel. yUnits=data*`yScale` + `yOffset`

`xScale` Valid for `EQUALSPACED` and `SUBSIDIARY` data only. Pointer to the variable to return the x interval between data points on this channel. You must supply a variable here and for `xOffset`, even if this channel is `MATRIX` data.

`xOffset` Valid for `EQUALSPACED` and `SUBSIDIARY` data only. Pointer to the variable to return the x value of the first data point on this channel.

If an error is detected it is reported through the `FileError` function.

GetChanData

This is a very useful function which will fill a buffer with data from a single channel in a given data section. This function extracts data from the channel and takes care of any un-interleaving required. The result is a simple array of data in memory.

```
WORD GetChanData(short handle,short channel,WORD dataSect,
long pointOff,WORD points,TpVoid dataADS, long areaSize);
```

`handle` The file handle returned by `OpenCFSFile` or `CreateCFSFile`.

`channel` The data section data channel for which information is required.

`dataSect` The data section within the file for which information is required.

`pointOff` The data point in the channel at which to start transferring data. The first data point is number 0.

`points` The number of data points to transfer. 0 means transfer to the end of the channel or the end of the buffer.

`dataADS` A pointer to the destination buffer.

`areaSize` The size of the destination buffer in bytes. This is here as a check against a catastrophic error if you attempt to transfer too much data.

The return value of the function indicates the number of points transferred. If this value is zero either an error has occurred or your parameters specify 0 points. If the return value is zero it is NOT safe to assume nothing has been transferred. (eg. A read error could occur part way through the loop transferring data by a series of buffers.)

A positive return value means that the transfer was successful and says how many points were transferred, (this is not necessarily equal to `points`).

Any error detected is reported through the `FileError` function.

CFSFileSize

This function is used to get the size of a CFS file, this is particularly useful with a new file that is being written, as the disk directory will not have the correct size in that circumstance.

```
long CFSFileSize(short handle);
```

`handle` The file handle returned by `OpenCFSFile` or `CreateCFSFile`.

The function returns the size of the file in bytes, or a negative error code. Error codes are listed in the description of the `FileError` function.

GetDSSize

This function is used to get the size of a particular data section data area for old files, or for the current data section for new files (this is solely the disk space taken to hold the channel data, not the data section header).

```
long GetDSSize(short handle, WORD dataSect);
```

`handle` The file handle returned by `OpenCFSFile` or `CreateCFSFile`.

`dataSect` The data section within the file for which information is required. Set 0 for files opened with `CreateCFSFile` for future compatibility.

The function returns the size of the data sections data area in bytes, or a negative error code. Error codes are listed in the description of the `FileError` function.

ReadData

This function reads data from a data section. Like `WriteData`, this routine treats the data section data area as a random access memory. It is up to you to extract the channels from this area. If you want a single channel of data you may find it much simpler to use `GetChanData` which takes care of all the chores associated with getting the data.

```
short ReadData(short handle, WORD dataSect, long byteOff,
               WORD bytes, TpVoid dataADS);
```

`handle` The file handle returned by `OpenCFSFile` or `CreateCFSFile`.

`dataSect` The data section within the file from which data is to be read.

`byteOff` The byte offset within the data section data area at which to start reading.

`bytes` The number of bytes to read.

`dataADS` A pointer to the buffer region into which the data is transferred. It is your responsibility to make sure that the buffer region is large enough to hold the number of bytes requested.

The function returns 0 if no error was detected or a negative error code. Error codes are listed in the description of the `FileError` function.

AppendDS

This function is used to add a new data section of known size to the end of a CFS file opened with `OpenCFSFile`. If used on a file opened with `CreateCFSFile`, it acts just as `InsertDS`. Once the data section has been added to the file, you can use `SetDSChan` and `WriteData` to set up the channels and write the data. You can also use `SetVarVal` and `DSFlags` to set all the data section variables and the flags.

```
short AppendDS(short handle, long lSize, TSFlags flagSet);
```

`handle` The file handle returned by `OpenCFSFile` or `CreateCFSFile`.

`lSize` The size of the new data section, in bytes. This is the size of the channel data for the section, the CFS library will take care of data section headers and variables.

`flagSet` Flag variable in which each bit can be used as a flag. It is up to you to set the bits representing the flags you wish to be associated with the data section. If you have no interest in flags set this to `noFlags`. (A global variable of the right type provided for this purpose).

The function returns 0 if no error was detected, or a negative error code. Error codes are listed in the description of the `FileError` function..

DSFlagValue

This function returns a data section flag value corresponding to the the supplied number from 0 to 15.

```
WORD DSFlagValue(int nflag);
```

`nflag` Variable indicating the flag number in range 0 to 15.

The function returns the value of the flag if found, or 0 if an invalid flag number was supplied.

DSFlags

This function is used to get or set the flags associated with a data section within a CFS file. You may only set the flags if the `OpenCFSFile` parameter enabled writing or if the file was opened with `CreateCFSFile`.

```
void DSFlags(short handle,WORD dataSect, short setIt,
             TpFlags pflagSet);
```

`handle` The file handle returned by `OpenCFSFile`.

`dataSect` The data section within the file from which data is to be read or written.

`setIt` 1 to set the data section flags, 0 to read them.

`pflagSet` Pointer to the variable containing the flags. If setIt is 1 the flag variable is to be written to the data section. Otherwise the variable will hold the flags returned.

If an error occurs it is reported through the `FileError` function.

C`FS.H` defines constants `FLAG0` to `FLAG15`, which contains the bit field for each flag corresponding to the flag names in the Pascal `SET`. (See types defined in Using CFS from C).

To set or clear flag 0:

```
flags = FLAG0
flags = flags - FLAG0
```

To test flag0:

```
if ((flags & DSFlagValue(0)) != 0) printf("Flag is set");
```

CommitCFSFile

This function writes all changed headers to disk and then ensures that the current state of the file is correctly shown in the disk directory. It does not close the file. It is intended that `CommitCFSFile` would be called after every data write and `InsertDS`, or after every `InsertDS` call. This will ensure that all the data written up to the time of the last `CommitCFSFile` call is recoverable if a complete system failure such as a power cut occurs. If this function is used after some data writes, but not others, the file may not be recoverable.

```
short CommitCFSFile(short handle);
```

`handle` The file handle returned by `OpenCFSFile` or `CreateCFSFile`.

The function returns 0 if no error was detected or a negative error code. See the `FileError` function for a list of error codes.

FileError

This is the general error function used to collect information on errors which are not instantly fatal.

```
short FileError(TpShort handleNo, TpShort procNo, TpShort errNo);
```

`handleNo` Pointer to the variable for return of the handle number of the file in which the error was found, or in the case of an invalid handle number, the offending number. WARNING: Do not make the mistake of using the variable holding the current file handle, as it will be altered.

`procNo` Pointer to the variable for return of the function number in which the error was found.

`errNo` Pointer to a variable for return of the error code. See the list below.

The function return is 1 if an error was encountered since the last time the function was called, 0 if not.

The `FileError` function returns three numbers. They contain information on the first function error encountered since the function was last called. Subsequent calls will return 0 until a new error is encountered, although the values of the error can still be accessed while the function returns 0. The function numbers are:

```
1   SetFileChan      10  GetFileChan      19  WriteData
2   SetDSChan        11  GetDSChan        20  ClearDS
3   SetWriteData     12  DSFlags          21  CloseCFSFile
4   RemoveDS         13  OpenCFSFile      22  GetDSSize
5   SetVarVal        14  GetChanData      23  ReadData
6   GetGenInfo       15  SetComment       24  CFSFileSize
7   GetFileInfo      16  CommitCFSFile    25  AppendDS
8   GetVarDesc       17  InsertDS
9   GetVarVal        18  CreateCFSFile
```

The possible error codes returned are:

-1 No spare file handles.
-2 File handle out of range 0-2.
-3 File not open for writing.
-4 File not open for editing/writing.
-5 File not open for editing/reading.
-6 File not open.
-7 The specified file is not a CFS file.
-8 Unable to allocate the memory needed for the filing system data.
-11 Creation of file on disk failed (writing only).
-12 Opening of file on disk failed (reading only).
-13 Error reading from data file.

-14 Error writing to data file.
-15 Error reading from data section pointer file.
-16 Error writing to data section pointer file.
-17 Error seeking disk position.
-18 Error inserting final data section of the file.
-19 Error setting the file length.
-20 Invalid variable description.
-21 Parameter out of range 0-99.
-22 Channel number out of range
-24 Invalid data section number (not in the range 1 to total number of sections).
-25 Invalid variable kind (not 0 for file variable or 1 for DS variable).
-26 Invalid variable number.
-27 Data size specified is out of the correct range.
-30 to
-39 Wrong CFS version number in file

The CFS Unit

To use the CFS library you need to include the file CFS.TPU (for Turbo-Pascal programs) and declare that the program USES the CFS unit. A typical Turbo-Pascal program will start:

```
PROGRAM Typical(INPUT,OUTPUT);
USES CFS;
```

The file CFS.Tpu should be in your current directory for this to compile correctly. This will include all the definitions you need to use the CFS into your program.

Constants defined

These two constants are defined to select between file variables and data section variables in some of the variable related routines:

```
fileVar= 0;          {used to pass as a variable parameter}
DSVar  = 1;          {indicating a file or data section var}
```

These 8 constants represent the ordinal part of the `TDataType`:

```
int1   = 0;          {single byte data}
wrd1   = 1;          {single byte data}
int2   = 2;          {2 byte integer data}
wrd2   = 3;          {2 byte integer data}
int4   = 4;          {4 byte integer data}
rl4    = 5;          {4 byte floating point}
rl8    = 6;          {8 byte floating point}
lstr   = 7;
```

Types defined

The CFS unit defines 6 data types for use by your program. `TFlags` and `TSFlags` are used to make a set of 16 marker flags. This is a very efficient use of flags as they each occupy one bit of memory; all 16 flags occupy only 2 bytes. The flags can be set in the data section header and are usually used to mark sections for special treatment in a data processing program.

```
Tflags = (flag7,flag6,flag5,flag4,flag3,flag2,flag1,flag0,flag15,
             flag14,flag13,flag12,flag11,flag10,flag9,flag8);
TSFlags =    SET OF TFlags;

TDataType =  (int1..lstr);
```

`TDataType` is used to set the kind of data in a file variable or a data section variable and the data type for the data section channel data. This type occupies 1 byte of memory.

```
TCFSKind =  (equalSpaced, matrix, subsidiary);
```

`TCFSKind` is used to specify the data storage method used for channel data. Users from other languages will see `equalSpaced` = 0, `matrix` = 1 and `subsidiary` = 2.

```
TVarDesc  =  RECORD
             varDesc   : STRING[21];    {description of variable}
             vType     : TDataType;     {variable type, e.g. int2}
             zeroByte  : BYTE;       {for MS Pascal compatibility}
             varUnits  : STRING[9];     {the variable units}
             vSize     : INTEGER;       {size in bytes}
             END;
TSuperDesc= ARRAY[0..100] OF TVarDesc;  {to hold var descriptions}
```

These two types are used to set-up both the file and the data section variables. When a file is created the user decides what variables are to be stored. Each variable must be given a description, a type, units and a size in bytes. The size is only used in the case of a `lstr` variable (all the others have known, fixed sizes).

Compiler directives

The following compiler directives must be set in order to use the CFS unit.

```
{$N+,E+,V-}
```

The `$N+` and `$E+` cause an 8087 maths co-processor to be used if one is present and to be emulated if not. This allows us to use the IEEE format reals. The `$V-` stops the compiler from checking that strings are of identical type. See the Turbo Pascal reference manual for further information.

CreateCFSFile

This function is used to create a CFS file for writing. If a file exists of the same name it is truncated to zero length and then rebuilt. This is the first step in writing a CFS file.

```
FUNCTION CreateCFSFile(VAR fName, comment:STRING; blockSize:WORD;
             channels:INTEGER; VAR fileArray, DSArray:TSuperDesc;
             fileVars, DSVars:INTEGER):INTEGER;
```

`fName` The name of the file to be created for writing. This name can include a MSDOS path. The file name will be stripped from this and written into the file header.

`comment` The file comment can be up to 72 characters long. If the comment is longer than this it is truncated to 72 characters.

`blockSize` The `blockSize` parameter is used to optimise disk usage for your particular application. Disk systems store data physically in sectors of a fixed size. Data is always read and written in integral units of disk sectors. Operating systems (like MS-DOS) hide this from the user and let you use the disk as though you had access to individual bytes. Disk performance can be enhanced, at the cost of using more disk space, by rounding up the sizes of the file header, the data section header and the data section to a multiple of the sector size. Under MS-DOS, set this to 512 to round up to sectors (this means a minimum data section size of 1024 bytes) or to 1 for the most compact, but slower, file.

`channels` The number of channels of data in each data section.

`fileArray` This is an array of variable description records for the file variables. You must set this up before you call this function by filling in the variable description, type, units and size for every variable you have set.

`DSArray` An array of data section variable descriptions. You must set this up in the same way as `fileArray` before you call this function.

`fileVars` The number of file variables described in `fileArray` in the range 0-100.

`DSVars` The number of data section variables in `DSArray` in the range 0-100.

This routine allocates space for the file header and for a data section header in memory, and initialises all the variables to default values. All `int1`, `wrd1`, `int2`, `wrd2` and `int4` variables are set to 0. Real variables are set to 0.0 and strings are set to empty.

The data channels defined are all set to a default state (the default values are given in `SetFileChan` and `SetDSChan` below).

The function returns either a positive file handle, or a negative error code. The CFS system can have up to 3 data files open at any one time. This is an arbitrary limit built into the CFS library. It is a simple matter to rebuild the library for more files, at the cost of more memory used to hold file information.

SetFileChan

This procedure sets the fixed channel information for a single channel. This procedure must be used to describe each data channel before any data is written to the file.

```
PROCEDURE SetFileChan(handle, channel:INTEGER;
             VAR chanName, yUnits, xUnits:STRING;
             dataType:TDataType;dataKind:TCFSKind;
             spacing, other:INTEGER);
```

`handle` The file handle returned from `CreateCFSFile` or `OpenCFSFile`, used to identify the file. You can have up to three files open at a time.

`channel` The channel number to be set-up. You must use this routine to set the fixed information for all the channel numbers that you requested in `CreateCFSFile`. NB 1st channel is number 0.

`chanName` This is a string describing or naming the channel. Only the first 20 characters will be saved.

`yUnits` The y axis units for the channel. If the data is scaled then these units should be the units to use after scaling. Only the first 8 characters are saved.

`xUnits` The x axis units for `equalSpaced` data channels. If the data is scaled then these units should be the units to use after scaling. This is saved, but not assigned special meaning for matrix data. Only the first 8 characters are saved.

`dataType` The type of each element of the data saved on this channel. This parameter would be `int1`, `wrd1`, `int2`, `wrd2`, `int4`, `rl4`, `rl8` or `lstr`. The `lstr` type indicates text, not a character string.

`dataKind` Set to either `equalSpaced`, `matrix` or `subsidiary`. (The `lstr` data type would normally be a 1 dimensional matrix).

`spacing` The number of bytes between the start of the first data element and the second data element in the data section on disk. This is telling the system how you intend to store your data. For `int2` data stored with no other intervening data, this would be 2. If you had 2 channels of interleaved `int2` data, this would be 4. If you had 1 channel of `int2` data and one channel of `int4` data interleaved this would be 6.

`other` The next channel number, for `MATRIX` data. For `SUBSIDIARY` data, the master channel to which the subsidiary data refers. For `EQUALSPACED` data, the subsidiary channel associated with it if it has one, or zero. If the channel is a one dimensional matrix, set this to the current channel number.

Any errors detected in this procedure are returned via the FileError function, described below. You should use this routine with great care if your file was opened with `OpenCFSFile`.

A note on units

It is advisable always to use SI units for data in a CFS file, as this enables generalised programs to use the data more easily.

Default state after CreateCFSFile

If this routine is not called for a channel all the `STRING` variables are set empty, `dataType` is set to `int2`, `dataKind` to `equalSpaced`, `spacing` to 2 and other to 0.

SetDSChan

This procedure is used to set the channel information which can change between data sections. We anticipate that most uses of the CFS will have invariant channel information (i.e. each data section will have the same parameters), so there is no need to call this routine for every data section. Once a channel is set-up the values are remembered and this routine need only be called for changes to the parameters.

When writing a new file you must call `SetFileChan` before you use this routine for each channel. You should call `SetDSChan` for each channel before the first time you use `InsertDS`. When reading this routine can be used (with care) to change the fixed channel parameters for a specific data section. Remember you are responsible for setting sensible parameters with which to interpret your data.

```
PROCEDURE SetDSChan(handle,channel:INTEGER; dataSect:WORD;
                    chOffset, points:LONGINT;yScale,yOffset,
                    xScale,xOffset:REAL);
```

`handle` The file handle returned by `CreateCFSFile` or `OpenCFSFile`.

`channel` The data section channel number to use. Channel numbers start at 0.

`dataSect` This is ignored if the file was opened with `CreateCFSFile` (and should be set to 0 for future compatibility) and the current data section is used. For a file opened with `OpenCFSFile` this is the data section in which to change the channel information.

`chOffset` The byte offset from the start of the data section data to the first byte of this channel. Data is written with the `WriteData` function described below. It is entirely your responsibility to set the format of the data you write. All you are doing here is telling the CFS where to find the data for a particular channel.

`points` The number of data points for this channel.

`yScale` The `yScale` and `yOffset` parameters are used to convert data stored in integer and word formats (`int1`, `wrd1`, `int2`, `wrd2`, `int4`) to the real world units defined for this channel in `SetFileChan`. This means that we are allowing the user to change gain and offset between successive data blocks. For `rl4`, `rl8` and `lstr` please set this to 1.0.

`yOffset` Used with `yScale` to convert data to real world units. The calculation implied is that real units=data*`yScale` + `yOffset`. We ignore the offset for `rl4`, `rl8` and `lstr` channels.

`xScale` Used only for `equalSpaced` and `subsidiary` channels. This is the increment in the "x" position between consecutive data points. Please set to 0.0 for `matrix` channels.

`xOffset` Used only for `equalSpaced` and `subsidiary` channels. This is the "x" position of the first point on the channel. Set to 0.0 for `matrix` channels.

Any errors detected in this procedure are returned via the FileError function, described below. If your file was opened with OpenCFSFile you should use this with care.

Default state after CreateCFSFile

If you do not call `SetDSChan` for a particular channel then `chOffset`, `points`, `yOffset` and `xOffset` are set to 0 and `yScale` and `xScale` are set to 1.

WriteData

Write the channel data into the current data section. This can be used on files opened with both `CreateCFSFile` (for writing) and `OpenCFSFile` (as long as the `enableWr` flag is set). If you are writing to a file opened with `OpenCFSFile` you will get an error if your write would extend beyond the data area originally written.

```
FUNCTION WriteData(handle:INTEGER; dataSect:WORD;
       byteOff:LONGINT; bytes:WORD;dataADS:POINTER):INTEGER;
```

`handle` The handle of the file to which to write the data.

`dataSect` If the file is opened with `CreateCFSFile` this item is ignored and data is written to the current data section, you should set this parameter to 0 for future compatibility. For files opened with `OpenCFSFile` this is the data section to be updated, and must already exist.

`byteOff` The byte offset within the data section data at which to start writing. You can write to any byte offset, but you should be aware that writes of integral numbers of disk sectors starting at a disk sector boundary are faster than random writes. See the `blockSize` parameter to the `CreateCFSFile` routine.

`bytes` The number of bytes of data to write.

`dataADS` The address of the data buffer to write to the file.

`WriteData` returns either 0 if no error was detected or a negative error code.

SetWriteData

This is an optional routine which can be used to speed up long disk writes. Because MS-DOS uses a sophisticated disk allocation strategy with tables of free disk space kept on disk and only subsets of these tables in memory, long disk writes are likely to exhaust the tables in memory and cause unnecessary disk head movement to read in the new table sections. These disk head movements take time, and can cause time critical operations, such as continuous high-speed ADC to disk to fail.

The `SetWriteData` routine writes one byte of data at the very end of the area you are about to write to and moves the disk head back to the start of the area, ready for your write. The act of writing at the end of the area causes the file to be extended to this new length (if it was shorter) and will load the disk tables into MS-DOS internal buffers. There is no guarantee that there will be room in memory for all the required buffers but experience shows that this speeds up subsequent writes, at the expense of a delay while memory is allocated. Please make sure that the `BUFFERS=` parameter in the MS-DOS system configuration file `CONFIG.SYS` is set to at least 20.

```
PROCEDURE SetWriteData(handle:INTEGER; byteOff,bytes:LONGINT);
```

`handle` The file handle returned by CreateCFSFile.

`byteOff` The byte offset within the data section data at which to start writing.

`bytes` The number of bytes you intend to write.

Any errors are returned through the FileError function.

InsertDS

This function is used to close the current data section, and writes the data section header to the file. It can only be used on a file opened with `CreateCFSFile`. You should have written all the channel data you require, and set all the data section variables you want to change before you use this function.

```
FUNCTION InsertDS(handle:INTEGER; dataSect:WORD;
            flagSet:TSFlags):INTEGER;
```

`handle` The file handle returned by `CreateCFSFile`.

`dataSect` The data section you wish this data to become, or 0 to add this data section to the end of the file. Data sections are numbered from 1. It is an error to give a data section number which would result in undefined data sections. For example, if there were 27 data sections already in a file, it would be legal to request that the new data section be given the number 1 to 28, or 0 (equivalent to 28), but 29 or above would be illegal.

`flagSet` The set of 16 flags which you wish to associate with this data section. If you have no interest in flags then set this to `[]`, the empty set.

The function returns 0 if no error was detected, or a negative error code. See the `FileError` function (below) for a list of error codes.

RemoveDS

You can delete a data section from a file opened by `CreateCFSFile`, or by `OpenCFSFile` with `enableWr` set `True`. The deletion process simply unlinks the data section from the linked list of data sections, the disk space is not recovered.

```
PROCEDURE RemoveDS(handle:INTEGER; dataSect:WORD);
```

`handle` The file handle returned by `CreateCFSFile` or `OpenCFSFile`.

`dataSect` The data section to be removed.

If an error occurs, it is reported through the `FileError` function.

ClearDS

This deletes any data already written in the current data section in a file opened by `CreateCFSFile`. The deletion process resets the size of the unclosed data section to zero and resets the file size accordingly. The disk space is recovered.

```
PROCEDURE ClearDS(handle:INTEGER);
```

`handle` The file handle returned by CreateCFSFile.

The function returns 0 if no error was detected, or a negative error code. See the `FileError` function (below) for a list of error codes.

SetVarVal

This procedure is used to set the value of specific file and data section variables. This procedure can be used with files opened by `CreateCFSFile`, or with files opened by `OpenCFSFile`, but the behaviour is different in each case.

When used with `CreateCFSFile` files, any data section variable changed stays changed for future data sections. The data section parameter is ignored and should be set to 0 for future compatibility. The current data section is always used.

When used with `OpenCFSFile` files (which must be write enabled) the system moves to the data section header requested and changes the variable. This does not affect future data sections. The data section header and file header are NOT written back to disk immediately. The file header is only written back when the file is closed. The data section header is written back either when the file is closed or when a request is made to read a different data section. These delayed writes are for performance reasons to minimise disk accesses.

```
PROCEDURE SetVarVal(handle,varNo,varKind:INTEGER; dataSect:WORD;
            varADS:POINTER);
```

`handle` The file handle returned by `CreateCFSFile` or `OpenCFSFile` with `enableWr` set `True`.

`varNo` The file variable or data section variable number. Numbering starts at 0.

`varKind` Set to 0 for a file variable or to 1 for a data section variable. The two constants `fileVar` and `DSVar` are provided for this purpose.

`dataSect` Ignored for file variables or if the file was opened by `CreateCFSFile` (please set this to 0 for future compatibility in these cases). This parameter is the data section to use for files opened with `OpenCFSFile`.

`varADS` The address of the variable holding the data to be written to the file. We use a `POINTER` type variable to allow us to use a single procedure to deal with all the various data types we wish to pass. It is up to you to make sure you pass the correct data type.

If an error occurs, it is reported through the `FileError` function.

SetComment

This updates the file comment information in the file header. The file comment is normally set by `CreateCFSFile`, the `SetComment` procedure makes it possible to change the comment to reflect the information actually written, or any changes made.

```
PROCEDURE SetComment(handle:INTEGER; VAR comment: STRING);
```

`handle` The file handle returned by `CreateCFSFile` or `OpenCFSFile` with enableWr set to `True`.

`comment` The file comment, can be up to 72 characters long. If the comment is longer than this it is truncated to 72 characters.

If an error occurs, it is reported through the `FileError` function.

CloseCFSFile

This procedure closes newly created files, and files opened for reading. If the file header or data section header of the current data section need to be saved, they are written to disk before the file is closed. Once the file is closed, the file handle becomes invalid.

```
FUNCTION CloseCFSFile(handle:INTEGER):INTEGER;
```

`handle` The file handle of the file to be closed.

The function returns 0 if no errors were detected or a negative error code. Error codes are listed in the description of the `FileError` function.

OpenCFSFile

This function is used to open an existing CFS file for reading or editing. If a file is opened in read only mode, any attempt to change the file will be regarded as an error and disallowed.

```
FUNCTION OpenCFSFile(VAR fName:STRING; enableWr,
             memTable:BOOLEAN):INTEGER;
```

`fName` The name of the file to be opened. This can include a full drive and path.

`enableWr` Set this `True` to allow modifications to the data file with `SetVarVal`, `SetFileChan`, `SetDSChan`, `SetComment`, `RemoveDS`, `WriteData` and `DSFlags`. If this is set `False`, no changes will be allowed.

`memTable` The CFS can speed up access to data sections, at the expense of 4 bytes of memory per data section, by holding the list of start positions of the data section headers in memory. Set this parameter to `True` to enable this feature. If the table cannot fit in memory a -8 error will be returned via `FileError` (out of memory) but processing will continue using the table on disk.

The routine returns a positive file handle if no error was detected or a negative error code if the file could not be opened.

GetGenInfo

After a file is opened with `CreateCFSFile` or `OpenCFSFile`, you can use this routine to read back the file creation date and time, and the file comment.

```
PROCEDURE GetGenInfo(handle:INTEGER;VAR time,date,comment:STRING);
```

`handle` The file handle returned by `CreateCFSFile` or `OpenCFSFile`

`time` An `STRING` variable of at least 8 characters to receive the time of the file creation as hh:mm:ss.

`date` An `STRING` variable of size at least 8 characters to receive the file creation date in the format dd/mm/yy.

`comment` An `STRING` variable of size at least 72 characters to receive the file comment.

If any `STRING` variable is too short to hold the entire string returned, the string will be truncated. This is not considered an error and will not overwrite memory.

GetFileInfo

This procedure is used to read back the characteristics of a file opened with `OpenCFSFile` or `CreateCFSFile`.

```
PROCEDURE GetFileInfo(handle:INTEGER;VAR channels,
      fileVars,DSVars:INTEGER; VAR dataSects:WORD);
```

`handle` The file handle returned by `OpenCFSFile` or `CreateCFSFile`.

`channels` The number of data channels held in each data section of the file.

`fileVars` The number of file variables in the file.

`DSVars` The number of data section variables in the file.

`dataSects` Returned as the number of data sections stored in the file.

GetVarDesc

This procedure returns the description of a particular file or data section variable in a CFS file opened with `OpenCFSFile` or `CreateCFSFile`.

```
PROCEDURE GetVarDesc(handle,varNo,varKind:INTEGER;
            VAR varSize:INTEGER; VAR varType:TDataType;
            VAR units,about:STRING);
```

`handle` The file handle returned by `OpenCFSFile` or `CreateCFSFile`.

`varNo` The file or data section variable number.

`varKind` Set to 0 for a file variable or to 1 for a data section variable. The two constants `fileVar` and `DSVar` are provided for this purpose.

`varSize` Returned as the maximum size, in bytes, of the data item. This is a fixed size for everything except `lstr` data. For `lstr` data this is returned as the size of the `STRING[n]` variable required to hold the longest string that could be saved in the variable. This will be `n+1` bytes. The CFS internally allows `n+2` bytes (1 for the string length, `n` for the maximum string and 1 for a zero character on the end).

`varType` Returned as the type of the data (`int1`, `wrd1`, `int2`, `wrd2`, `int4`, `rl4`, `rl8`, `lstr`).

`units` Returned as an `STRING` holding the units. The string should be at least 8 characters long to guarantee space for the longest allowed unit string.

`about` Returned as an `STRING` holding the variable description. If the `STRING` is not long enough, the returned description will be truncated.

GetVarVal

This procedure is used to get the value of a particular file or data section variable. It is up to the user to make sure that the variable into which the data is returned is large enough to accommodate the data passed back. This is particularly important with character strings.

```
PROCEDURE GetVarVal(handle,varNo,varKind:INTEGER; dataSect:WORD;
            varADS:POINTER);
```

`handle` The file handle returned by `OpenCFSFile` or `CreateCFSFile`.

`varNo` The file or data section variable number.

`varKind` Set to 0 for a file variable or to 1 for a data section variable. The two constants `fileVar` and `DSVar` are provided for this purpose.

`dataSect` This is ignored for file variable data and for files opened with `CreateCFSFile`, where it should be set to 0 for compatibility with future revisions. For data section variables it is the data section for which the data

is required for files opened with `OpenCFSFile`. For files opened with `CreateCFSFile` the relevant data section will always be the current one..

`varADS` The address of the data item into which the data is returned. It is your responsibility to ensure that the data area is large enough to accommodate the returned data.

Please note that the data can be returned into any type of variable. We have used the `POINTER` type to avoid the need for a separate procedure for each data type or the need for a complicated `RECORD CASE TDataType... END` data type. The effect of sidestepping the strong type checking of Pascal is that it is your responsibility, rather than the compilers, to check that the destination of the returned data is large enough.

If your program misbehaves badly after a call to `GetVarVal` please check that you have not accidentally passed the address of a variable which is too small to hold the returned value.

GetFileChan

This procedure returns the constant information for a particular data channel. This routine may only be used with a file opened with `OpenCFSFile`. This routine is the logical inverse of `SetFileChan`.

```
PROCEDURE GetFileChan(handle,channel:INTEGER;
            VAR chanName,yUnits,xUnits:STRING;
            VAR dataType:TDataType;
            VAR dataKind:TCFSKind; VAR spacing,other:INTEGER );
```

`handle` The file handle returned by `OpenCFSFile` or `CreateCFSFile`.

`channel` The data channel within the data section for which data is required.

`chanName` Returned with the channel description string set in the `SetFileChan` call.

`yUnits` The units in which this data is measured. This would be the y axis units label if the data were plotted.

`xUnits` For `equalSpaced` and `subsidiary` data only. This is the units of the equal spaces between the data points.

`dataType` Returned as the type of data stored in the channel. (`int1`, `wrd1`, `int2`, `wrd2`, `int4`, `rl4`, `rl8` or `lstr`).

`dataKind` Returned as the way the data is stored on this channel (`equalSpaced`, `matrix` or `subsidiary`).

`spacing` Returned as the number of bytes between successive data values, on this channel, on the disk.

`other` Returned as the next channel in the matrix for `matrix` data or associated channel for `subsidiary` data. For a one dimensional matrix this returns the channel number. When not used it is set to 0.

GetDSChan

This procedure reads the channel information for a particular data section for a file opened with `OpenCFSFile` and the channel information from the current data section for a file opened with `CreateCFSFile`. The values returned are those which are allowed to change between data sections. The first Data Section number is 1.

```
PROCEDURE GetDSChan(handle,channel:INTEGER; dataSect:WORD;
                    VAR chOffset,points:LONGINT;
                    VAR yScale,yOffset,xScale,xOffset:REAL);
```

`handle` The file handle returned by `OpenCFSFile` or `CreateCFSFile`.

`channel` The data section data channel for which information is required.

`dataSect` The data section within the file for which information is required for files opened with `OpenCFSFile`. If the file was opened with CreateCFsFile this parameter is ignored and should be set to 0 for future compatibility.

`chOffset` Returned as the offset in bytes from the start of the data section data to the first byte of data for this channel.

`points` Returned as the number of data points stored on this channel.

`yScale` Returned as the scaling factor to convert `int1`, `wrd1`, `int2`, wrd2 and `int4` data to the real `yUnits` for this channel.

`yOffset` Returned as the offset to convert `int1`, `wrd1`, `int2`, `wrd2` and `int4` data to the real `yUnits` for this channel. `yUnits=data*yScale+yOffset`

`xScale` Valid for `equalSpaced` and `subsidiary` data only. Returned as the x interval between data points on this channel. You must supply a variable here and for `xOffset`, even if this channel is `matrix` data.

`xOffset` Valid for `equalSpaced` and `subsidiary` data only. Returned as the x value of the first data point on this channel.

GetChanData

This is a very useful routine which will fill a buffer with data from a single channel in a given data section. This routine extracts data from the channel and takes care of any un-interleaving required. The result is a simple array of data in memory.

```
FUNCTION GetChanData(handle, channel:INTEGER;
             dataSect:WORD; pointOff:LONGINT;
             numPoints:WORD; dataAds:POINTER; buffSize:WORD):WORD;
```

`handle` The file handle returned by `OpenCFSFile` or `CreateCFSFile`.

`channel` The data section data channel for which information is required.

`dataSect` The data section within the file for which information is required.

`pointOff` The data point in the channel at which to start transferring data. The first data point is number 0.

`numPoints` The number of data points to transfer. 0 means transfer to the end of the channel or the end of the buffer.

`dataAds` The address of the destination buffer.

`buffSize` The size of the destination buffer in bytes. This is here as a check against a catastrophic error if you attempt to transfer too much data.

The function returns the number of points actually read. Errors are reported through `FileError`.

GetDSSize

This function is used to get the size of a particular data section data area for old files, or for the current data section for new files (this is solely the disk space taken to hold the channel data, not the data section header).

```
FUNCTION GetDSSize(handle:INTEGER; dataSect:WORD):LONGINT;
```

`handle` The file handle returned by `OpenCFSFile` or `CreateCFSFile`.

`dataSect` The data section within the file for which information is required. Set 0 for files opened with `CreateCFSFile` for future compatibility.

The function returns the size of the data sections data area in bytes, or a -ve error code.

ReadData

This routine reads data from a data section. Like `WriteData`, this routine treats the data section data area as a random access memory. It is up to you to extract the channels from this area. If you want a single channel of data you may find it much simpler to use `GetChanData` which takes care of all the chores associated with getting the data.

```
FUNCTION ReadData(handle:INTEGER; dataSect:WORD;
            byteOff:LONGINT; bytes:WORD;dataAds:POINTER):INTEGER;
```

`handle` The file handle returned by `OpenCFSFile` or `CreateCFSFile`.

`dataSect` The data section within the file from which data is to be read.

`byteOff` The byte offset within the data section data area at which to start reading.

`bytes` The number of bytes to read.

`dataAds` The address of the buffer region into which the data is transferred. It is your responsibility to make sure that the buffer region is large enough to hold the number of bytes requested.

The routine returns 0 if no error was detected or a negative error code.

DSFlags

This procedure is used to get or set the flags associated with a data section within a CFS file opened with `OpenCFSFile`. You should use `InsertDS` to set the block flags for a file opened with `CreateCFSFile`. You may only set the flags if the `OpenCFSFile` enabled writing.

```
PROCEDURE DSFlags(handle:INTEGER; dataSect:WORD; setIt:BOOLEAN;
             VAR flagSet:TSFlags);
```

`handle` The file handle returned by `OpenCFSFile`.

`dataSect` The data section within the file from which data is to be read.

`setIt` `True` to set the data section flags, `False` to read them.

`flagSet` If set is `True`, the flags to write to the data section. Otherwise a variable to hold the flags returned.

CommitCFSFile

This function writes all changed headers to disk and then ensures that the current state of the file is correctly shown in the disk directory. It does not close the file. It is intended that `CommitCFSFile` would be called after every data write and `InsertDS`, or after every `InsertDS` call. This will ensure that all the data written up to the time of the last `CommitCFSFile` call is recoverable if a complete system failure such as a power cut occurs. If this function is used after some data writes, but not others, the file may not be recoverable.

```
FUNCTION CommitCFSFile(handle:INTEGER):INTEGER;
```

handle The file handle returned by `OpenCFSFile` or `CreateCFSFile`.

The function returns 0 if no error was detected or a negative error code. See the `FileError` function for a list of error codes.

FileError

This is the general error function used to collect information on errors which are not instantly fatal.

```
FUNCTION FileError(VAR handleNo, procNo, errNo:INTEGER):BOOLEAN;
```

`handleNo` Returned with the handle number of the file in which the error was found, or in the case of an invalid handle number, the offending number. WARNING: Do not make the mistake of passing the current file handle in as it will be altered.

`procNo` Returned with the procedure number in which the error was found.

`errNo` Returns the error code. See the list below.

The `FileError` function returns three numbers. They contain information on the first procedure error encountered since the function was last called. Subsequent calls will return `False` until a new error is encountered, although the values of the error can still be accessed while the function returns `False`. The procedure numbers are:

1	`SetFileChan`	9	`GetVarVal`	17	`InsertDS`
2	`SetDSChan`	10	`GetFileChan`	18	`CreateCFSFile`
3	`SetWriteData`	11	`GetDSChan`	19	`WriteData`
4	`RemoveDS`	12	`DSFlags`	20	`ClearDS`
5	`SetVarVal`	13	`OpenCFSFile`	21	`CloseCFSFile`
6	`GetGenInfo`	14	`GetChanData`	22	`GetDSSize`
7	`GetFileInfo`	15	`SetComment`	23	`ReadData`
8	`GetVarDesc`	16	`CommitCFSFile`		

The possible error codes returned are:

-1 No spare file handles.
-2 File handle out of range 0-2.
-3 File not open for writing.
-4 File not open for editing/writing.
-5 File not open for editing/reading.
-6 File not open.
-7 The specified file is not a version 2 filing system file.
-8 Unable to allocate the memory needed for the filing system data.
-11 Creation of file on disk failed (writing only).
-12 Opening of file on disk failed (reading only).
-13 Error reading from data file.
-14 Error writing to data file.
-15 Error reading from data section pointer file.
-16 Error writing to data section pointer file.
-17 Error seeking disk position.
-18 Error inserting final data section of the file.
-19 Error setting the file length.
-20 Invalid variable description.
-21 Parameter out of range 0-99.
-22 Channel number out of range
-24 Invalid data section number (not in the range 1 to total number of sections).
-25 Invalid variable kind (not 0 for file variable or 1 for DS variable).
-26 Invalid variable number.
-27 Data size specified is out of the correct range.
-30 to
-39 Wrong CFS version number in file

Appendix 1: Changes from CFS version 1

Changes from CFS version 1

This section is for programmers who have used CFS version 1 and who are now converting, or considering converting their programs to version 2. We have changed the names of all procedures and functions to avoid any possible confusions of names between the old and the new. We have reduced the number of procedures and functions to simplify the programmer interface, but we have not lost any functionality. We have converted several programs ourselves; notes and warnings are given at the end of this Appendix.

Reasons for the changes

The changes are not for fun! Remember that CED will have more programs to change than anyone else. Version 1 of CFS made setting up of files needlessly difficult. There were also several features missing (such as a yOffset for channel data) that in retrospect are essential for any complete data storage system. CFS files were also not space efficient where the user needed many small data sections because of the rounding up of both the data section header and the data section itself to 512 byte boundaries. All data reads and writes had to be to 512 byte blocks which was often inconvenient (though fast).

We have preserved what was good from the old, at the same time adding new ideas, and responding to the constructive criticism (for which we are most grateful) of users and potential users.

We have left some space in the various data structures to allow us to implement new features, should they be needed, without any change to existing files.

We have also made gestures in the data structures to make it much easier to implement the CFS library in other languages. The definitions in this manual are for Turbo-Pascal under MS-DOS but the underlying structure can be implemented from almost any language. (We now provide a Microsoft C version with the functions having the same names as the functions and procedures of the Borland Turbo Pascal version.)

Main changes

The main changes are:

1. The last character of file marker (`CEDFILE!` in version 1) becomes the version marker, i.e. `!`=version 1 , `"`=version 2 etc. The new version is version 2 with the file marker `CEDFILE"`.

2. The terminology is made consistent. The term block is no longer used as it is confusing. The documentation is (we hope) much improved.

3. Procedure calls tidied up in general. Of special note:

a) `ChanInfo1` and `ChanInfo2` are now `GetFileInfo` and `GetDSChan`, separating the reading of constant and varying information.

b) A new function `GetChanData` allows the user to read back a single channel of data using the channel information stored in the file.

c) The function `InsertDS`, replacing `InsertBlock` and `CloseBlock` uses the number of the data section BEFORE WHICH the data is to be stored.

4. You can now edit all the data in an existing file, but you cannot change its structure (number of variables, number of channels, or length of data in a channel). File writing is essentially unchanged but there are two modes of reading according to the `enableWr` parameter in `OpenCFSFile`. An existing CFS file can be opened either for reading only or for reading and editing. Data sections can be re-written (edited) provided the size of the data is not increased. Variable values can be altered and

deletion of data sections is allowed. The reading only mode guarantees that the file will not be altered, it also increases speed and reduces memory requirements.

5. Access via the block pointer table is now transparent to the user and always enabled. When a file is open for reading the table is stored on disk to reduce memory requirements, or in memory to increase speed, according to a parameter set when opening the file. When editing a file the pointer table is copied to a temporary file to allow the size of the file to be altered. If a program is then terminated without closing the file the table will not be restored to the end of the file. However, the filing system will detect this condition the next time the file is opened and will recreate the table.

6. The file and data section variable numbers now start at zero. File variable zero should be used as the file's label, to identify the source of the data. This will allow specialised analysis programs to know if the data is useful to them. Please set the variable type to `int2` (`INT2` if you are using C) and its value to 100 times the program revision level (version 2.03 would be stored as 203). Set the units of the variable to the program name in upper case (for example `PATCH` or `SIGAVG`) and the variable description to a longer description of the source of the data.

7. Variable descriptions are now defined on opening a file for writing. They are passed in as `TSuperDesc` in the parameters of `CreateCFSFile`. Variable values are accessed by reference. This places the responsibility of type checking on the user. In particular take care when reading variables from the file (e.g. don't read an `int4` back into an `int2`).

8. Data types have been rationalised. Both variable types and data types use `TDataType` which remains unchanged apart from `CHAR` data which is replaced by lstr. The `lstr` type is the Turbo-Pascal `STRING[n]`. It's maximum size (`n`) is defined in the variable description and it's actual size is contained in the first byte of the variable. Channel data (as opposed to variables) of this type is treated as simple text with lines terminated by CR LF and of whatever size as determined by the points parameter stored in the channel information. This allows text to be stored in a CFS file.

9. Two kinds of data are allowed: `equalSpaced` and `matrix`. `matrix` data for a channel referencing itself is equivalent to the old discrete kind. In later version 2 releases the `subsidiary` data kind has been added.

10. Data storage as 512 byte blocks is optional. The disk block size must be specified when creating a file and so a value of 1 will allow access to the nearest byte giving much more compact files, but at the expense of slightly slower disk access.

11. Scale and offset values are provided for both X and Y values to allow easier gain and offset changes.

12. A separate program, `TRANSCFS` transfers files from version 1 format to version 2.

13. Less important filing functions have been made into procedures (`void` return in C) and a new error function informs the user of any errors occurring in these procedures.

14. Nulls are placed at the end of all strings on disk to make life easier for C programmers. (The C version takes care of the storage length byte of the Pascal type Strings.)

15. Channel numbers are allocated in sequence from 0. Previously the channel number was a freely chosen identifier.

Tips and pitfalls when converting

At the time of writing this document we are gaining experience with the new CFS by converting programs to use it. In general, programs get smaller when the conversion is done and are easier to understand. Some points to watch out for are:

1. Variable descriptions must now be set-up before you create a new file. This is now done by filling in an array of records (structures in C version). Each record describes one variable. Previously a routine was called to describe each variable, and as variables all occupied the same space it was possible to decide upon the type of a variable after a file was created. We know of no program which took advantage of this, so no-one should be inconvenienced.

2. You will need to request one more file and data variable to allow for the fact that they now start from 0 (this is to allow file variable 0 to be used as a source identifier).

3. `InsertDS`, which replaces `InsertBlock` and `CloseBlock`, now uses the position passed as the data section number of the new data section, not the block number (in the version 1 terminology) of the block it is to be placed after. Also note that data sections are numbered from 1, not 0.

4. Be careful reading variables which are now returned using a Type escape, so it is your responsibility to check type compatibility. For example, do not read an `int4` (`INT4`) variable into a space which is passed as the `ADS` of an `INTEGER` (`short *`). This change was made to greatly simplify and shorten code.

5. Non-critical errors are now reported through `FileError`. There is now no excuse for not checking errors from the more important functions.

Appendix 2: Examples

Examples

The distribution disk for the CFS holds the Turbo Pascal and Microsoft C implementations of the CFS libraries together with some examples plus the version 1 to version 2 data update program. At the time of writing there are two example programs (both have Turbo Pascal and C versions):

SimpleR This is an example of how to read and extract data from a CFS file. The program opens the CFS file and displays the file creation date, time and comment, plus the file variables. For each data section it lists the data section variables followed by the first few values from each channel in the data section.

SimpleW This is an example of creating a CFS file. It demonstrates how to set up the file and data section variables, and writes several data sections, each with varying values of some data section variables, and also modifies some channel parameters. Two fixed length interleaved channels of data are saved to disk for each data section.

Building the examples

The examples programs contain the instructions for compiling and linking at the start.

Pascal version

There are four files used to define the Turbo-Pascal CFS implementation:

`CFS.PAS`	The Pascal source of the CFS library.
`CFS.TPU`	The object file produced by compiling `CFS.PAS`.
`MSDOSLIB.PAS`	Source of the `MSDOS` library used by the Turbo Pascal implementation of CFS. This library is written and the copyright owned by CED.
`MSDOSLIB.TPU`	The object file produced by compiling `MSDOSLIB.PAS`.

A typical program is compiled and linked as follows:

```
C>TPC /m typical
```

C version

There are 8 files used to define the MS-C CFS implementation for DOS and Windows.

`CFS.H`	The `header` file containing the constant, type and function definitions you will need to use the CFS functions.
`CFS.C`	The source `code` of the CFS functions.
`MACHINE.H`	Included at the start of `CFS.H` header file, it contains definitions needed to make Macintosh DOS and Windows sources compatible.
`CFS.OBJ`	The DOS object file produced by compiling `CFS.C`. This was compiled with for the large memory model and floating point emulation libraries.
`CFS16.LIB`	The library module for the 16-bit Windows DLL.
`CFS16.DLL`	The dynamic-link library file for 16-bit Windows applications.
`CFS32.LIB`	The library module for the 32-bit Windows DLL.
`CFS32.DLL`	The dynamic-link library file for 32-bit Windows applications.

linking for DOS

A typical program for DOS is linked as follows:

```
C>LINK typical+...+CFS,typical.exe,,LLIBCE
```

`LLIBCE` is the name of the appropriate large model, floating point emulation Microsoft C library. You may also require the linker option `/ST:20000`

linking for Windows

A typical program for 16-bit Windows is linked as follows:

```
C>LINK typical,,,libw mlibcew cfs16 commdlg,typical.def
```

More typically, you will be compiling and linking within a development environment, in which case you have to ensure that you link with `CFS16.LIB` or `CFS32.LIB` as appropriate.

Appendix 3: TRANSCFS update program

The TRANSCFS update program

The TRANSCFS program converts CFS version 1 data files to the equivalent format under version 2. All variable numbers are maintained, but file variable 0 is added holding the source of the data and data section variable 0 is added holding an integer value which is set to 0. The program would be used to convert data files from programs which have been upgraded from version 1 to version 2. This presupposes that the conversion of the program has maintained all the variables and channels.

Please note that CFS version 1 allowed a free choice of channel numbers and allocated space on disk in the order in which the channels were described to the system. Version 2 uses channels numbered from 0 in sequence. The translation program replaces the previous channel number with the sequence number. Most programs used sequenced channels in any case, but if your program used non-consecutive channel numbers to identify CFS file channels you should be aware of this change.

Using TRANSCFS

The program is driven with command line arguments to allow you to use a batch file to convert a large number of files. The syntax is:

```
C>TRANSCFS source dest description identifier revision
```

`source` The full file name (can include a path) of the CFS version 1 file to be converted.

dest The full file name (can include a path) of the CFS version 2 file to be created.

`identifier` Either a single word, or string enclosed in single quotes, which identifies the source of the data, for example 'PATCH'. Only the first 8 characters are used.

`about` Either a single word, or a string enclosed in single quotes, which describes the data source. Only the first 20 characters are used.

`revision` An integer which is the source program revision. This should be 100 times the major revision plus the minor revision, so if the program is version 2.03 you should set 203.

The only required parameters are the two file names, the rest can be omitted, in order, from the right. Examples are:

```
C>TRANSCFS name0aa.dat name0aa.v02 PATCH 'Continuous data' 307
C>TRANSCFS \EPC\EXAMPLE.DAT D:\NEW\EXAMPLE.DAT 'ID 27'
```

Errors

Most transfer errors reported will correspond to the version 2 error code, although a few retain the old format. In practice, provided the old file is not corrupt the only likely problem is a disk access error, such as a full disk.

Index