CFHT FITS Handling Library
Revision 1.4

Sidik Isani <isani@cfht.hawaii.edu>

2001 April 8th
Last revised: 2013 December 19th

The information here is available as HTML. The URL is: http://software.cfht.hawaii.edu/libfh/

Abstract

This library of C functions can be included in your code to simplify the task of manipulating FITS data in either basic FITS format, or Multi-Extension FITS (MEF). Special features of the library are:

- Simple mechanisms to handle FITS with IMAGE and TABLE extensions.
- Novel keyword sorting mechanism using index numbers.
- File locking allows multiple programs to run in parallel on the same file.
- Designed to interface with a status server some time in the future.

References and related documents:

- Download latest copies of the C header, **fh.h** and the C source code, **fh.c**
- Printable version of the C-language header file (PostScript)
- MEMO: Proposed Long Term Goals for FITS at CFHT (HTML)
- FITS Standard NOST 100-2.0 (HTML version at STSCI)
- NOAO master keyword dictionary (HTML)

Contents

1 Summary of All Functions

NOTE: In this table, **hu** is a type **HeaderUnit** that was returned from a previous call to fh_create() or fh_ehu(), **fd** is an open file descriptor (type **int**) returned by a previous call to open(), and **idx** is a **double** floating point sorting number used internally by the library. Finally, where-ever **name** and **comment** are found, these are ASCII character strings (type **char***). They must contain only those characters allowed by the FITS standard.

Table 1: Functions to Open, Lock, Close, or create new FITS files

Return	Function	Parameters	Description
HeaderUnit	**fh_create**	*No parameters*	Allocate a table to store FITS cards.
HeaderUnit	**fh_copy_hu**	(HeaderUnit hu)	Allocate a table to store FITS cards and copy an initial set from 'hu'.
fh_result	**fh_destroy**	(HeaderUnit hu)	Free memory associated with 'hu'. Check return!
fh_result	**fh_read**	(HeaderUnit hu, int fd, double idx)	Read header from an open file descriptor.
fh_result	**fh_reindex**	(HeaderUnit hu)	Restore fh_registry.h indices to the cards in 'hu'.
fh_result	**fh_file**	(HeaderUnit hu, const* filespec, fh_mode)	Open and read header from a file or extension.
int	**fh_file_desc**	(HeaderUnit hu)	Get the file descriptor used by fh_file
fh_result(??)	**fh_reserve**	(hu, int n)	Reserve at least *n* cards slots for downstream use.
fh_result	**fh_validate**	(hu)	Test if 'hu' is complete and valid (fh_validate.c)
fh_result	**fh_count_cards**	(hu)	Put library in special mode to count cards
fh_result	**fh_write**	(hu, fd)	Write header, padding, and END to an open file.
fh_result	**fh_write_padding**	(hu, fd)	Write padding to go *after the IMAGE*.
fh_result	**fh_write_padded_image**	(hu, fd, void* data, size, typesize)	Write 'size' image bytes of 'data' plus padding.
fh_result	**fh_read_padded_image**	(hu, fd, void* data, size, typesize)	Read 'size' bytes into user allocated 'data'.
fh_result	**fh_copy_padded_image**	(hu, fd_out, fd_in)	Copy image and padding from another open file.
fh_result	**fh_rewrite**	(hu)	Write the header back to the original file, if possible.

Table 2: Functions for Looking at Extensions and Image data

Return	Function	Parameters	Description
int	**fh_extensions**	(hu)	Reads NEXTEND to determine number of extensions.
int	**fh_image_bytes**	(hu)	Reads BITPIX and NAXIS* to find (unpadded) image size.
int	**fh_image_blocks**	(hu)	Converts fh_image_bytes to size in 2880-byte blocks.
int	**fh_header_blocks**	(hu)	Returns size required for header in 2880-byte blocks.
HeaderUnit	**fh_ehu**	(hu, int number)	Seek to extension by its offset in the file (caution!)
HeaderUnit	**fh_ehu_by_imageid**	(hu, int imageid)	Seek to extension with matching IMAGEID keyword.
HeaderUnit	**fh_ehu_by_extname**	(hu, char* extname)	Seek by EXTNAME (e.g. "im07" or "chip03b")

Table 3: Functions for Looking at Keywords

Return	Function	Parameters	Description
fh_result	**fh_get_bool**	(hu, char* name, fh_bool* value)	*value gets FH_TRUE or FH_FALSE
fh_result	**fh_get_int**	(hu, char* name, int* value)	*value gets integer value of 'name'
fh_result	**fh_get_flt**	(hu, char* name, double* value)	*value gets double-float value of 'name'
fh_result	**fh_get_str**	(hu, char* name, char* value, maxlen)	*value gets string value of 'name'
const char*	**fh_first**	(hu)	Returns the first 80 character record (or NULL)
const char*	**fh_next**	(hu)	Returns the next 80 character record (or NULL)
double	**fh_idx**	(hu)	Return the index for the card last returned by fh_next .
fh_result	**fh_search**	(hu, char* name, double* idx)	Returns FH_SUCCESS if found (and *idx if !NULL)
fh_result	**fh_show**	(hu)	Display a list on stdout (for debugging).
double	**fh_idx_after**	(hu, char* name)	Get 'idx' for a new card after 'name'
double	**fh_idx_before**	(hu, char* name)	Get 'idx' for new card before 'name'

Table 4: Functions for Changing Keywords

Return	Function	Parameters	Description
fh_result	**fh_remove**	(hu, char* name)	Removes a card from the header unit in memory. It will disappear from the file on the next fh_rewrite()
fh_result	**fh_set_all_units**	(hu)	Overrides default behavior of fh_set*() functions so they change extensions too.
void	**fh_set_com**	(hu, idx, char* name, char* comment)	Creates new comment card containing up to 72 ASCII characters. "name" should be "HISTORY" or "COMMENT".
void	**fh_set_bool**	(hu, idx,name, fh_bool value, comment)	Changes or adds the card 'name' with new value of 'T' if value is non-zero, and sets comment field if comment is not empty.
void	**fh_set_int**	(hu, idx, name, int value, comment)	Changes or adds keyword 'name' with new integer value and comment fields.
void	**fh_set_flt**	(hu, idx, name, double value, double flt, double prec, comment)	Changes or adds keyword 'name' with new floating-point value.
void	**fh_set_str**	(hu, idx, name, char* value, comment)	Changes or adds keyword 'name' with new string value and comment fields.
void	**fh_set_val**	(hu, idx, name, char* value, comment)	Changes or adds keyword 'name' with new preformatted value and comment fields. (Use typed functions above instead.)
fh_result	**fh_merge**	(target_hu, source_hu)	Merges all cards from one list into another. Final order depends on the sorting numbers in both lists.

Table 5: Redirecting Error Messages

Return	Function	Parameters	Description
void	**fh_log_warning**	(fh_logger_t log_function)	Set warning log handler
void	**fh_log_error**	(fh_logger_t log_function)	Set error log handler
void	**fh_log_perror**	(fh_logger_t log_function)	Set system error log handler (check errno)

2 Using 'idx' Sorting Numbers

2.1 What are 'idx' numbers?

With other FITS libraries, the final order of the FITS cards in the header is detemined by one of two things:

1. The order in which the cards were added in the source, OR
2. The order in which the cards appear in a template.

For a large system where it may not alway be possible to write all cards from the same section of code, or even the same program, these methods are not the most convenient. A novel feature of this library is that it can sort cards based on numbers, like library of congress reference numbers, which you choose.

The 'idx' argument required by many functions in this library is a floating point number used internally for sorting the cards before they are written to a FITS header. Values below 10.0 are reserved for use by the library itself. Other than that, values can be arbitrarily chosen.

2.2 Reserved Keywords

The following keywords must be placed in a specific order to conform to the FITS standard:

```
SIMPLE, EXTENSION, BITPIX, NAXIS, NAXIS*, EXTEND, NEXTEND, GROUPS,
PCOUNT, GCOUNT, TFIELDS, TFORM*, TBCOL*.
```

The 'idx' values for these are ignored. Use FH_AUTO.

2.3 Auto-assigned 'idx' Numbers

It is also possible to use FH_AUTO for *everything*, in which case the order of the FITS header depends only on the order in which cards were added (except for the specific ones listed in the previous section, which will always be first.)

The original 'idx' numbers are lost once a header has been re-read from a FITS file. In this case they are assigned numbers 10.001, 10.002, 10.003, ... You can change the base starting number (10.0) by passing a different value to fh_read(). Something like that would be useful if the program uses fh_merge() after.

3 Opening, Locking, Closing and Creating FITS files

3.1 fh_create()

Before calling fh_file() to read from a filename (or fh_read() to read from a file descriptor which you opened yourself) this function must be used to create a new HeaderUnit. The HeaderUnit will be empty, until fh_file(), fh_read(), fh_set*(), or fh_merge() functions are used to add cards to it.

3.2 fh_copy_hu()

This creates a new HeaderUnit from an existing one, copying all the FITS cards from the existing one into the otherwise empty new one.

3.3 fh_destroy()

For each HeaderUnit returned by fh_create(), don't forget to pass it to fh_destroy() when it is no longer needed. The following happens in fh_destroy()

1. Any remaining advisory file locks are released.
2. The file is closed, if and only if it was opened with fh_file().
3. Any linked extension header units are fh_destroy()'d.
4. All resources allocated to the list are freed.

The first two steps may have errors, so always check the return code from fh_destroy(). For example:

```
if (fh_destroy(hu) != FH_SUCCESS) rtn = FAIL;
```

WARNING: fh_destroy() should *not* be used on HeaderUnit's returned by the fh_ehu*() functions. These are destroyed automatically when their parent header unit is fh_destroy'ed.

3.4 fh_read()

This is not needed if you use fh_file(). But if you open() your own file descriptor, pass it to this function, along with an empty HeaderUnit from fh_create() to read the header from a FITS file.

Whether you used fh_read() or fh_file(), at this point if the file happens to be an MEF file, you can find out using fh_extensions() and access each extension unit using fh_ehu*().

3.5 fh_file()

While it is possible to open your own file descriptor, and use the library's routines only to parse the header, using fh_file() to open the file for you will give all tools using this library a similar behavior. Several things happen when you call this function:

1. If no filespec, or the special name '-' is given, the library will try to read from Standard Input. If stdin is a terminal device, an error will result.
2. Next, if the filespec exists exactly as given, it is opened, the primary header unit is parsed and returned in HeaderUnit.
3. Next, if the filespec string contains '[]', that portion of the string is removed and treated as an EXTNAME. The header unit of a matching extension is returned in HeaderUnit instead of the primary header. If the file exists, but no matching EXTNAME could be found, the error is the same as if the file was not found.
4. Next, filespec + ".fits" is attempted (with EXTNAME, if any).
5. Finally, if all of the above fail, filespec + "/" + filebase + extnum + ".fits" is tried. In this step, filespec does not include ".fits" if it had it, and extnum is extname with a leading "chip" or "im" removed. This hack allows our current split-file convention to be accessed as if it was MEF.

In addition to a new HeaderUnit, which you must obtain from fh_create() and the 'filespec' parameter, a file mode is required. If you intend to use fh_rewrite() to update any FITS cards, you must choose FH_FILE_RDWR. Otherwise, use FH_FILE_RDONLY for this parameter.

3.5.1 Advisory File Locks

Advisory file locking will automatically be done before reading the header unit. If the file was opened RDWR, the lock is not released until the first call of fh_rewrite(). Otherwise, with RDONLY, the lock is already released by the time fh_file() returns. (*EXCEPTION: If the file contains extensions, such a RDONLY file lock is left in place until fh_destroy() is called.*)

Tests between HP-UX 10.20, Solaris 2.6, and Linux-2.2.16-cfht (includes our NFS3 patches) have been completed. All combinations of NFS server and NFS client (and local file access) were tested including competing access from all three architectures and from six hosts to one file, at the same time. Per run, each test client makes 6 calls to fh_file() and inserts 6 new FITS cards, checks that they appear correctly (PID's are saved in the value field) and checks that other test client's cards are valid as well. The conclusions from the tests are:

- File locking works as advertised on our Linux/HP-UX/Solaris boxes.
- NFS efficiency is not adversely affected by locking.

Read performance was tested with /cfht/bin/fhtool -Vv file.fits, before, during, and after file locking:

1. Before ever applying a lock to a file, the first time a file is accessed over NFS for reading, the speed is limited by either the remote disk, or the network (plus some NFS overheads). This is typically in the range of 3-8 MB/sec on our systems.
2. A second access (even for full CFH12K mosaic images) is always much faster, as it gets read from the cache. Druid (Linux) gets about 300 MB/sec, Ohia (Linux) gets about 250 MB/sec, Mahina (Solaris) gets about 100 MB/sec, and Neptune gets 25 MB/sec (except for large files, which it doesn't seem to be able to cache, so they go back to the 3-8 MB/sec speed.)
3. Access during an exclusive (RDWR) lock will not happen (unless a client ignores the lock.)
4. Access after the lock is back to the speed of (1), except for when the file was not modified (no call to fh_rewrite()). In that case, the next access immediately returns to the cached speed. This means, with our current NFS implementations, use of the cache is apparently put "on hold" during a lock, and *only* invalidated if the file actually changed. At that point, there is no mechanism to determine which blocks of the file have been modified, so the whole thing must be read over the network again. This would actually be the same without locking, by the way.

   Local access to the file *does* figure out which individual blocks have changed, and so in that case the whole file does not have to be re-read into local buffers each time a FITS card is changed.
5. After a file has been locked, changed, unlocked, read once over the network . . . subsequent reads immediately return to the fast cached speed of (2).

These tests have not revealed any problem with file locking on our system, nor have they shown any disadvantage to using file locks. Therefore, the default of the library will be to lock, since no harm is done. It should be safe to run programs which update FITS cards in parallel.

3.6 fh_reindex()

During an fh_read() or fh_file(), libfh assigns arbitrary index numbers to the keywords read (with specific ones only for the first few required cards). fh_reindex() reassigns the numbers from fh_registry.h to the cards in the provided HeaderUnit.

3.7 fh_file_desc()

fh_file() opens a file on a HeaderUnit. Pass this header unit to fh_file_desc() if you need to do any operations directly on the file descriptor. File position after a call to fh_file() is at the start of the image (or first extension header for MEF). To be certain of file position, see fh_ehu() for a way to seek to a specific section of data (including data associated with the primary/only header.)

3.8 fh_reserve()

Use this before calling fh_write() to select the number of reserved COMMENT cards in the header. These cards will be used by *other* programs to add more keywords to the FITS file.

(These *other* programs do not call fh_reserve() themselves. The just call fh_rewrite() and fail if there wasn't enough room.)

3.9 fh_write()

Write the FITS header to a file descriptor. After this, make a call to fh_write_padded_image() to add the data onto the new FITS file. Use fh_rewrite() instead of fh_write() for programs which only update FITS cards without generating a new FITS file.

3.10 fh_write_padding()

This function writes 0-byte padding intended to go after the image data. If the header unit contains XTENSION='TABLE' then ASCII space (0x20-byte) padding is used instead, to satisfy the requirements of an ASCII table.

You don't need this function if you use fh_write_padded_image() or fh_copy_padded_image(). If you choose to write the data to the file descriptor yourself, however, you might find it useful to let the library calculate the padding for you.

3.11 fh_write_padded_image()

This routine can be used to both image data and preformatted ASCII tables. For image data, the data can be either in host byte-order or FITS byte-order. 'data' is a buffer of 'size' bytes which you must allocate first. The library calculates its own value for 'typesize' and 'size' based on BITPIX and NAXIS (see fh_image_bytes()), which must match your value, or an error will be returned. This, 'size' is completely redundant, but required as a sanity check, while 'typesize' is critical in controlling byte-swapping.

File pointer is left at the end of the padding after the image data. When building MEF files, this is the right place to begin the next header unit. If not, just close the file here... and check for errors when calling close()!

The 'typesize' parameter affects the byte-order in which 'data' is written to the FITS file. For the most common case, where 'data' is a buffer of host-byte-order 8, 16, 32, or 64 bit values, 'typesize' should be set to sizeof(char), sizeof(short), or sizeof(long), or sizeof(double). The library will automatically determine if the data needs to be swapped when written to FITS-byte-order. This is the recommended way to let byte-swapping happen. If your program has already taken care of putting 'data' in FITS-byte-order, then pass FH_TYPESIZE_RAW (a value of 0) for 'typesize' to force no byte-swapping. Finally, to force byte-swapping to happen, even on a Sparc or other type of computer where host- and FITS-byte-order are the same, pass the negative of the sizeof(<datatype>). Note that 'typesize' is a number of *bytes*, i.e. is equal to —BITPIX—/8.

Note that the -sizeof() options will swap *even* on architectures such as Sparc and HP-PA where the data is normally already stored in the correct byte order for FITS files.

Also note that typesize must be either 0, or equal to BITPIX/8 or library will fail and return FH_BAD_VALUE.

Table 6: Choices for 'typesize' Parameter

BITPIX	typesize=	swap by:	data written to file is:
8	sizeof(unsigned char)		unchanged
16	sizeof(signed short)	2	swapped if needed
32	sizeof(signed long)	4	swapped if needed
-32	sizeof(float)	4	swapped if needed
-64	sizeof(double)	8	swapped if needed
(any)	FH_TYPESIZE_RAW		never swapped
16	-sizeof(signed short)	2	always swapped
32	-sizeof(signed long)	4	always swapped
-32	-sizeof(float)	4	always swapped
-64	-sizeof(double)	8	always swapped

3.12 fh_read_padded_image()

See description for fh_write_padded_image(). Instead of writing data in your buffer, this reads the image from the file and places it in the buffer (which you must still allocate.) After reading image data, padding in the input file is verified, but not copied to your buffer. (I.e. 'size' must still be exactly equal to fh_image_bytes() or the function will fail.)

Byte-swapping happens according to the table in fh_write_padded_image. Pass the correct size of the data (in bytes) or FH_TYPESIZE_RAW (a value of 0) if you want to read raw bytes from the FITS file in order.

3.13 fh_copy_padded_image()

This performs a straight copy from one FITS file to another. Both are expected to be at the start of their image data. The number of bytes to read and the padding are determined from HeaderUnit's BITPIX and NAXIS* values (see fh_image_bytes()).

This routine does not work on the primary header unit (it lacks image data.) To copy multiple image extensions, each extension must be copied by doing the following:

- read the header from the original file (after fh_read() or fh_ehu(), this input file is now at the start of image data for the corresponding extension.)
- write (with fh_write()) the header to the new file, which should be at the end of the padding from the previous image extension.
- then use fh_copy_padded_image(), passing it the current extension header used in the two steps above, and the two file descriptors used above.

See the source code to fhtool.c, which splits and joins MEF, for some examples using this call. (But be warned, there's a lot of other cruft in fhtool.c.)

3.14 fh_rewrite()

Using the same 'fd' obtained by fh_file(), or passed to fh_read(), this call seeks back to the start of the header and attempts to rewrite it to reflect any changes made by fh_remove() or fh_set() calls on the HeaderUnit.

fh_rewrite() does *not* take a file descriptor argument. It will fail if there was no previously successful fh_file() or fh_read() call from which it can obtain a file descriptor. It will also return delayed failure from any fh_set() calls which may have failure due to incorrect usage or out of memory errors.

4 Examining Extensions and Image Data

4.1 fh_extensions()

If no EXTEND keyword is found, or if it is not set to T(rue) then 0 is returned. Otherwise the value of the NEXTEND keyword is returned. Use this to test for a mutli-extension FITS file.

4.2 fh_image_bytes()

This returns the expected unpadded image size in bytes. It should be called only after valid BITPIX and NAXIS* values have been set or read from a file. Here is the formula used to obtain the image size in bytes:

$NAXIS1 \times NAXIS2 \times ... \times BytesPerPixel$

Where $BytesPerPixel$ is 1 for BITPIX=8, 2 for BITPIX=16, 4 for BITPIX=+/-32, and 8 for BITPIX=-64.

4.3 fh_image_blocks()

This returns the following:

$(fh\_image\_bytes() + 2880 - 1)/2880$

4.4 fh_header_blocks()

This returns the number of 2880-byte blocks fh_write() or fh_rewrite() will need for the *header*.

4.5 fh_ehu()

Use this only if you are making a program which loops through all extensions (see the example fhextname.c). In all other cases, refer to extensions by IMAGEID or EXTNAME using one of the following two functions.

Do not fh_destroy() the HeaderUnit returned by the fh_ehu() functions. It is destroyed automatically when the corresponding primary header is destroyed.

4.6 fh_ehu_by_imageid()

Returns a HeaderUnit which contains a matching IMAGEID keyword and seeks the file to the start of the data for that extension.

If no matching IMAGEID is found, 0 is returned.

Do not fh_destroy() the HeaderUnit returned by the fh_ehu() functions. It is destroyed automatically when the corresponding primary header is destroyed.

4.7 fh_ehu_by_extname()

Returns a HeaderUnit which contains a matching EXTNAME keyword and seeks the file to the start of the data for that extension.

If no matching EXTNAME is found, 0 is returned.

Do not fh_destroy() the HeaderUnit returned by the fh_ehu() functions. It is destroyed automatically when the corresponding primary header is destroyed.

5 Looking at Keywords

5.1 fh_get_bool()

Returns FH_SUCCESS if 'name' is found *and* contains either T or F (and not as a string, but as a 'logical' FITS card.)

- If the card contains T, *value will contain FH_TRUE.
- If the card contains F, *value will contain FH_FALSE.

5.2 fh_get_int()

Returns FH_SUCCESS if 'name' is found *and* contains a valid integer. A floating point value (with a decimal point) is not a valid integer. *value will contain the value as converted by the C strtol() function.

5.3 fh_get_flt()

Returns FH_SUCCESS if 'name' is found *and* contains a valid float (real) value. *value will contain the result of the C strtod() function.

5.4 fh_get_str()

Returns FH_SUCCESS if 'name' is found *and* contains a FITS string in single quotes ('...'). *value will contain a '\0'-terminated list of (up to) the first maxlen characters. (Always pass buffers of (FH_MAX_STRLEN + 1) characters to be sure returned strings will not have to be truncated by the library.)

5.5 fh_first() and fh_next()

For access to each of the raw, 80 (FH_CARD_SIZE) records in the header, these functions can be used in a loop (see the "second way" in example fhlist.c).

5.6 fh_idx()

This returns the index of the card last returned by fh_next().

5.7 fh_search()

Use this to see if a keyword exists. A return of FH_SUCCESS means the keyword exists. If you pass a non-NULL pointer in 'idx', the current 'idx' will be returned in *idx.

5.8 fh_show()

Prints the first 79 columns of each card to stdout. Column 80 is replaced by a newline, so the terminal doesn't have to have exactly 80 columns to display properly, and the search-forward ('/' key) of 'less' can be used. See the example fhlist.c.

5.9 fh_idx_before() and fh_idx_after()

Use these functions to obtain 'idx' numbers which would cause a new card to appear just before or after an existing card. Note that if the "new" card already exists too, it will be replaced and will *not* be relocated in the header.

Pass the return of this function as the 'idx' argument to the fh_set*() routines. Alternatively, you can pass FH_AUTO or a fixed value (see the section on 'idx' numbers.)

6 Changing Keywords

fh_set_int(), fh_set_flt(), fh_set_str(), fh_set_bool(), and fh_set_val() all follow the same logic to change or add a keyword to a FITS header:

1. If the keyword already exists:
   - The *original* position in the header and 'idx' assignment are kept. (So the 'idx' argument to fh_set() is ignored in this case.)
   - The new value is substituted.
   - A new comment is inserted, unless comment is NULL, in which case the old comment is left in tact.
2. If the keyword is not found:
   - A new card is inserted according to the 'idx' number given (at the end of the list if idx == FH_AUTO.)
   - The new value is used.
   - The new comment is used, or if it is NULL, no comment field is inserted (not even the '/' character.)

For fh_set_com(), each call results in (case 2), a new card being added, *unless* an matching keyword with exactly the same 'idx' number (and not FH_AUTO) is found. So, specific COMMENT idx=1001.1 could be set and re-set any number of times with fh_set_com(), but if the 'idx' numbers were different, or equal to FH_AUTO then multiple fh_set_com() calls would result in multiple COMMENT cards being added to the header.

All of this is done in memory. Nothing happens to the file until fh_rewrite() is called. At that time, if new keywords were added, the call may fail because of lack of space in the header.

6.1 fh_remove()

Remove a card from the list. Upon the following fh_rewrite(), the card will disappear from the file, and an extra reserved COMMENT card will appear at the end to replace it, keeping the header the same size, no matter how many cards you remove.

6.2 fh_set_all_units()

By default, if fh_set() operations are performed on a PHU read from a file, they are not applied to extension units. Call this function *before* calling any other fh_set* functions on a PHU when you also want the extensions (which you'd get with fh_ehu()) to be changed, otherwise fh_set() only applies to the current HeaderUnit.

6.3 fh_set_com()

'name' should be either "COMMENT" or "HISTORY" and 'value' is the rest of the line (will be truncated at 72 characters.)

6.4 fh_set_bool()

Set a logical or boolean FITS keyword. A non-zero 'value' or FH_TRUE is saved as 'T' and FH_FALSE is saved as 'F'.

6.5 fh_set_int()

Set an integer FITS keyword.

6.6 fh_set_flt()

Use this to set float (real) values. The value must be of type 'double' and will be inserted in the FITS card as a number with a decimal point, and possibly an exponent (as determined by the %G format instruction to printf). 'digits' is the number of significant digits in the quantity 'value'. It must be an integer value 1 or greater. Alternatively, 'digits' can be specified as .1, .2, .3, ...9 to specify up to 9 fixed digits after the decimal point (and used instead of plus a trailing decimal to keep it a legal FITS floating point keyword. When printing a year, with a century, prec should be 0, or *at least* 4 to ensure that the value does not get printed with an exponent by %G.

6.7 fh_set_str()

Set a string FITS keyword. If string is longer than 18 characters, the start of the comment field will be shifted, and a comment may get truncated.

6.8 fh_set_val()

Similar to fh_set_str(), but assumes that you have already sprintf'd the value into a buffer. Use this to get precise control over the formatting of the value field.

6.9 fh_merge()

Merge source_hu into target_hu and sort according to the 'idx' numbers in both lists. source_hu is not modified.

7 Redirecting Error Messages

By default, libfh will print any error and warning messages directly to "stderr." You do not have to use any of these functions if that behavior is acceptable.

It is possible to define your own log handlers which the libfh will call with a single string constant argument whenever it has an error or warning to report. It is also possible to set all three to NULL using the functions below to suppress all log output.

7.1 fh_log_warning()

Pass NULL to suppress warnings, or a pointer to your own function which should be called whenever the library has warning to report. Warnings are used to report illegal characters, values which had to be truncated, or invalid characters in FITS padding regions.

7.2 fh_log_error()

Pass NULL to suppress library errors, or a pointer to your own function which should be called whenever the library has an error (typically, and error code is returned in fh_result at the same time.) Error strings correspond roughly to the descriptions of the fh_result error codes, but may also contain additional information.

7.3 fh_log_perror()

The default handler for this is the ANSI C call "perror()". If you install your own call, it should print the message passed, but also check the value of errno. This is used when a system call (such as open, close, read, write, fcntl) fails.

8 Including and Linking the Library

There are two components to the library: a header file "fh.h" and the implementation, "fh.c".

Programs which libfh as part of Pegasus can include the header file like this:

```
#include "fh/fh.h"
```

Linking with the library is accomplished by adding -lfh to the $(EXECNAME) line of the Makefile:

```
$(EXECNAME): $(OBJS) -lfh
```

The above applies to projects in the CFHT source tree, /cfht/src/...

If you made your own copies if fh.h and fh.c, include just "fh.h" instead of "fh/fh.h" and link fh.c into your program as just another C file.

9 Recipes for Converting to libfh

9.1 Opening a File

Find the place(s) in the code where a FITS file is opened and closed. For a Pegasus handler based on libff, the old call to open the file would look something like this:

```
ff_init(filename);
```

and replace it with:

```
HeaderUnit hu = fh_create();

/* Use FH_FILE_RDONLY below, if you only want to read the file. */
if (fh_file(hu, filename, FH_FILE_RDWR) != FH_SUCCESS)
   /* ERROR ... */
```

- If the code fopen()'d a file itself, it must be upgraded to use fh_file because FILE streams are buffered and will not work reliably in conjunction with file locking. *You MUST use file descriptor I/O instead of streams with this library!*

- If the code open()'d the file itself and read the headers, both of these steps are replaced by the fh_file() call. If you need to obtain the file descriptor yourself, from your own call to open(), then use fh_create() to obtain an empty header unit, followed by fh_read() to read from your own file descriptor *instead* of the single call to fh_file(). In either case, fh_file() / fh_read() has taken care of reading all the FITS cards into a table, so remove any old code that did that.
- Your code should only close() the file if you opted to also open() it yourself. In this case, be sure to *check the return code of close()!* If there is not enough room to write a file, some filesystems return a delayed error, on close() instead of on the write().

  If fh_file() was used instead of open(), fh_destroy() will close() the file, but for the same reasons, *check the return of fh_destroy()* if you opened a file for writing.

The FITS header, and all extension headers are now available by passing "hu" to the other functions of the library. Note: Since fh_file() returns a handle, it is possible to have multiple FITS files open at once.

9.2 Changing Keywords

Calls to the libff routine to set a string:

```
ff_chng_value(FF_CARDNAME, FITS_STRING, string);
```

are changed to:

```
fh_set_str(hu, FH_AUTO, "CARDNAME", string, comment);
```

The actual "CARDNAME" and 'comment' field can be found by looking inside the "handler.def" template file. After a handler is converted, *and* if DetCom is generating the FITS file (i.e., leaving room with fh_reserve) then the .def template is no longer needed.

An upgraded handler still works fine with templates too, though, because it finds the template slots in the FITS file and uses those before looking for COMMENT reserve space.

ff_chng_value() calls with FITS_REAL, FITS_INTEGER, and FITS_LOGICAL are easiest to convert to directly to calls to fh_set_val(), which allows the calling program to do the sprintf'ing and precisely control the formatting of the value:

```
double juliandate = ...;
char String[FH_MAX_STRLEN+1];

sprintf(String, "%15.6f", juliandate);
ff_chng_value(FF_MJDOBS, FITS_REAL, String);
```

becomes:

```
double juliandate = ...;
char String[FH_MAX_STRLEN+1];

sprintf(String, "%15.6f", juliandate);
fh_set_val(hu, FH_AUTO, "MJD-OBS", String, comment);
```

(Note how the actual name of the FITS keyword is not always the name of the defined symbol less the FF_. You must check the template and ff/ff.h for the real name.)

See sections on fh_set_int(), fh_set_flt(), and fh_set_bool() for a way to set integer, float, and T/F values using the library's internal formatting. This is recommended over fh_set_val(). The above call could be changed to:

```
double juliandate = ...;
fh_set_flt(hu, FH_AUTO, "MJD-OBS", juliandate, .6, comment);
```

9.3 "Flushing" Changes to a File

After *all* calls to fh_set_*() have been done, make one call to write the changes back to the file. *Check for errors here!*

```
    if (fh_rewrite(hu) != FH_SUCCESS) rtn = FAIL;
    if (fh_destroy(hu) != FH_SUCCESS) rtn = FAIL;
    return rtn;
```

WARNING: Do not combine the two statements above with '||' (short-circuit logic), or simply "return FAIL" on the first failure because the memory and file locks associated with "hu" may not be properly released!

In Pegasus handlers with libff, this replaces calls to ff_flush() and ff_free().

9.4 Closing a File

Whether fh_file() or fh_create() was used, when the HeaderUnit is no longer needed, make a call to:

```
    if (fh_destroy(hu) != FH_SUCCESS) return FAIL;
```

In Pegasus handlers, this replaces the call to ff_free().

10 Examples

10.1 Display any Primary or Extension Header (fhlist.c)

This program will show the FITS header from any file, or any extension within the file (by using a feature of fh_file() which searches for an extension when the filename contains "fitsfile[extname]".)

This small program is also a good way to test the behavior of fh_file().

```
   #include "fh/fh.h"

   int
   main(int argc, const char* argv[])
   {
      HeaderUnit hu = fh_create();

      if (fh_file(hu, argv[1], FH_FILE_RDONLY) != FH_SUCCESS)
 exit(1);
      fh_show(hu);
      fh_destroy(hu); /* No errors possible, since RDONLY */
      exit(0);
   }
```

```
   #include "fh/fh.h"

   int
   main(int argc, const char* argv[])
   {
      HeaderUnit hu = fh_create();
      const char* card;
```

```
     if (fh_file(hu, argv[1], FH_FILE_RDONLY) != FH_SUCCESS)
exit(1);
     for (card = fh_first(hu); card; card = fh_next(hu))
     {
        printf("%.*s\n", strlen(card)==80?80:79, card);
     }

     fh_destroy(hu); /* No errors possible, since RDONLY */
     exit(0);
  }
```

10.2 Loop through all extensions by EXTNAME (fhextname.c)

When there is a separate FITS file for each amplifier, finding the names of all the "extensions" is as easy as listing all the files in a subdirectory. Here's a utility which generates an equivalent list of names for MEF Files. The output from this program can be used as the arguments to a pipeline program which uses fh_file() to open one file or extension each time it is run. Multiple FITS files may be given on the command line. In this example, the 'fhextname' program is used just as 'ls' would have been with basic FITS files:

```
#!/bin/sh
for i in `fhextname *.fits`
do
   reduce "$i"
done
```

Note that this also works for split files, since 'fhextname' just prints the name of the fits file itself, if there are no extensions. Here is the source code for 'fhextname':

```
#include <stdio.h>
#include <stdlib.h>
#include "fh/fh.h"

int
main(int argc, const char* argv[])
{
   HeaderUnit hu, ehu;
   char extname[FH_MAX_STRLEN+1];
   int i, ext;

   for (i = 1; i < argc; i++)
   {
      hu = fh_create();
      if (fh_file(hu, argv[i], FH_FILE_RDONLY) != FH_SUCCESS)
         exit(EXIT_FAILURE);

      if (fh_extensions(hu) < 1)
      {
         printf("%s\n", argv[i]); /* No extensions?  Print the filename. */
      }
      else for (ext = 1; ext <= fh_extensions(hu); ext++)
      {
         if (!(ehu = fh_ehu(hu, ext)) ||
         fh_get_str(ehu, "EXTNAME", extname, sizeof(extname)) != FH_SUCCESS)
         {
            fprintf(stderr, "error: Cannot read EXTNAME from `%s' for extension #%d\n", argv[i], ext);
```

```
            fh_destroy(hu);
            exit(EXIT_FAILURE);
         }
         printf("%s[%s]\n", argv[i], extname);
      }
      fh_destroy(hu); /* No errors to check, since RDONLY */
   }
   exit(EXIT_SUCCESS);
}
```

10.3 Tool to Set any Keyword(s) (fhset.c)

Here is the source code for 'fhset', a command line utility which sets arbitrary keywords and comment fields in a FITS file (and extension units, if it has any.) The following would cause this tool to update/add GAIN (a float value), OBSID (an integer), CCD (a string) and INHERIT (boolean).

```
#!/bin/sh
fhset 12345o.fits "GAIN=1.35" "OBSID=12345" "CCD='Cam corder CCD'" "INHERIT=F"
```

Here is the source for the fhset.c program:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "fh/fh.h"

int
main(int argc, const char* argv[])
{
   HeaderUnit hu = fh_create();
   int i, exitcode = EXIT_SUCCESS;

   if (argc < 2)
   {
      fprintf(stderr, "usage: fhset <fitsfile> KEYWORD=value [/comment] [KEYWORD=value [/comment] ...]\n
      exit(2);
   }

   if (fh_file(hu, argv[1], FH_FILE_RDWR) != FH_SUCCESS)
      exit(EXIT_FAILURE);

   for (i = 2; i < argc; i++)
   {
      char* arg = strdup(argv[i]);
      char* val = strchr(arg, '=');
      const char* com;

      com = 0;
      if (!val)
      {
         fprintf(stderr, "fhset: error: Specify KEYWORD=value (with no spaces)\n");
         fh_destroy(hu);
         exit(2);
      }
      *val++ = '\0';
      if (i + 1 < argc && *argv[i + 1]=='/')
```

```
      { com = argv[++i]; com++; } /* Don't include the '/' itself. */
      fh_set_val(hu, FH_AUTO, arg, val, com);
   }
   /*
    * Write the changes to the file, and unlock it.
    */
   if (fh_rewrite(hu) != FH_SUCCESS) exitcode = EXIT_FAILURE;
   if (fh_destroy(hu) != FH_SUCCESS) exitcode = EXIT_FAILURE;
   exit(exitcode);
}
```

10.4 Tool for making Reserve Space (fhreserve.c)

This one still has to be written . . .

Something which locks the entire file and shifts everything without removing and creating a completely new file would probably work the best over NFS.

For now, we must make sure the DetCom or anything else which creates FITS files is leaving enough reserved cards for the downstream programs.

10.5 Update a Card with known Keyword

In this example, the keyword WEATHER may or may not exist.

- If it exists, the keyword's value and comment fields are changed.
- If it doesn't exist, it is added at the end of the FITS header.
- If there is no room, the program fails.

```
#include <stdio.h>
#include <stdlib.h>
#include "fh/fh.h"

int
main(int argc, const char* argv[])
{
   HeaderUnit hu = fh_create();
   int exitcode = EXIT_SUCCESS;

   /*
    * This program does not like to read from pipes.  Check explicitly.
    */
   if (!argv[1] || !strcmp(argv[1], "-"))
   {
      fprintf(stderr, "error: Updating FITS cards requires an input file.\n")
      exit(EXIT_FAILURE);
   }
   if (fh_file(hu, argv[1], FH_FILE_RDWR) != FH_SUCCESS)
      exit(EXIT_FAILURE);
   fh_set_str(hu, FH_AUTO, "WEATHER", "Excellent", "This card is bogus");
   if (fh_rewrite(hu) != FH_SUCCESS) exitcode = EXIT_FAILURE;
   /* File is now changed (and automatically unlocked) */
   if (fh_destroy(hu) != FH_SUCCESS) exitcode = EXIT_FAILURE;
   exit(exitcode);
}
```

10.6 Add set of COMMENT cards to already open file

This example is not a complete program, but a function within a larger program which has its own way of opening a file descriptor (and thus, does not want to use fh_file.)

```
PASSFAIL
add_some_comments(int fd)
{
   HeaderUnit hu = fh_create();
   PASSFAIL rtn = PASS;

   if (fh_read(hu, fd, FH_AUTO) != FH_SUCCESS) rtn = FAIL;
   else
   {
      fh_set_com(hu, FH_AUTO, "COMMENT", "Here's the 1st of 3 lines");
      fh_set_com(hu, FH_AUTO, "COMMENT", "which will be added");
      fh_set_com(hu, FH_AUTO, "COMMENT", "to the end of the header");
      if (fh_rewrite(hu) != FH_SUCCESS) rtn = FAIL;
   }

   /* Don't forget to free memory, even on failure, */
   /* especially since this unlocks the file too. */
   if (fh_destroy(hu) != FH_SUCCESS) rtn = FAIL;
   return rtn;
}
```

10.7 Build a FITS File from Scratch

Here a complete FITS file is built from scratch, using routines in libfh. It demonstrates several other things which the examples above do not:

- Writing a FITS file to a pipe (STDOUT). (The program would also work if 'fd' was a network socket or a real file.)
- Use of fh_reserve() to make room for other programs which will add cards.
- Use of 'idx' numbers (although it would work just as well using FH_AUTO for all the calls in this example.)

Source code follows:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "fh/fh.h"

int
main(int argc, const char* argv[])
{
   HeaderUnit hu = fh_create();
   time_t date = time(0);
   short* data;
   int exitcode = EXIT_SUCCESS;
   int fd = STDOUT_FILENO; /* By default, write to stdout */
   double etime = 10.0; /* Normally these values might */
   int w = 2048, h = 4096; /*  not be hardcoded like this */

   data = malloc(sizeof(short)*w*h);

   fh_reserve(hu, 50); /* Reserve space for 50 cards (TCS, Elixir?, etc.) */
   fh_set_bool(hu, FH_AUTO, "SIMPLE", 1, "Standard FITS");
   fh_set_int(hu,  FH_AUTO, "BITPIX", 16,"16-bit data");
   fh_set_int(hu,  FH_AUTO, "NAXIS",  2, "Number of axes");
   fh_set_int(hu,  FH_AUTO, "NAXIS1", w, "Number of pixel columns");
   fh_set_int(hu,  FH_AUTO, "NAXIS2", h, "Number of pixel rows");
   fh_set_int(hu,  FH_AUTO, "PCOUNT", 0, "No 'random' parameters");
   fh_set_int(hu,  FH_AUTO, "GCOUNT", 1, "Only one group");

   strftime(str, sizeof(str)-1, "%Y-%m-%dT%T", gmtime(&date));
   fh_set_str(hu, 104, "DATE", str, "UTC Date of file creation");
   strftime(str, sizeof(str)-1, "%a %b %d %H:%M:%S %Z %Y", localtime(&date));
   fh_set_str(hu, 104.1,"HSTTIME", str, "Local time in Hawaii");
   fh_set_str(hu, 105,  "ORIGIN", "CFHT", "Canada-France-Hawaii Telescope");
   fh_set_flt(hu, 141., "BZERO",0.0, 6,"Zero factor");
   fh_set_flt(hu, 142., "BSCALE",1.0, 2, "Scale factor");
   fh_set_flt(hu, 150, "DATAMIN", datamin, 6, "Minimum value of the data");
   fh_set_flt(hu, 151, "DATAMAX", datamax, 6, "Maximum value of the data");
   fh_set_flt(hu, 160, "SATURATE", 4016.0, 6, "Saturation value");
   fh_set_flt(hu, 220, "EXPTIME", etime, 5, "Integration time (seconds)");
   sprintf(str, "%d %d", binmode + 1, binmode + 1);
   fh_set_str(hu, 230, "CCDSUM", str, "Binning factors");
   fh_set_com(hu, 1400.0, "COMMENT", "");
   fh_set_com(hu, 1400.1, "COMMENT", " Camera status record:");
   fh_set_com(hu, 1400.2, "COMMENT", "");
   if (camera_status==0)
     fh_set_str(hu, 1410, "DETSTAT", "ok", "(camera_status is 0)");
   else
     fh_set_int(hu, 1411, "DETSTAT", sbstat.imaging_ccd_status, "error!");
   fh_set_str(hu, 1601,"DETECTOR", info.camera_name, info.serial_number);
   if (fh_write(hu, fd) != FH_SUCCESS ||
       fh_write_padded_image(hu, fd, data, w*h*sizeof(unsigned short)) != FH_SUCCESS)
      exitcode = EXIT_FAILURE;
   if (fh_destroy(hu) != FH_SUCCESS) exitcode = EXIT_FAILURE;
   exit(exitcode);
}
```

10.8 Build an MEF File from Scratch

The fh_ehu*() functions are only used to access extensions within an already existing MEF. The are not used when building an MEF File for the first time. This task must be done by writing the appropriate interleaved header units and image data manually.

The following example constructs a very minimal MEF File with only two extensions. For a more comprehensive example, see the source code of DetCom, /cfht/src/medusa/detcom/detcom/det_data.c.

```
#include <stdio.h>
#include <stdlib.h>
#include "fh/fh.h"

#define NEXTEND 2

int
main(int argc, const char* argv[])
{
   short dummy_data[3 * 2] = { 1, 2, 3, 4, 5, 6 };
   HeaderUnit phu, ehu;
   int i, fd = STDOUT_FILENO;

   phu = fh_create();
   fh_set_bool(phu, FH_AUTO, "SIMPLE",  1, "Standard FITS");
   fh_set_int( phu, FH_AUTO, "BITPIX", 16, "Bits per pixel (not appl. to PHU)");
   fh_set_int( phu, FH_AUTO, "NAXIS",   0, "No image data with primary header");
   fh_set_bool(phu, FH_AUTO, "EXTEND",  1, "File contains extensions");
   fh_set_int( phu, FH_AUTO, "NEXTEND", NEXTEND, "Number of extensions");
   fh_write(phu, fd);
   fh_destroy(phu);

   for (i = 0; i < NEXTEND; i++)
   {
      char str[FH_MAX_STRLEN+1];

      sprintf(str, "im%02d", i);
      ehu = fh_create();
      fh_set_str(ehu, FH_AUTO, "XTENSION", "IMAGE", "Image extension");
      fh_set_int(ehu, FH_AUTO, "BITPIX", 16, "Bits per pixel");
      fh_set_int(ehu, FH_AUTO, "NAXIS",   2, "Number of axes");
      fh_set_int(ehu, FH_AUTO, "NAXIS1",  3, "Number of pixel columns");
      fh_set_int(ehu, FH_AUTO, "NAXIS2",  2, "Number of pixel rows");
      fh_set_int(ehu, FH_AUTO, "PCOUNT",  0, "No 'random' parameters");
      fh_set_int(ehu, FH_AUTO, "GCOUNT",  1, "Only one group");
      fh_set_str(ehu, FH_AUTO, "EXTNAME", str, "Extension name");
      fh_write(ehu, fd);
      fh_write_padded_image(ehu, fd, dummy_data, sizeof(dummy_data));
      fh_destroy(ehu);
   }
}
```