HEASOFT Developer's Guide

Version 1.3

HEASARC
Code 662
Goddard Space Flight Center
Greenbelt, MD 20771
USA

June 2018

Contents

1 Introduction **3**

1.1 Configuration Management and HEASOFT Setup. 3

1.2 Input and Output 4

1.2.1 Input 4

1.2.2 Output 5

1.3 General Notes 6

1.4 Task Name and Version 6

1.5 CALDB Access 7

1.6 Scripting 7

2 Overview of HEASOFT Libraries **9**

2.1 heainit 10

2.2 heaio 11

2.3 heautils 12

2.4 heasp 15

2.4.1 PHA files 15

2.4.2 RMF files 16

2.4.3 ARF 17

2.4.4 Utility routines 18

3 HEADAS Makefiles **19**

3.1 Introduction 19

3.2 Simplest case: A Makefile for a single task 19

3.3 Standard Macros 21

3.3.1 Macros Pertaining To All Build Actions 21

3.3.2 Macros Pertaining To Tasks 22
3.3.3 Macros Pertaining To Libraries 23
3.3.4 Macros Pertaining To Installation 23
3.3.5 Macros Pertaining To Subdirectories 25

4 HEADAS Error Handling Facility 27

4.1 Introduction 27
4.2 HEADAS Error Handling API 28

Chapter 1

Introduction

HEASOFT is the HEASARC's software suite that currently encompasses both new and legacy mission-independent or multi-mission FTOOLS, the XANADU suite (XSPEC, XRONOS, and XIMAGE), XSTAR, and numerous mission-specific packages.

HEASOFT uses the HEADAS build environment (described in more detail below), and is composed of two basic parts: a set of mission- independent (HEATOOLS, HEASPTOOLS, et al.) and multi-mission (HEAGEN, ATTITUDE, et al.) FITS utilities which are developed and maintained within the HEASARC, as well as a set of mission-specific packages developed and maintained (in whole or in part) by outside groups. The core modern HEASARC utilities are generally written in ANSI C for maximum portability. Missions are, however, free to use any language they wish within their own package, bearing in mind that certain choices may limit the platforms and/or compilers on which their tasks will build and run.

Many of the older, legacy FTOOLS now have newer, improved analogs in HEASOFT, e.g. in the HEATOOLS or HEASPTOOLS packages, and all users and developers are encouraged to migrate their applications to use these new tasks instead. For the forseeable future, however, many tasks (predominantly Perl scripts) in mission-specific packages will continue to call tasks which still reside in the legacy FTOOLS portion of HEASOFT, and software distributions by the HEASARC will generally include all necessary components in a single tar file to make things as easy for users as possible. However, in keeping with a modular approach it is important that mission-specific tasks not link against the older FTOOLS libraries (e.g., xanlib). If mission developers find that they require those subroutines in their tasks then they should be migrated to an appropriate library, either in the HEASOFT core or a mission-specific library, and developers should coordinate with the HEASARC programming staff in this process.

1.1 Configuration Management and HEASOFT Setup

All configuration management of HEASOFT will be handled by the HEASARC, including such things as Makefiles and configure scripts. Code revision control is handled via a central Git repository resident on HEASARC servers. Currently only developers with NASA/GSFC credentials may access this server.

The directory structure for any given subpackage will be reasonably flexible so that mission software developers can arrange things to suit their own needs and/or preferences. Certain constraints may, however, be required to accomodate the overall HEADAS build paradigm.

Full, top-down builds are started from the heasoft/BUILD_DIR directory using the typical GNU-based steps of './configure' followed by 'make' and then finally 'make install'. Internally, however, builds will be managed by the HEASARC's 'hmake' utility. 'hmake' is designed to also make it easy to build all or part of the distribution on any supported platform without having to make modifications by hand specifying the location and/or names of necessary flags and libraries. Before using hmake, one simply needs to initialize HEASOFT by sourcing the file $HEADAS/BUILD_DIR/headas-init.csh (or .sh), where $HEADAS is the environment variable which points to the location of the relevant HEASOFT installation, e.g. /usr/local/heasoft/x86_64-pc-linux-gnu-libc2.12.

1.2 Input and Output

1.2.1 Input

All input to HEASOFT tasks is controlled by a parameter interface library (APE) which is developed and maintained by the HEASARC, but was originally based on the INTEGRAL Science Data Center's "PIL" code. APE has a very similar look and feel to XPI (the original parameter interface used in FTOOLS) but includes such additional features as enumerated values, minimum-maximum range checking, the ability to use environment variables in parameters and a dedicated "filename" type. APE is callable by C, C++, Fortran and Perl tasks.

There are three commonly-used parameters which are handled intrinsically by the internal HEASOFT initialization routines and thus developers do not need to explicitly read them at the individual task level. (A fourth parameter, "mode", is an APE internal and operates exactly as in XPI). The standard HEASOFT parameters are:

- CHATTER The (integer) "chatter" parameter may be used to control the verbosity of a HEASOFT task. (This is similar to the "verbose" parameter used e.g. in CIAO, however, since a number of FTOOLS tasks use "chatter" we have chosen to keep the same name for consistency across the HEASOFT suite.) Developers are free to specify any range (via the parameter min/max) but we recommend the following (0-5):

  0 suppresses all but absolutely essential output

  1-4 normal levels. The different levels can be used on a task by task basis to control the amount of output information. The distinction between the different chatter levels (if any) must be documented in the task's help file. For many tasks, all 4 chatter levels might produce exactly the same output.

  5 debug mode: prints detailed messages about each step in the program

  The desired chatter value will be specified by the user at runtime and read automatically during the initialization phase. The task developer may then funnel diagnostic output through the supplied routines (see output section below) which take as their first argument a threshold chatter level below which the output will be suppressed. A chatter parameter is not required for any task, however, calling headas_chat()/hdchat() in a task having no chatter parameter will result in an error.

- CLOBBER

  If a (boolean) "clobber" parameter is present for a given task it will be read during the initialization phase. Developers may then call headas_clobberfile(filename) which will delete the specified file if it exists and if the "clobber" parameter was set to "yes". Note that an alternative to clobber exists for FITS files since CFITSIO will clobber any file which begins with the "!" character.

- HISTORY

  The (boolean) "history" parameter controls whether or not the user wishes to allow a set of HISTORY keywords listing the runtime values of all task parameters to be written into any FITS file header. The developer simply calls HDpar_stamp() specifying the desired FITS file and extension and, if the history parameter value at runtime permits it, the HISTORY block will be written. If the task has no history parameter then a call to HDpar_stamp() will return an error. Each HISTORY keyword block will be clearly delimited and will include the task name/version and a timestamp. Use of HDpar_stamp() is not required, but is recommended both as a means of documenting the runtime conditions and as input for a planned utility which can rerun any task using the information recorded in the HISTORY block.

1.2.2 Output

Diagnostic output and other text messages must be able to be separated from the standard output stream to enable, e.g., piping FITS files between tasks. Developers should never write directly to stdout but should instead funnel screen output through the dedicated HEADAS streams. These streams are set up during task initialization and are controlled by environment variables. Task developers should never have to read or otherwise deal with these variables. The following methods for diagnostic output are currently available to developers writing tasks in C:

headas_printf(char *, ...) Operates exactly like the stdio version of printf but the stream will be directed to the location specified by the environment variable HEADASOUTPUT (if present).

headas_chat(int, char *, ...) Identical to headas_printf() except for an additional integer argument which specifies the threshold "chatter" level below which the message will be suppressed (depending on the runtime value of the chatter parameter, see discussion of "chatter" above).

fprintf(heaout, char *, ...) The "heaout" stream (which replaces stdout in HEADAS) may be written to directly, as shown.

printf(char *, ...) The usual stdio printf() routine can still be used (eg, in legacy code) but will be dynamically replaced by headas_printf() during compilation.

Fortran tasks should use the dedicated routines hdecho() and hdchat(). The former is exactly equivalent to the fcecho() routine in the FTOOLS package while the latter adds the chatter threshold argument as in headas_chat() above. Note that unlike the C versions above, formatting of the output strings must be done prior to calling hdecho()/hdchat(), e.g. via an internal write.

Future GUI development and/or other enhancements to HEASOFT will likely require that the standard error stream and parameter prompts be monitored and/or redirected as well. The environment variables HEADASERROR and HEADASPROMPT, respectively, control these but developers should not need to deal with them directly. C tasks may simply use fprintf(stderr, ...) to print error messages as usual, while Fortran tasks should use hderr() (which is exactly like the old FTOOLS fcerr() routine). As with hdecho(), the output error message must be constructed internally prior to calling hderr().

1.3 General Notes

All tasks should:

- Follow ANSI standards for maximum portability.
- Be written as a subprogram (not a main) which returns an integer status value.
- (For C only) Contain the following block of code near the top of the task subroutine:

```
#include "fitsio.h" /* assuming CFITSIO routines will be called */
#include "pil.h" /* assuming PIL routines will be called */
#include "headas.h"

#define TOOLSUB my_task_subroutine_name /* use actual subroutine name here */
#include "headas_main.c"
```

- Check return status after all CFITSIO and APE calls and use the relevant error reporting routine if status is non-zero.
- Register a task name and version number (see below).

1.4 Task Name and Version

Every HEADAS task should register its name and version number so that the information is available to other routines which may need it. A set of routines in the heautils library (see headas_toolname.c in the library inventory section, below) has been provided for this purpose. Each task should call set_toolname() and set_toolversion() to record the information and developers may retrieve the information via get_toolname()/get_toolversion() or by the simpler get_toolnamev() which returns both in a single string with name and version joined with an underscore. The Fortran equivalents are hdnameset(), hdverset(), hdnameget(), hdverget() and hdnamevget(). Note that a default name is recorded during task initialization based on the executable name. A default version number of "0.0" will likewise be used. These defaults will be superceded via calls at the task level to set_toolname() and set_toolversion().

1.5 CALDB Access

The HEASARC Calibration Database (CALDB) will be accessible to HEASOFT tasks, and missions choosing to use HEASOFT should plan on submitting their FITS calibration files for inclusion in the CALDB.

1.6 Scripting

All scripting should be done using Perl or Python. The HEASARC currently provides C/Perl interface libraries for CFITSIO, APE, and other core libraries. Development of Python libraries is ongoing.

Chapter 2

Overview of HEASOFT Libraries

The "heacore" directory contains source code for a number of libraries which are expected to be generally useful and mission-independent. These core libraries will automatically be built and available for linking by any or all of the tool packages via the normal hmake build process. Some of these libraries are developed at the HEASARC, while others are external packages which are simply redistributed as part of HEASOFT. Documentation for each library component may be found under the directory containing that component.

The software components of the directory "heacore" are:

- CFITSIO: A standard FITS file I/O library developed at the HEASARC
- CCFITS: An object-oriented interface to the CFITSIO library designed to make the capabilities of CFITSIO available to programmers working in C++
- APE: An IRAF-style parameter interface library, based on the PIL code developed for the INTEGRAL mission at the Integral Science Data Center (ISDC)
- READLINE: A standard input library which supports shell-style tab completion and command recall functions
- Astro-FITS-CFITSIO: A CFITSIO Perl module, distributed by Pete Ratzlaff (CfA)
- AST: The Starlink AST library for handling world coordinate systems in astronomy
- WCSLIB: Mark Calabretta's FITS World Coordinate System standard library
- FFTW: Library for computing the discrete Fourier transform (DFT)
- MPFIT: Uses the Levenberg-Marquardt technique to solve the least-squares problem
- heainit: HEADAS initialization functions callable at global scope
- heaio: HEADAS input/output library
- heautils: Assorted HEADAS utilities

- heasp: HEADAS utilities for spectra and responses
- heaapp: HEADAS facility for allowing client applications to select, initialize and use various available HEASoft libraries for standard support functions, such as parameter handling and output/error logging.
- ahfits: Mission-independent wrapper to CFITSIO, providing a more convenient way to perform common operations on FITS files
- ahgen: HEADAS library provides functions for string manipulation, file path handling, random number generation, and access to global clobber, buffer, and history states
- ahlog: HEADAS logging routines (wrapper to st_stream)
- st_stream: HEADAS library for formatting and manipulating output streams

More detailed documentation for a few of the HEADAS native heacore libraries follows below.

The software components of the directory "attitude" are:

- atFunctions: Library of attitude-related routines developed at ISAS
- aber: Support routines for computing aberration
- coord/coordfits: Additional routines for attitude, coordinate transformations, etc.
- ephemeris: Routines for calculating the positions of celestial bodies
- geomag: Code for calculating geomagnetic rigidity at a given time and position on an orbit
- param_wrappers: Wrappers for parameter file input/output
- random: Random number generator (but see also heautils below)

2.1 heainit

- `headas.h`

  Header file containing function prototypes, etc.

- `headas_init.c`

  Contains routines called internally – developers should not ever need to call these explicitly:

  - `int headas_init(int, char **)`

    Calls routines to initialize PIL and output streams. It also deals with standard HEADAS parameters (chatter, clobber, history).
  - `int headas_close()`

    Closes output streams and PIL.

  - `int hdIOInit()`

    Checks environment variables (e.g. HEADASOUTPUT, HEADASERROR) and sets up output streams.
  - `int hd_pil_err_logger(char *s)`

    Default PIL error logging routine. Simply prints to stderr.
  - `int headas_start_up(int, char **, const char *)`

    Set up logging.

- `headas_main.c`

  This file comprises the main program unit for every task. It calls headas_init() followed by the task subroutine itself and then calls headas_close() after the task completes. This main routine is not part of any library and must be explicitly included by all C tasks (via #include "headas_main.c"). For Fortran tasks it is automatically compiled and linked in by the Makefile.

2.2 heaio

- `headas_stdio.c`

  Contains routines used to write diagnostic and error output to the standard HEADAS output/error streams (replacing C stdio routines):

  - `int headas_printf(const char *, ...)`

    Replaces the stdio printf() with identical arguments. Text is written to the location specified by the HEADASOUTPUT environment variable (if present) instead of to stdout.
  - `int headas_chat(int, const char *, ...)`

    Same as headas_printf() but takes an integer argument which specifies the threshold chatter level below which the text will not be output.
  - `int pil_printf(const char *, ...)`

    A substitute for printf() for internal use by PIL only. Should not be called by tasks directly (NOT CURRENTLY USED).
  - `void headas_f77echo(const char *)`

    A Fortran-callable version of headas_printf(). It is called hdecho() from Fortran programs.
  - `void headas_f77err(const char *)`

    A Fortran-callable version of fprintf(stderr, ...). Called hderr() from Fortran. This routine will write to the stderr stream (which may have been redirected via the HEADASERROR environment variable).
  - `void headas_f77chat(int, const char *)`

    A Fortran-callable version of headas_chat(). It is called hdchat() from Fortran.

2.3 heautils

- `headas_utils.c`

  Contains utility routines:

  - `int headas_parstamp(fitsfile *, int)`

    Writes a block of HISTORY keywords into a FITS file header listing all the runtime parameter values. Arguments are a FITS file pointer and extension number. Callable from Fortran as hdparstamp(). *** DEPRECATED *** PLEASE USE HDpar_stamp (see below) INSTEAD ***

  - `int HDpar_stamp(fitsfile *, int, int *)`

    Writes a block of HISTORY keywords into a FITS file header listing all the runtime parameter values. Arguments are a FITS file pointer, extension number, and status pointer. Callable from Fortran as hdpar_stamp().

  - `char *hdbasename(char *)`

    Equivalent to the basename() function (returns the filename portion of an input pathname).

  - `int headas_clobberfile(char *)`

    Deletes the specified file if it already exists and if the clobber parameter for the current task is set to "yes". Callable from Fortran as hdclobber().

  - `float hd_ran2(long *)`

    Random number generator based on ran2() from Numerical Recipes in C, 2nd ed., p282. Returns a uniform random deviate between 0.0 and 1.0 (exclusive of the endpoint values). Call with a negative integer argument to initialize. Callable from Fortran as hd_ran2().

- `headas_toolname.c`

  Contains routines to get/set the name/version of the current task:

  - `void set_toolname(const char *)`

    Use this to register the task's name. The Fortran version is hdnameset().

  - `void get_toolname(char *)`

    Use this to retrieve the task's name. If it hasn't been set (via set_toolname()) a default name is determined from the name of the executable file. The Fortran version is hdnameget().

  - `void set_toolversion(const char *)`

    Use this to register a version number string for a task. The Fortran version is hdverset().

  - `void get_toolversion(char *)`

    Use this to retrieve a string containing the task's version number. If it hasn't been set (via set_toolversion()) a default version number string of "0.0" is returned. The Fortran version is hdverget().

  - `void get_toolnamev(char *)`
    Use this to retrieve a single string containing both the task's name and version number (joined by a "_"). The Fortran version is hdnamevget().

- `headas_history.c`

  Contains routines to get/set the value of the history parameter. Designed primarily for internal use and under normal circumstances should not be called by tasks explicitly.

  - `void get_history(int *)`
    This routine returns the value of the history parameter (if present) or "-1" if unspecified. Called by headas_parstamp(). Fortran version is hdghis().
  - `void set_history(int)`
    This registers the value of the history parameter. If it is called explicitly from a task it will override the user-specified value. Fortran version is hdphis().

- `headas_copykeys.c`

  Contains routine to copy non-critical keywords from one HDU to another..

  - ```
    int HDcopy_keys(fitsfile *inptr, fitsfile *outptr, int docomments,
                    int dohistory, int *status)
    ```
    Both the input and output FITS files should be positioned at the correct HDU. If docomments is true then COMMENT records will be copied and if dohistory is true then HISTORY records will be copied.

- `headas_polyfit.c`

  Contains routine to do a least-square polynomial fit.

  - `void HDpoly_fit(double * x, double * y, double * c, int n, int degree)`
    where x is the input n-element array of independent variables, y is the input n-element array of dependent variables, degree is the degree of the polynomial, and c is the output degree+1-element array of coefficients.

- `headas_sort.c`

  Contains routine to do a quick sort on the input array, returning sorted index (instead of data as with C qsort).

  - `void HDsort(float * base, int * index, int n)`
    where base is the input n-element unsorted data array and index is the input/output n-element array index.

- `headas_smooth.c`

  Contains routine to do a boxcar average on input data:

  - `void HDsmooth(float * input, float * output, int num, int width)`
    where input is the num-element unsmoothed array, output is the num-element smoothed array, and width is the width of the boxcar.
    ```

- `headas_svdfit.c`

  Contains routines used by HDpoly_fit (slightly modified versions of routines from Press, William H., Brian P. Flannery, Saul A Teukolsky and William T. Vetterling, 1986, "Numerical Recipes: The Art of Scientific Computing" (Fortran), Cambridge University Press.

- `headas_rand.c`

  Contains routines to generate (0,1) uniformly distributed pseudo-random numbers (using "Mersenne Twister" method).

  - `void HDmtInit(unsigned long int seed)`
    Initialize the algorithm. Must be called first.
  - `void HDmtFree()`
    Clear the algorithm.
  - `double HDmtRand()`
    Return a pseudo-random number.

- `headas_file_check.c`

  Checks for the existence and/or access mode of a file.

  - `void HDfile_check(const char *file_name, const char *open_mode)`
    Returns 0 if file is accessible in the given mode or 1 to indicate problems.

- `HDgtcalf.c`

  Routine to return the location of calibration data sets located in the CALDB.

  - ```
    int HDgtcalf(const char* tele, const char* instr, const char* detnam,
                 const char* filt, const char* codenam, const char* strtdate,
                 const char* strtime, const char* stpdate, const char* stptime,
                 const char* expr, int maxret, int filenamesize, char** filenam,
                 long* extno, char** online, int* nret, int* nfound, int* status)
    ```
    Returns a CALDB file based on input criteria.

- `headas_gti.c`

  Contains utility routines for manipulating Good Time Intervals.

  - `int HDgti_init(struct gti_struct *gti)`
    Initialize an already-existing GTI structure
  - `int HDgti_free(struct gti_struct *gti)`
    Deallocate memory associated with GTI structure
  - `int HDgti_copy(struct gti_struct *dest, struct gti_struct *src, int *status)`
    Deep copy GTI from one structure to another
  - ```
    int HDput_frac_time(fitsfile *fileptr, char *key, double vali, double valf,
                        int force, char *comment, int *status)
    ```
    Write (potentially) fractional time keyword from FITS header

- ```
  int HDgti_grow(struct gti_struct *gti, int new, int *status)
  ```
  Enlarge the storage of an existing GTI structure
- ```
  int HDgti_read(char *filename, struct gti_struct *gti, char *extname,
                 char *start, char *stop, struct gti_struct *refer_to,
                 fitsfile **fptr, int *status)
  ```
  Read a GTI extension from a FITS file
- ```
  int HDgti_write(fitsfile *fptr, struct gti_struct *gti, char *extname,
                  char *start, char *stop, int *status)
  ```
  Create a GTI extension and write it
- ```
  int HDgti_merge(int mode, struct gti_struct *gti, struct gti_struct *agti,
                  struct gti_struct *bgti, int *status)
  ```
  Merge two GTIs either using intersection or union
- ```
  int HDgti_clean(struct gti_struct *gti, struct gti_struct *ogti,
                  int *status)
  ```
  Clean a GTI by sorting, removing duplicates, overlaps
- ```
  int HDgti_where(struct gti_struct *gti, int ntimes, double *times,
                  int *segs, int *status)
  ```
  Which good time intervals a set of times falls into

2.4 heasp

HEASP is a C/C++/Python library to manipulate files related to spectroscopic analysis ie. PHA, RMF, ARF, and table model files. The library is described more fully in its own document. The C interface routines are summarized below. To use any of these include the Cheasp.h file which should be consulted for descriptions of the structs defined.

2.4.1 PHA files

- ```
  int ReadPHAtypeI(char *filename, long PHAnumber, struct PHA *phastruct)
  ```
  Read the type I PHA extension from a FITS file - if there are multiple PHA extensions then read the PHAnumber instance.

- ```
  int ReadPHAtypeII(char *filename, long PHAnumber, long NumberSpectra,
                    long *SpectrumNumber, struct PHA **phastructs)
  ```
  Read the type II PHA extension from a FITS file - if there are multiple PHA extensions then read the PHAnumber instance - within the typeII extension reads the spectra listed in the SpectrumNumber vector.

- ```
  int WritePHAtypeI(char *filename, struct PHA *phastruct)
  ```
  Write the type I PHA extension to a FITS file.
  ```

- `int WritePHAtypeII(char *filename, long NumberSpectra,`
  `                   struct PHA **phastructs)`

  Write the type II PHA extension to a FITS file.

- `int ReturnPHAtype(char *filename, long PHAnumber)`

  Return the type of a PHA extension.

- `void DisplayPHAtypeI(struct PHA *phastruct)`

  Write information about spectrum to stdout.

- `void DisplayPHAtypeII(long NumberSpectra, struct PHA **phastructs)`

  Write information about spectra to stdout.

- `int RebinPHA(struct PHA *phastruct, struct BinFactors *bin)`

  Rebin spectrum.

- `int CheckPHAcounts(char *filename, long PHAnumber)`

  Return 0 if COUNTS column exists and is integer or COUNTS column does not exist.

- `long ReturnNumberofSpectra(char *filename, long PHAnumber)`

  Return the number of spectra in a type II PHA extension.

2.4.2 RMF files

- `int ReadRMFMatrix(char *filename, long RMFnumber, struct RMF *rmf)`

  Read the RMF matrix from a FITS file - if there are multiple RMF extensions then read the RMFnumber instance.

- `int WriteRMFMatrix(char *filename, struct RMF *rmf)`

  Write the RMF matrix to a FITS file.

- `int ReadRMFEbounds(char *filename, long EBDnumber, struct RMF *rmf).`

  Read the RMF ebounds from a FITS file - if there are multiple EBOUNDS extensions then read the EBDnumber instance.

- `int WriteRMFEbounds(char *filename, struct RMF *rmf)`

  Write the RMF ebounds to a FITS file.

- `void DisplayRMF(struct RMF *rmf)`

  Write information about RMF to stdout.

- `void ReturnChannel(struct RMF *rmf, float energy, int NumberPhotons,`
  `                   long *channel)`

  Return the channel for a photon of the given input energy - draws random numbers to return NumberPhotons entries in the channel array.

- `void NormalizeRMF(struct RMF *rmf)`

  Normalize the response to unity in each energy.

- `void CompressRMF(struct RMF *rmf, float threshold)`

  Compress the response to remove all elements below the threshold value.

- `int RebinRMFChannel(struct RMF *rmf, struct BinFactors *bins)`

  Rebin the RMF in channel space.

- `int RebinRMFEnergy(struct RMF *rmf, struct BinFactors *bins)`

  Rebin the RMF in energy space.

- `void TransposeRMF(struct RMF *rmf, struct RMFchan *rmfchan)`

  Transpose the matrix.

- `float ReturnRMFElement(struct RMF *rmf, long channel, long energybin)`

  Return a single value from the matrix.

- ```
  float ReturnRMFchanElement(struct RMFchan *rmfchan, long channel,
                    long energybin)
  ```

  Return a single value from the transposed matrix.

- `int AddRMF(struct RMF *rmf1, struct RMF *rmf2)`

  Add rmf2 onto rmf1.

2.4.3 ARF

- `int ReadARF(char *filename, long ARFnumber, struct ARF *arf)`

  Read the effective areas from a FITS file - if there are multiple SPECRESP extensions then read the ARFFnumber instance.

- `int WriteARF(char *filename, struct ARF *arf)`

  Write the ARF to a FITS file.

- `void DisplayARF(struct ARF *arf)`

  Write information about ARF to stdout.

- `int AddARF(struct ARF *arf1, struct ARF *arf2)`

  Add arf2 onto arf1.

- `long MergeARFRMF(struct ARF *arf, struct RMF *rmf)`

  Multiply the ARF into the RMF.
  ```

2.4.4 Utility routines

- `int SPReadBinningFile(char *filename, struct BinFactors *binning)`

  Read an ascii file with binning factors and load the binning array.

- ```
  int SPSetGroupArray(int inputSize, struct BinFactors *binning,
                      int *groupArray)
  ```

  Set up a grouping array using the BinFactors structure.

- ```
  int SPBinArray(int inputSize, float *input, int *groupArray, int mode,
                 float *output)
  ```

  Bin an array using the information in the grouping array.

- `void SPsetCCfitsVerbose(int mode)`

  Set the CCfits verbose mode.

- `int SPcopyExtensions(char *infile, char *outfile)`

  Copy all HDUs which are not manipulated by this library.

- `int SPcopyKeywords(char *infile, char *outfile, char *hduname, int hdunumber)`

  Copy all non-critical keywords for the hdunumber instance of the extension hduname.

Chapter 3

HEADAS Makefiles

3.1 Introduction

HEADAS Makefiles are designed so that the most common steps needed to build and install software can be accomplished with minimal effort, but the flexibility exists to override and extend standard behavior. This is accomplished by including a "standard Makefile" in every HEADAS Makefile. In this way, developers can perform most functions simply by filling in definitions for a standard set of macros before the standard Makefile is included. Most of the time, there is no need to add explicit targets. In fact, explicit targets should be avoided whenever possible because it increases the possibility for breakage if/when the standard Makefile is changed.

Before giving a full exposition of all the features available through the standard Makefile, it will be useful to give a simple example. The most common situation for a developer is that he/she would like to add a new standalone compiled task to an existing (probably mission-specific) component. If this is the reader's need, understanding this example should suffice, and there will be no need for a deeper understanding of the more general options which are available.

Note that when using a standard HEASOFT installation, a Makefile generator utility ("hdmk") is available for use on the command line. hdmk prompts the developer for some basic information and then scans the current directory for source code files (.c, .cxx, .f90, etc.), scripts, parameter files, (.par), and help files (.html, .txt) which it uses to put together a first attempt ("Makefile.new").

3.2 Simplest case: A Makefile for a single task

Suppose one wishes to build a task called sample for the Swift mission from the files sample1.c and sample2.c. This task has a help file called sample.html, and a parameter file called sample.par. Furthermore, this task has a unit test in the form of a perl script named ut-sample, which produces a FITS file ut-sample.fits. The following Makefile would supply all the necessary targets and macros to make the task behave (build, install, clean, test, etc.) like all other HEADAS tasks:

```
# Component (mission) name. Developers need not change/delete this.
```

```
HD_COMPONENT_NAME = swift

# Software release version number. Developers need not change/delete this.
HD_COMPONENT_VERS =

# If this directory needs to build a task, list its name here.
HD_CTASK = sample

# C language source files (.c) to use for the task.
HD_CTASK_SRC_c = sample1.c sample2.c

# C flags to use in every compilation.
HD_CFLAGS = ${HD_STD_CFLAGS}

# Library flags to use when linking C task.
HD_CLIBS = ${HD_STD_CLIBS}

# Task(s) to be installed.
HD_INSTALL_TASKS = ${HD_CTASK}

# Help file(s) to install.
HD_INSTALL_HELP = ${HD_CTASK}.html

# Parameter file(s) to install.
HD_INSTALL_PFILES = ${HD_CTASK}.par

# Perl unit test script(s) to install.
HD_TEST_PERL_SCRIPTS = ut-sample

# Extra item(s) to remove during a clean or distclean.
HD_CLEAN = ut-sample.fits

# Include the standard HEADAS Makefile to do the real work.
include ${HD_STD_MAKEFILE}
```

Note that, as in all UNIX Makefiles, macro definitions must start at the beginning of a line, with no whitespace of any kind before the macro name.

This Makefile will in effect provide the following targets, which will have the stated behaviors:

- default: Build each source file to produce an object file, then link the object files to create the compiled task. This is also the target which will be "maked" if one invokes make with no explicit target.
- all: Perform the same actions as the default target, and in addition, "publish" the compiled task into the local "build" area.

- clean: Remove all object files and other build by-products, as well as the FITS file created by the unit test script.
- distclean: Remove the compiled task as well as the items removed by the clean target.
- install: Install the compiled task into the proper destination in the "installed" location.
- install-test: Install the test script into the proper destination in the "installed" location.

In general the best way to create a new Makefile for a directory is to copy a Makefile from the most similar example possible. This way, if there are component-specific features which are present in one component's Makefiles and not others, the new Makefile will remain as similar as possible to others in its same component group. In other words, when starting a new task for the SWIFT mission, start with another SWIFT task Makefile. For a new library for Hitomi, start with an existing Hitomi library Makefile, etc.

3.3 Standard Macros

Most if not all actions a Makefile needs to perform can be controlled entirely by setting one or more standard macros. In general, defining a macro to have a (non-trivial) body enables a particular behavior, while omitting a macro or defining it to have a trivial body (i.e. an empty definition) disables that behavior. This way, a single Makefile can in principle control many distinct build actions. Note that it is not always a good idea to structure Makefiles this way, but this flexibility exists to allow individual mission teams to lay out their directories in the way they choose.

3.3.1 Macros Pertaining To All Build Actions

- HD_CFLAGS: Compiler flags used in every C compilation, regardless of whether the object file is included in a library or a task. Usually this should be defined to be equal to ${HD_STD_CFLAGS}, which is set by hmake to be the correct flags for the current component, architecture and compiler.
- HD_CLIBS: Flags specifying the libraries in the link line for tasks linked with C. This includes path information (-L flags) to find the libraries as well as the library names themselves (-l flags). Usually this should be defined to be equal to ${HD_STD_CLIBS}, which is set by hmake to be the standard C link information for the given software component.
- HD_CXXFLAGS: Compiler flags used in every C++ compilation, regardless of whether the object file is included in a library or a task. Usually this should be defined to be equal to ${HD_STD_CFLAGS}, which is set by hmake to be the correct flags for the current component, architecture and compiler.
- HD_CXXLIBS: Flags specifying the libraries in the link line for tasks linked with C++. This includes path information (-L flags) to find the libraries as well as the library names themselves (-l flags). Usually this should be defined to be equal to ${HD_STD_CXXLIBS}, which is set by hmake to be the standard C++ link information for the given software component.

Linking Fortran code with C or C++ usually requires some additional libraries to be included at link time. The macro ${F77LIBS4C} is defined by hmake to hold this information for the current architecture and compiler. If Fortran code is involved in a task, it may be necessary to add this macro to the definition of HD_CLIBS and/or HD_CXXLIBS. Of course, if this ends up being a component-wide requirement, it is also possible for hmake to include the contents of the F77LIBS4C macro directly in HD_STD_CLIBS and/or HD_STD_CXXLIBS.

3.3.2 Macros Pertaining To Tasks

The standard Makefile provides direct support for building tasks which have source files in C, C++, and/or Fortran. In principle a task may arbitrarily blend these languages.

At present, it is required that a C or C++ "main" function be used, rather than a Fortran "program" statement. Consistent with this is the fact that the standard Makefile only supports linking using a supported C or C++ compiler. This is controlled with two families of macros. The first family controls tasks which are linked with C:

- HD_CTASK: The name of the executable which will be produced.
- HD_CTASK_SRC_c: A list of C source files with the suffix .c.
- HD_CTASK_SRC_f: A list of Fortran 77 source files with the suffix .f.
- HD_CTASK_SRC_f90: A list of Fortran 90/95 source files with the suffix .f90.
- HD_CTASK_SRC_f03: A list of Fortran 03 source files with the suffix .f03.

Note that there are no macros associated with C++ in this group. This is because if C++ code is mixed with C, it is required that the task be linked with a supported C++ compiler. Tasks linked with the C++ compiler are controlled with a second family of macros:

- HD_CXXTASK: The name of the executable which will be produced.
- HD_CXXTASK_SRC_C: A list of C++ source files with the suffix .C.
- HD_CXXTASK_SRC_cc: A list of C++ source files with the suffix .cc.
- HD_CXXTASK_SRC_cpp: A list of C++ source files with the suffix .cpp.
- HD_CXXTASK_SRC_cxx: A list of C++ source files with the suffix .cxx.
- HD_CXXTASK_SRC_c: A list of C source files with the suffix .c.
- HD_CXXTASK_SRC_f: A list of Fortran 77 source files with the suffix .f.
- HD_CXXTASK_SRC_f90: A list of Fortran 90/95 source files with the suffix .f90.
- HD_CXXTASK_SRC_f03: A list of Fortran 03 source files with the suffix .f03.

For C and Fortran source files, it is a very widespread convention to use .c and .f as the suffix, respectively. For C++ a number of different conventions are in use, which is why there are a number of macros whose names are distinguished by the suffix of the source files each macro includes.

If the HD_CTASK macro is defined, the HD_CTASK_SRC_c macro must contain at least one source file. If the HD_CXXTASK macro is defined, at least one of the HD_CXXTASK_SRC* macros which contain C or C++ source files must contain at least one source file.

When one or both of these macro families are properly defined, the standard Makefile will build the task(s) as part of the default target. First, all the source files in all the macros will be compiled, using the flags specified in the relevant HD_*FLAGS macro. Then the resulting object files will be linked to the libraries specified in the relevant HD_*LIBS macro.

3.3.3 Macros Pertaining To Libraries

The standard Makefile provides direct support for building shared and/or static libraries which have source files in C, C++ and/or Fortran. The following family of macros controls this process:

- HD_LIBRARY_ROOT: The root name of the library, without the prefix lib and without any suffix. For example, to build libmylib.so this macro would be simply mylib.
- HD_LIBRARY_SRC_c: A list of C source files with the suffix .c.
- HD_LIBRARY_SRC_C: A list of C++ source files with the suffix .C.
- HD_LIBRARY_SRC_cc: A list of C++ source files with the suffix .cc.
- HD_LIBRARY_SRC_cpp: A list of C++ source files with the suffix .cpp.
- HD_LIBRARY_SRC_cxx: A list of C++ source files with the suffix .cxx.
- HD_LIBRARY_SRC_f: A list of Fortran 77 source files with the suffix .f.
- HD_LIBRARY_SRC_f90: A list of Fortran 90/95 source files with the suffix .f90.
- HD_LIBRARY_SRC_f03: A list of Fortran 03 source files with the suffix .f03.

At least one of the source macros must contain at least one file if the HD_LIBRARY_ROOT macro is defined. With these macros properly defined, the standard Makefile will build the library as part of the default target. First all the source files listed in the source macros will be compiled, then they will be placed into the relevant shared or static library.

3.3.4 Macros Pertaining To Installation

Another family of macros determine which items will be installed during the "install" step. For maximum flexibility, no items are ever installed automatically. For example, in order to install a binary which was built by the current Makefile, it is necessary to list that binary explicitly in the list of binaries to install. In addition, a variation on install, called publish, is supported, in which items will be installed into the local build area by the "all" target, but not installed into the final installed area given by the configure prefix. The macros which control installation are:

- HD_INSTALL_EXTRA: Explicit targets which will be "maked" at publish and install time, and which take custom actions which do not fit into any category. This should be used sparingly.
- HD_INSTALL_ONLY_EXTRA: Explicit targets which will be "maked" only at install time, and which take custom actions which do not fit into any category. This should be used sparingly.
- HD_INSTALL_HEADERS: C and/or C++ header files to be installed into the top-level include/ directory. Not every header file necessarily needs to be installed. It is recommended that some care go into designing header files so that no more are installed than necessary. If the macro HD_INC_SUBDIR is also defined, the files will be installed in a subdirectory of the top-level include directory with the defined name instead (i.e. include/(HD_INC_SUBDIR)).
- HD_INSTALL_HELP: Help/documentation files to be installed into the top-level help/ directory.
- HD_INSTALL_LIBRARIES: Libraries to be installed into the top-level lib/ directory.
- HD_INSTALL_PERL_LIBS: Perl libraries to be installed into the top-level lib/perl directory. If the macro HD_PERL_SUBDIR is also defined, the libraries will be installed in a subdirectory of the top-level lib/perl with the defined name instead (i.e. lib/perl/(HD_PERL_SUBDIR)).
- HD_INSTALL_PERL_SCRIPTS: Perl scripts to be installed into the top-level scripts directory (currently the same as the top-level bin/ directory). At install time, Perl scripts will be edited to make sure that they use the version of Perl specified in the LHEA_PERL environment variable.
- HD_INSTALL_PFILES: Parameter files to be installed into the top-level syspfiles/ directory.
- HD_INSTALL_PYTHON_LIBS: Python libraries to be installed into the top-level lib/python directory. If the macro HD_PYTHON_SUBDIR is also defined, the libraries will be installed in a subdirectory of the top-level lib/python with the defined name instead (i.e. lib/python/(HD_PYTHON_SUBDIR)).
- HD_INSTALL_PYTHON_SCRIPTS: Python scripts to be installed into the top-level scripts directory (currently the same as the top-level bin/ directory).
- HD_INSTALL_REFDATA: Data files to be installed into the top-level refdata/ directory. If the macro HD_REFDATA_SUBDIR is also defined, the files will be installed in a subdirectory of the top-level refdata directory with the defined name instead (i.e. refdata/(HD_REFDATA_SUBDIR)).
- HD_INSTALL_TASKS: Compiled tasks to be installed into the top-level bin/ directory.
- HD_INSTALL_SHELL_SCRIPTS: Shell scripts to be installed into the top-level scripts directory (currently the same as the top-level /bin directory).

- HD_INSTALL_XML: XML files to be installed into the top-level xml directory. If the macro HD_XML_SUBDIR is also defined, the files will be installed in a subdirectory of the top-level xml directory with the defined name instead (i.e. xml/(HD_XML_SUBDIR)).

All items specified in the above macros will also be installed automatically as part of the "publish" step. In addition, the publish step publishes the items listed in the following macros:

- HD_PUBLISH_HEADERS: C and/or C++ header files to be published into the local build include/ directory. Not every header file necessarily needs to be published.
- HD_PUBLISH_HELP: Help/documentation files to be published into the local build help/ directory.
- HD_PUBLISH_LIBRARIES: Libraries to be published into the local build lib/ directory.
- HD_PUBLISH_PERL_LIBS: Perl libraries to be published into the local build lib/perl directory.
- HD_PUBLISH_PERL_SCRIPTS: Perl scripts to be published into the local build scripts directory (same as the local build tasks directory). At publish time, Perl scripts will be edited to make sure that they use the version of Perl specified in the LHEA_PERL environment variable.
- HD_PUBLISH_PFILES: Parameter files to be published into the local build syspfiles/ directory.
- HD_PUBLISH_REFDATA: Data files to be published into the local build refdata/ directory.
- HD_PUBLISH_TASKS: Compiled tasks to be published into the local build bin/ directory.
- HD_PUBLISH_SHELL_SCRIPTS: Shell scripts to be published into the local build scripts directory (same as the local build tasks directory).

3.3.5 Macros Pertaining To Subdirectories

Subdirectories into which the current Makefile should recurse when performing all standard actions (default, all, install, publish, clean, distclean) can be specified in the HD_SUBDIRS macro.

Chapter 4

HEADAS Error Handling Facility

4.1 Introduction

The heautils library contains the HEADAS error handling functions, which provide a means by which calling code can set and clear error conditions, as well as manage supporting information about errors. When an error condition is first encountered, a call to the HDerror_throw function sets the error state of the error handler, and adds an optional message. Subsequent calls to the HDerror_hint function can be used to place additional messages on the error stack. Each of these can produce an error message containing the file name and line number where the error occurred. When used, this feature in effect provides a stack trace of the error, but without using a debugger. The HDerror_reset function can be used to reset the error handler to a non-error condition. In this case, the stack is erased, and no error code is retained by the stack.

In addition, the HEADAS error facility provides a means of establishing default error messages for error codes. Several error ranges already exist within HEADAS. For example, error numbers between 1 and 999 are associated with CFITSIO errors, while -3999 to -3000 are associated with APE. Each of these error codes has an associated standard error message. The HEADAS error handler contains a map of each error code to its message. Also there is a means to add new error maps, in order to provide standard error messsages for new software components within different numeric code ranges. These maps may be used as a task exits to print the standard message if no custom message is available.

For convenience, several macros which wrap these functions are also provided. The macros HD_ERR_SET, HD_ERR_THROW, and HD_ERR_HINT use the functions HDerror_throw and HDerror_hint to do their work. There were two reasons for defining these macros. One is that the macros can check their status arguments before calling their underlying functions, thus eliminating the overhead of a function call unless an error actually occurs. The second is that the functions HDerror_throw and HDerror_hint have arguments which can be used to give the location (source file and line number) of the error. The macros listed above do not take these arguments, but use the __FILE__ and __LINE__ macros to create them when they call the appropriate error function.

4.2 HEADAS Error Handling API

The HEADAS Error Handling API contains a number of functions and macros. The functions are described first, but in practice the macros should be used whenever possible, because they are easier to use and offer performance advantages. Central concepts are that each error message has an associated integer error number, and that the error handler as a whole also has an error number, which corresponds to the first error which was encountered.

1 int HDerror_get(void)

Returns the current value of the error handler's integer error status variable.

2 int HDerror_throw(const char * msg, const char * fileName, int line, int errNum)

This function uses the errNum argument to determine if an error has occurred, and if one has, changes the state of the error handler as appropriate to include the given information about the error. The argument msg is a string describing the error. If it is NULL no text message will be added. Arguments fileName and line are the file name and line number where the error was thrown. If fileName is NULL this information will not be included. The argument errNum is used in conjunction with the error handler's internal error status as follows: if either errNum or the error handler's internal status is non-zero, the information about the error will be added to the error handler's error message stack. If the error handler's internal status is zero, it will be set equal to errNum. Otherwise, the error handler's internal status will not be changed.

This function cannot be used to reset the error handler's internal status to zero after an error. To do this, use HDerror_reset().

3 int HDerror_hint(const char * msg, const char * fileName, int line, int errNum)

This function is similar to HDerror_throw, described above. The arguments given may be used to add to the description of an error. The difference between the two functions is that HDerror_hint never affects the overall error status of the error handler. HDerror_hint may thus not be used to create an error condition, only to comment on an existing error condition.

4 int HDerror_reset(void)

Resets the error handler's status to HD_OK and clears the error message stack.

5 int HDerror_get_stack(const char** stack)

This function returns a NULL-terminated array of const char* pointers which contains the current stack of messages. The standard HEADAS shutdown code which executes just before a task exits will print this stack to stderr if the task exits with non-zero status.

6 void HDerror_dump_silence(int silent)

Sets a silent mode which suppresses all reporting of errors. Calling it with a non-0 argument activates silent mode, while calling it with an argument of 0 sets non-silent (normal) mode. Note that silencing error reporting does not prevent error messages from being added to the error stack; rather it merely prevents these messages from being displayed.

This function should not be used in general. It is present only for the benefit of very unusual tools which return a non-0 exit status to indicate something other than an error. For example, ftdiff uses a non-0 status to indicate that it detected differences between the two input files.

7 int HDerror_dump_is_silent(void)

Returns the current silent mode of the error reporting mechanism. A non-0 value indicates errors are silenced, while a value of 0 indicates errors will be reported.

8 HD_OK

Macro whose value is 0, used throughout HEADAS software to indicate normal (non-error) status.

9 HD_ERROR_GET()

Macro which simply calls HDerror_get(). This is provided mainly for completeness and to allow a consistent look and feel if other similar macros are used.

10 HD_ERROR_THROW(MSG, STATUS)

This macro is provided for more convenient access to the function HDerror_throw(). This macro simply calls the function, using MSG for the input argument msg, STATUS for the input argument errNum, and filling in the fileName and line arguments using ANSI C's __FILE__ and __LINE__ macros.

11 HD_ERROR_SET(STATUS)

This macro is provided for more convenient access to the function HDerror_throw(). This macro simply calls the function, using NULL for the input argument msg, STATUS for the input argument errNum, and filling in the fileName and line arguments using ANSI C's __FILE__ and __LINE__ macros.

12 HD_ERROR_HINT(MSG, STATUS)

This macro is provided for more convenient access to the function HDerror_hint(). This macro simply calls the function, using MSG for the argument msg, STATUS for the argument errNum, and filling in the fileName and line arguments using ANSI C's __FILE__ and __LINE__ macros.