WCSLIB

1 WCSLIB 7.11 and PGSBOX 7.11

1.1 Contents

- Introduction

- FITS-WCS and related software

- Overview of WCSLIB

- WCSLIB data structures

- Memory management

- Diagnostic output

- Vector API

- Thread-safety

- Limits

- Example code, testing and verification

- WCSLIB Fortran wrappers

- PGSBOX

- WCSLIB version numbers

1.2 Copyright

2 Introduction

WCSLIB is a C library, supplied with a full set of Fortran wrappers, that implements the "World Coordinate System" (WCS) standard in FITS (Flexible Image Transport System). It also includes a PGPLOT-based routine, PGSBOX, for drawing general curvilinear coordinate graticules, and also a number of utility programs.

The FITS data format is widely used within the international astronomical community, from the radio to gamma-ray regimes, for data interchange and archive, and also increasingly as an online format. It is described in

- "Definition of The Flexible Image Transport System (FITS)", FITS Standard, Version 3.0, 2008 July 10.

available from the FITS Support Office at `http://fits.gsfc.nasa.gov`.

The FITS WCS standard is described in

- "Representations of world coordinates in FITS" (Paper I), Greisen, E.W., & Calabretta, M.R. 2002, A&A, 395, 1061-1075
- "Representations of celestial coordinates in FITS" (Paper II), Calabretta, M.R., & Greisen, E.W. 2002, A&A, 395, 1077-1122
- "Representations of spectral coordinates in FITS" (Paper III), Greisen, E.W., Calabretta, M.R., Valdes, F.G., & Allen, S.L. 2006, A&A, 446, 747
- "Representations of distortions in FITS world coordinate systems", Calabretta, M.R. et al. (WCS Paper IV, draft dated 2004/04/22), available from `http://www.atnf.csiro.au/people/Mark.Calabretta`
- "Mapping on the HEALPix Grid" (HPX, Paper V), Calabretta, M.R., & Roukema, B.F. 2007, MNRAS, 381, 865
- "Representing the 'Butterfly' Projection in FITS: Projection Code XPH" (XPH, Paper VI), Calabretta, M.R., & Lowe, S.R. 2013, PASA, 30, e050
- "Representations of time coordinates in FITS: Time and relative dimension in space" (Paper VII), Rots, A.H., Bunclark, P.S., Calabretta, M.R., Allen, S.L., Manchester R.N., & Thompson, W.T. 2015, A&A, 574, A36

Reprints of all published papers may be obtained from NASA's Astrophysics Data System (ADS), `http://adsabs.harvard.edu/`. Reprints of Papers I, II (including HPX & XPH), and III are available from `http://www.atnf.csiro.au/people/Mark.Calabretta`. This site also includes errata and supplementary material for Papers I, II and III.

Additional information on all aspects of FITS and its various software implementations may be found at the FITS Support Office `http://fits.gsfc.nasa.gov`.

3 FITS-WCS and related software

Several implementations of the FITS WCS standards are available:

- The WCSLIB software distribution (i.e. this library) may be obtained from `http://www.atnf.csiro.au/people/Mark.Calabretta/WCS/`. The remainder of this manual describes its use.

  WCSLIB is included in the Astrophysics Source Code Library (**ASCL** `https://ascl.net`) as record ascl:1108.003 (`https://ascl.net/1108.003`), and in the Astrophysics Data System (**ADS** `https://ui.adsabs.harvard.edu`) with bibcode 2011ascl.soft08003C (`https://ui.adsabs.harvard.edu/abs/2011ascl.soft08003C`).

- **wcstools**, developed by Jessica Mink, may be obtained from http://tdc-www.harvard.↩edu/software/wcstools/.

  ASCL: https://ascl.net/1109.015
  ADS: https://ui.adsabs.harvard.edu/abs/2011ascl.soft09015M

- **AST**, developed by David Berry within the U.K. Starlink project, http://www.starlink.↩ac.uk/ast/ and now supported by JAC, Hawaii http://starlink.jach.hawaii.↩edu/starlink/. A useful utility for experimenting with FITS WCS descriptions (similar to *wcsgrid*) is also provided; go to the above site and then look at the section entitled "FITS-WCS Plotting Demo".

  ASCL: https://ascl.net/1404.016
  ADS: https://ui.adsabs.harvard.edu/abs/2014ascl.soft04016B

- **SolarSoft**, http://sohowww.nascom.nasa.gov/solarsoft/, primarily an IDL-based system for analysis of Solar physics data, contains a module written by Bill Thompson oriented towards Solar coordinate systems, including spectral, http://sohowww.nascom.nasa.↩gov/solarsoft/gen/idl/wcs/.

  ASCL: https://ascl.net/1208.013
  ADS: https://ui.adsabs.harvard.edu/abs/2012ascl.soft08013F

- The IDL Astronomy Library, http://idlastro.gsfc.nasa.gov/, contains an independent implementation of FITS-WCS in IDL by Rick Balsano, Wayne Landsman and others. See http↩://idlastro.gsfc.nasa.gov/contents.html#C5.

Python wrappers to WCSLIB are provided by

- The **Kapteyn Package** http://www.astro.rug.nl/software/kapteyn/ by Hans Terlouw and Martin Vogelaar.

  ASCL: https://ascl.net/1611.010
  ADS: https://ui.adsabs.harvard.edu/abs/2016ascl.soft11010T

- **pywcs**, http://stsdas.stsci.edu/astrolib/pywcs/ by Michael Droettboom, which is distributed within Astropy, https://www.astropy.org.

  ASCL (Astropy): https://ascl.net/1304.002
  ADS (Astropy): https://ui.adsabs.harvard.edu/abs/2013ascl.soft04002G

Java is supported via

- CADC/CCDA Java Native Interface (JNI) bindings to WCSLIB 4.2 http://www.cadc-ccda.↩hia-iha.nrc-cnrc.gc.ca/cadc/source/ by Patrick Dowler.

and Javascript by

- **wcsjs**, https://github.com/astrojs/wcsjs, a port created by Amit Kapadia using Emscripten, an LLVM to Javascript compiler. wcsjs provides a code base for running WCSLIB on web browsers.

Julia wrappers (https://en.wikipedia.org/wiki/Julia_(programming_language)) are provided by

- **WCS.jl**, https://github.com/JuliaAstro/WCS.jl, a component of Julia Astro, https↩://github.com/JuliaAstro.

An interface for the R programming language (`https://en.wikipedia.org/wiki/R_(programming↩_language)`) is available at

- **Rwcs**, `https://github.com/asgr/Rwcs/` by Aaron Robotham.

Recommended WCS-aware FITS image viewers:

- Bill Joye's **DS9**, `http://hea-www.harvard.edu/RD/ds9/`, and
  ASCL: `https://ascl.net/0003.002`
  ADS: `https://ui.adsabs.harvard.edu/abs/2000ascl.soft03002S`

- **Fv** by Pan Chai, `http://heasarc.gsfc.nasa.gov/ftools/fv/`.
  ASCL: `https://ascl.net/1205.005`
  ADS: `https://ui.adsabs.harvard.edu/abs/2012ascl.soft05005P`

both handle 2-D images.

Currently (2013/01/29) I know of no image viewers that handle 1-D spectra properly nor multi-dimensional data, not even multi-dimensional data with only two non-degenerate image axes (please inform me if you know otherwise).

Pre-built WCSLIB packages are available, generally a little behind the main release (this list will probably be stale by the time you read it, best do a web search):

- archlinux (tgz), `https://www.archlinux.org/packages/extra/i686/wcslib`.

- Debian (deb), `http://packages.debian.org/search?keywords=wcslib`.

- Fedora (RPM), `https://admin.fedoraproject.org/pkgdb/package/wcslib`.

- Fresh Ports (RPM), `http://www.freshports.org/astro/wcslib`.

- Gentoo, `http://packages.gentoo.org/package/sci-astronomy/wcslib`.

- Homebrew (MacOSX), `https://github.com/Homebrew/homebrew-science`.

- RPM (general) `http://rpmfind.net/linux/rpm2html/search.php?query=wcslib`, `http://www.rpmseek.com/rpm-pl/wcslib.html`.

- Ubuntu (deb), `https://launchpad.net/ubuntu/+source/wcslib`.

Bill Pence's general FITS IO library, **CFITSIO** is available from `http://heasarc.gsfc.nasa.↩gov/fitsio/`. It is used optionally by some of the high-level WCSLIB test programs and is required by two of the utility programs.

ASCL: `https://ascl.net/1010.001`
ADS: `https://ui.adsabs.harvard.edu/abs/2010ascl.soft10001P`

PGPLOT, Tim Pearson's Fortran plotting package on which PGSBOX is based, also used by some of the WCSLIB self-test suite and a utility program, is available from `http://astro.caltech.edu/~tjp/pgplot/`.

ASCL: `https://ascl.net/1103.002`
ADS: `https://ui.adsabs.harvard.edu/abs/2011ascl.soft03002P`

4 Overview of WCSLIB

WCSLIB is documented in the prologues of its header files which provide a detailed description of the purpose of each function and its interface (this material is, of course, used to generate the doxygen manual). Here we explain how the library as a whole is structured. We will normally refer to WCSLIB 'routines', meaning C functions or Fortran 'subroutines', though the latter are actually wrappers implemented in C.

WCSLIB is layered software, each layer depends only on those beneath; understanding WCSLIB first means understanding its stratigraphy. There are essentially three levels, though some intermediate levels exist within these:

- The **top layer** consists of routines that provide the connection between FITS files and the high-level WCSLIB data structures, the main function being to parse a FITS header, extract WCS information, and copy it into a wcsprm struct. The lexical parsers among these are implemented as Flex descriptions (source files with .l suffix) and the C code generated from these by Flex is included in the source distribution.

    - wcshdr.h,c – Routines for constructing wcsprm data structures from information in a FITS header and conversely for writing a wcsprm struct out as a FITS header.
    - wcspih.l – Flex implementation of wcspih(), a lexical parser for WCS "keyrecords" in an image header. A **keyrecord** (formerly called "card image") consists of a **keyword**, its value - the **keyvalue -** and an optional comment, the **keycomment.**
    - wcsbth.l – Flex implementation of wcsbth() which parses binary table image array and pixel list headers in addition to image array headers.
    - getwcstab.h,c – Implementation of a -TAB binary table reader in CFITSIO.

    A generic FITS header parser is also provided to handle non-WCS keyrecords that are ignored by wcspih():

    - fitshdr.h,l – Generic FITS header parser (not WCS-specific).

    The philosophy adopted for dealing with non-standard WCS usage is to translate it at this level so that the middle- and low-level routines need only deal with standard constructs:

    - wcsfix.h,c – Translator for non-standard FITS WCS constructs (uses wcsutrne()).
    - wcsutrn.l – Lexical translator for non-standard units specifications.

    As a concrete example, within this layer the `CTYPEia` keyvalues would be extracted from a FITS header and copied into the *ctype*[] array within a wcsprm struct. None of the header keyrecords are interpreted.

- The **middle layer** analyses the WCS information obtained from the FITS header by the top-level routines, identifying the separate steps of the WCS algorithm chain for each of the coordinate axes in the image. It constructs the various data structures on which the low-level routines are based and invokes them in the correct sequence. Thus the wcsprm struct is essentially the glue that binds together the low-level routines into a complete coordinate description.

    - wcs.h,c – Driver routines for the low-level routines.
    - wcsunits.h,c – Unit conversions (uses wcsulexe()).
    - wcsulex.l – Lexical parser for units specifications.

    To continue the above example, within this layer the *ctype*[] keyvalues in a wcsprm struct are analysed to determine the nature of the coordinate axes in the image.

- Applications programmers who use the top- and middle-level routines generally need know nothing about the **low-level routines**. These are essentially mathematical in nature and largely independent of FITS itself. The mathematical formulae and algorithms cited in the WCS Papers, for example the spherical projection equations of Paper II and the lookup-table methods of Paper III, are implemented by the routines in this layer, some of which serve to aggregate others:

- **cel.h**,c – Celestial coordinate transformations, combines prj.h,c and sph.h,c.
- **spc.h**,c – Spectral coordinate transformations, combines transformations from spx.h,c.

The remainder of the routines in this level are independent of everything other than the grass-roots mathematical functions:

- **lin.h**,c – Linear transformation matrix.
- **dis.h**,c – Distortion functions.
- **log.h**,c – Logarithmic coordinates.
- **prj.h**,c – Spherical projection equations.
- **sph.h**,c – Spherical coordinate transformations.
- **spx.h**,c – Basic spectral transformations.
- **tab.h**,c – Coordinate lookup tables.

As the routines within this layer are quite generic, some, principally the implementation of the spherical projection equations, have been used in other packages (AST, wcstools) that provide their own implementations of the functionality of the top and middle-level routines.

- At the **grass-roots level** there are a number of mathematical and utility routines.

When dealing with celestial coordinate systems it is often desirable to use an angular measure that provides an exact representation of the latitude of the north or south pole. The WCSLIB routines use the following trigonometric functions that take or return angles in degrees:

- cosd(), sind(), sincosd(), tand(), acosd(), asind(), atand(), atan2d()

These "trigd" routines are expected to handle angles that are a multiple of $90°$ returning an exact result. Some C implementations provide these as part of a system library and in such cases it may (or may not!) be preferable to use them. wcstrig.c provides wrappers on the standard trig functions based on radian measure, adding tests for multiples of $90°$.

However, wcstrig.h also provides the choice of using preprocessor macro implementations of the trigd functions that don't test for multiples of $90°$ (compile with `-DWCSTRIG_MACRO`). These are typically 20% faster but may lead to problems near the poles.

- **wcsmath.h** – Defines mathematical and other constants.
- **wcstrig.h**,c – Various implementations of trigd functions.
- **wcsutil.h**,c – Simple utility functions for string manipulation, etc. used by WCSLIB.

Complementary to the C library, a set of wrappers are provided that allow all WCSLIB C functions to be called by Fortran programs, see below.

Plotting of coordinate graticules is one of the more important requirements of a world coordinate system. WCSLIB provides a PGPLOT-based subroutine, PGSBOX (Fortran), which handles general curvilinear coordinates via a user-supplied function - `PGWCSL` provides the interface to WCSLIB. A C wrapper, *cpgsbox()*, is also provided, see below.

Several utility programs are distributed with WCSLIB:

- *wcsgrid* extracts the WCS keywords for an image from the specified FITS file and uses *cpgsbox()* to plot a 2-D coordinate graticule for it. It requires WCSLIB, PGSBOX and CFITSIO.

- *wcsware* extracts the WCS keywords for an image from the specified FITS file and constructs wcsprm structs for each coordinate representation found. The structs may then be printed or used to transform pixel coordinates to world coordinates. It requires WCSLIB and CFITSIO.

- *HPXcvt* reorganises HEALPix data into a 2-D FITS image with HPX coordinate system. The input data may be stored in a FITS file as a primary image or image extension, or as a binary table extension. Both NESTED and RING pixel indices are supported. It uses CFITSIO.

- *fitshdr* lists headers from a FITS file specified on the command line, or else on stdin, printing them as 80-character keyrecords without trailing blanks. It is independent of WCSLIB.

5 WCSLIB data structures

The WCSLIB routines are based on data structures specific to them: wcsprm for the wcs.h,c routines, celprm for cel.h,c, and likewise spcprm, linprm, prjprm, tabprm, and disprm, with struct definitions contained in the corresponding header files: wcs.h, cel.h, etc. The structs store the parameters that define a coordinate transformation and also intermediate values derived from those parameters. As a high-level object, the wcsprm struct contains linprm, tabprm, spcprm, and celprm structs, and in turn the linprm struct contains disprm structs, and the celprm struct contains a prjprm struct. Hence the wcsprm struct contains everything needed for a complete coordinate description.

Applications programmers who use the top- and middle-level routines generally only need to pass wcsprm structs from one routine that fills them to another that uses them. However, since these structs are fundamental to WCSLIB it is worthwhile knowing something about the way they work.

Three basic operations apply to all WCSLIB structs:

- Initialize. Each struct has a specific initialization routine, e.g. wcsinit(), celini(), spcini(), etc. These allocate memory (if required) and set all struct members to default values.

- Fill in the required values. Each struct has members whose values must be provided. For example, for wcsprm these values correspond to FITS WCS header keyvalues as are provided by the top-level header parsing routine, wcspih().

- Compute intermediate values. Specific setup routines, e.g. wcsset(), celset(), spcset(), etc., compute intermediate values from the values provided. In particular, wcsset() analyses the FITS WCS keyvalues provided, fills the required values in the lower-level structs contained in wcsprm, and invokes the setup routine for each of them.

Each struct contains a *flag* member that records its setup state. This is cleared by the initialization routine and checked by the routines that use the struct; they will invoke the setup routine automatically if necessary, hence it need not be invoked specifically by the application programmer. However, if any of the required values in a struct are changed then either the setup routine must be invoked on it, or else the *flag* must be zeroed to signal that the struct needs to be reset.

The initialization routine may be invoked repeatedly on a struct if it is desired to reuse it. However, the *flag* member of structs that contain allocated memory (wcsprm, linprm, tabprm, and disprm) must be set to -1 before the first initialization to initialize memory management, but not subsequently or else memory leaks will result.

Each struct has one or more service routines: to do deep copies from one to another, to print its contents, and to free allocated memory. Refer to the header files for a detailed description.

6   Memory management

The initialization routines for certain of the WCSLIB data structures allocate memory for some of their members:

- wcsinit() optionally allocates memory for the *crpix*, *pc*, *cdelt*, *crval*, *cunit*, *ctype*, *pv*, *ps*, *cd*, *crota*, *colax*, *cname*, *crder*, and *csyer* arrays in the wcsprm struct (using lininit() for certain of these). Note that wcsinit() does not allocate memory for the *tab* array - refer to the usage notes for wcstab() in wcshdr.h. If the *pc* matrix is not unity, wcsset() (via linset()) also allocates memory for the *piximg* and *imgpix* arrays.

- lininit(): optionally allocates memory for the *crpix*, *pc*, and *cdelt* arrays in the linprm struct. If the *pc* matrix is not unity, linset() also allocates memory for the *piximg* and *imgpix* arrays. Typically these would be used by wcsinit() and wcsset().

- tabini(): optionally allocates memory for the *K*, *map*, *crval*, *index*, and *coord* arrays (including the arrays referenced by *index*[]) in the tabprm struct. tabmem() takes control of any of these arrays that may have been allocated by the user, specifically in that tabfree() will then free it. tabset() also allocates memory for the *sense*, *p0*, *delta* and *extrema* arrays.

- disinit(): optionally allocates memory for the *dtype*, *dp*, and *maxdis* arrays. disset() also allocates memory for a number of arrays that hold distortion parmeters and intermediate values: *axmap*, *Nhat*, *offset*, *scale*, *iparm*, and *dparm*, and also several private work arrays: *disp2x*, *disx2p*, and *tmpmem*.

The caller may load data into these arrays but must not modify the struct members (i.e. the pointers) themselves or else memory leaks will result.

wcsinit() maintains a record of memory it has allocated and this is used by wcsfree() which wcsinit() uses to free any memory that it may have allocated on a previous invokation. Thus it is not necessary for the caller to invoke wcsfree() separately if wcsinit() is invoked repeatedly on the same wcsprm struct. Likewise, wcsset() deallocates memory that it may have allocated on a previous invokation. The same comments apply to lininit(), linfree(), and linset(), to tabini(), tabfree(), and tabset(), and to disinit(), disfree() and disset().

A memory leak will result if a wcsprm, linprm, tabprm, or disprm struct goes out of scope before the memory has been *free'd*, either by the relevant routine, wcsfree(), linfree(), tabfree(), or disfree() or otherwise. Likewise, if one of these structs itself has been *malloc'd* and the allocated memory is not *free'd* when the memory for the struct is *free'd*. A leak may also arise if the caller interferes with the array pointers in the "private" part of these structs.

Beware of making a shallow copy of a wcsprm, linprm, tabprm, or disprm struct by assignment; any changes made to allocated memory in one would be reflected in the other, and if the memory allocated for one was *free'd* the other would reference unallocated memory. Use the relevant routine instead to make a deep copy: wcssub(), lincpy(), tabcpy(), or discpy().

7   Diagnostic output

All WCSLIB functions return a status value, each of which is associated with a fixed error message which may be used for diagnostic output. For example

```
int status;
struct wcsprm wcs;

...

if ((status = wcsset(&wcs)) {
  fprintf(stderr, "ERROR %d from wcsset(): %s.\n", status, wcs_errmsg[status]);
  return status;
}
```

This might produce output like

```
ERROR 5 from wcsset(): Invalid parameter value.
```

The error messages are provided as global variables with names of the form *cel_errmsg*, *prj_errmsg*, etc. by including the relevant header file.

As of version 4.8, courtesy of Michael Droettboom (pywcs), WCSLIB has a second error messaging system which provides more detailed information about errors, including the function, source file, and line number where the error occurred. For example,

```
struct wcsprm wcs;

/* Enable wcserr and send messages to stderr. */
wcserr_enable(1);
wcsprintf_set(stderr);

...

if (wcsset(&wcs) {
  wcsperr(&wcs);
  return wcs.err->status;
}
```

In this example, if an error was generated in one of the prjset() functions, wcsperr() would print an error traceback starting with wcsset(), then celset(), and finally the particular projection-setting function that generated the error. For each of them it would print the status return value, function name, source file, line number, and an error message which may be more specific and informative than the general error messages reported in the first example. For example, in response to a deliberately generated error, the `twcs` test program, which tests wcserr among other things, produces a traceback similar to this:

```
ERROR 5 in wcsset() at line 1564 of file wcs.c:
  Invalid parameter value.
ERROR 2 in celset() at line 196 of file cel.c:
  Invalid projection parameters.
ERROR 2 in bonset() at line 5727 of file prj.c:
  Invalid parameters for Bonne's projection.
```

Each of the structs in WCSLIB includes a pointer, called *err*, to a wcserr struct. When an error occurs, a struct is allocated and error information stored in it. The wcserr pointers and the memory allocated for them are managed by the routines that manage the various structs such as wcsinit() and wcsfree().

wcserr messaging is an opt-in system enabled via wcserr_enable(), as in the example above. If enabled, when an error occurs it is the user's responsibility to free the memory allocated for the error message using wcsfree(), celfree(), prjfree(), etc. Failure to do so before the struct goes out of scope will result in memory leaks (if execution continues beyond the error).

8 Vector API

WCSLIB's API is vector-oriented. At the least, this allows the function call overhead to be amortised by spreading it over multiple coordinate transformations. However, vector computations may provide an opportunity for caching intermediate calculations and this can produce much more significant efficiencies. For example, many of the spherical projection equations are partially or fully separable in the mathematical sense, i.e. $(x, y) = f(\phi)g(\theta)$, so if θ was invariant for a set of coordinate transformations then $g(\theta)$ would only need to be computed once. Depending on the circumstances, this may well lead to speedups of a factor of two or more.

WCSLIB has two different categories of vector API:

- Certain steps in the WCS algorithm chain operate on coordinate vectors as a whole rather than particular elements of it. For example, the linear transformation takes one or more pixel coordinate vectors, multiples by the transformation matrix, and returns whole intermediate world coordinate vectors.
  The routines that implement these steps, wcsp2s(), wcss2p(), linp2x(), linx2p(), tabx2s(), tabs2x(), disp2x() and disx2p() accept and return two-dimensional arrays, i.e. a number of coordinate vectors. Because WCSLIB permits these arrays to contain unused elements, three parameters are needed to describe them:

  - *naxis:* the number of coordinate elements, as per the FITS `NAXIS` or `WCSAXES` keyvalues,

  - *ncoord:* the number of coordinate vectors,

  - *nelem:* the total number of elements in each vector, unused as well as used. Clearly, *nelem* must equal or exceed *naxis*. (Note that when *ncoord* is unity, *nelem* is irrelevant and so is ignored. It may be set to 0.)

  ncoord and *nelem* are specified as function arguments while *naxis* is provided as a member of the wcsprm (or linprm or disprm) struct.
  For example, wcss2p() accepts an array of world coordinate vectors, *world[ncoord][nelem]*. In the following example, *naxis* = 4, *ncoord* = 5, and *nelem* = 7:

  ```
  s1  x1  y1  t1  u   u   u
  s2  x2  y2  t2  u   u   u
  s3  x3  y3  t3  u   u   u
  s4  x4  y4  t4  u   u   u
  s5  x5  y5  t5  u   u   u
  ```

  where *u* indicates unused array elements, and the array is laid out in memory as

  ```
  s1  x1  y1  t1  u   u   u   s2  x2  y2  ...
  ```

  Note that the *stat[]* vector returned by routines in this category is of length *ncoord*, as are the intermediate *phi[]* and *theta[]* vectors returned by wcsp2s() and wcss2p().
  Note also that the function prototypes for routines in this category have to declare these two-dimensional arrays as one-dimensional vectors in order to avoid warnings from the C compiler about declaration of "incomplete types". This was considered preferable to declaring them as simple pointers-to-double which gives no indication that storage is associated with them.

- Other steps in the WCS algorithm chain typically operate only on a part of the coordinate vector. For example, a spectral transformation operates on only one element of an intermediate world coordinate that may also contain celestial coordinate elements. In the above example, spcx2s() might operate only on the *s* (spectral) coordinate elements.
  Routines like spcx2s() and celx2s() that implement these steps accept and return one-dimensional vectors in which the coordinate element of interest is specified via a starting address, a length, and a stride. To continue the previous example, the starting address for the spectral elements is *s1*, the length is 5, and the stride is 7.

8.1 Vector lengths

Routines such as spcx2s() and celx2s() accept and return either one coordinate vector, or a pair of coordinate vectors (one-dimensional C arrays). As explained above, the coordinate elements of interest are usually embedded in a two-dimensional array and must be selected by specifying a starting point, length and stride through the array. For routines such as spcx2s() that operate on a single element of each coordinate vector these parameters have a straightforward interpretation.

However, for routines such as celx2s() that operate on a pair of elements in each coordinate vector, WCSLIB allows these parameters to be specified independently for each input vector, thereby providing a much more general interpretation than strictly needed to traverse an array.

This is best described by illustration. The following diagram describes the situation for cels2x(), as a specific example, with *nlng = 5*, and *nlat = 3:*

```
          lng[0]   lng[1]   lng[2]   lng[3]   lng[4]
          ------   ------   ------   ------   ------
lat[0]  |  x,y[0]   x,y[1]   x,y[2]   x,y[3]   x,y[4]
lat[1]  |  x,y[5]   x,y[6]   x,y[7]   x,y[8]   x,y[9]
lat[2]  |  x,y[10]  x,y[11]  x,y[12]  x,y[13]  x,y[14]
```

In this case, while only 5 longitude elements and 3 latitude elements are specified, the world-to-pixel routine would calculate $nlng * nlat = 15$ *(x,y)* coordinate pairs. It is the responsibility of the caller to ensure that sufficient space has been allocated in **all** of the output arrays, in this case *phi[]*, *theta[]*, *x[]*, *y[]* and *stat[]*.

Vector computation will often be required where neither *lng* nor *lat* is constant. This is accomplished by setting *nlat = 0* which is interpreted to mean *nlat = nlng* but only the matrix diagonal is to be computed. Thus, for *nlng = 3* and *nlat = 0* only three *(x,y)* coordinate pairs are computed:

```
          lng[0]   lng[1]   lng[2]
          ------   ------   ------
 lat[0]  |  x,y[0]
 lat[1]  |           x,y[1]
 lat[2]  |                    x,y[2]
```

Note how this differs from *nlng = 3, nlat = 1*:

```
          lng[0]   lng[1]   lng[2]
          ------   ------   ------
 lat[0]  |  x,y[0]   x,y[1]   x,y[2]
```

The situation for celx2s() is similar; the *x*-coordinate (like *lng*) varies fastest.

Similar comments can be made for all routines that accept arguments specifying vector length(s) and stride(s). (tabx2s() and tabs2x() do not fall into this category because the −TAB algorithm is fully *N*-dimensional so there is no way to know in advance how many coordinate elements may be involved.)

The reason that WCSLIB allows this generality is related to the aforementioned opportunities that vector computations may provide for caching intermediate calculations and the significant efficiencies that can result. The high-level routines, wcsp2s() and wcss2p(), look for opportunities to collapse a set of coordinate transformations where one of the coordinate elements is invariant, and the low-level routines take advantage of such to cache intermediate calculations.

8.2   Vector strides

As explained above, the vector stride arguments allow the caller to specify that successive elements of a vector are not contiguous in memory. This applies equally to vectors given to, or returned from a function.

As a further example consider the following two arrangements in memory of the elements of four *(x,y)* coordinate pairs together with an *s* coordinate element (e.g. spectral):

- *x1 x2 x3 x4 y1 y2 y3 y4 s1 s2 s3 s4*
  the address of *x[]* is *x1*, its stride is 1, and length 4,
  the address of *y[]* is *y1*, its stride is 1, and length 4,
  the address of *s[]* is *s1*, its stride is 1, and length 4.

- *x1 y1 s1 x2 y2 s2 x3 y3 s3 x4 y4 s4*
  the address of *x[]* is *x1*, its stride is 3, and length 4,
  the address of *y[]* is *y1*, its stride is 3, and length 4,
  the address of *s[]* is *s1*, its stride is 3, and length 4.

For routines such as cels2x(), each of the pair of input vectors is assumed to have the same stride. Each of the output vectors also has the same stride, though it may differ from the input stride. For example, for cels2x() the input *lng[]* and *lat[]* vectors each have vector stride *sll*, while the *x[]* and *y[]* output vectors have stride *sxy*. However, the intermediate *phi[]* and *theta[]* arrays each have unit stride, as does the *stat[]* vector.

If the vector length is 1 then the stride is irrelevant and so ignored. It may be set to 0.

9 Thread-safety

Thanks to feedback and patches provided by Rodrigo Tobar Carrizo, as of release 5.18, WCSLIB is now completely thread-safe, with only a couple of minor provisos.

In particular, a number of new routines were introduced to preclude altering the global variables NPVMAX, NPS-MAX, and NDPMAX, which determine how much memory to allocate for storing PVi_ma, PSi_ma, DPja, and DQia keyvalues: wcsinit(), lininit(), lindist(), and disinit(). Specifically, these new routines are now used by various WC-SLIB routines, such as the header parsers, which previously temporarily altered the global variables, thus posing a thread hazard.

In addition, the Flex scanners were made reentrant and consequently should now be thread-safe. This was achieved by rewriting them as thin wrappers (with the same API) over scanners that were modified (with changed API), as required to use Flex's "reentrant" option.

For complete thread-safety, please observe the following provisos:

- The low-level routines wcsnpv(), wcsnps(), and disndp() are not thread-safe, but they are not used within WCSLIB itself other than to get (not set) the values of the global variables NPVMAX, NPSMAX, and NDPMAX.

  wcsinit() and disinit() only do so to get default values if the relevant parameters are not provided as function arguments. Note that wcsini() invokes wcsinit() with defaults which cause this behavior, as does disini() invoking disinit().

  The preset values of NPVMAX(=64), NPSMAX(=8), and NDPMAX(=256) are large enough to cover most practical cases. However, it may be desirable to tailor them to avoid allocating memory that remains unused. If so, and thread-safety is an issue, then use wcsinit() and disinit() instead with the relevant values specified. This is what WCSLIB routines, such as the header parsers wcspih() and wcsbth(), do to avoid wasting memory.

- wcserr_enable() sets a static variable and so is not thread-safe. However, the error reporting facility is not intended to be used dynamically. If detailed error messages are required, enable wcserr when execution starts and don't change it.

Note that diagnostic routines that print the contents of the various structs, namely celprt(), disprt(), linprt(), prjprt(), spcprt(), tabprt(), wcsprt(), and wcsperr() use printf() which is thread-safe by the POSIX requirement on `stdio`. However, this is only at the function level. Where multiple threads invoke these routines simultaneously their output is likely to be interleaved.

10 Limits

While the FITS WCS standard imposes a limit of 99 on the number of image coordinate axes, WCSLIB has a limit of 32 on the number it can handle – enforced by wcsset(), though allowed by wcsinit(). This arises in wcsp2s() and wcss2p() from the use of the *stat[]* vector as a bit mask to indicate invalid pixel or world coordinate elements.

In the unlikely event that it ever becomes necessary to handle more than 32 axes, it would be a simple matter to modify the *stat[]* bit mask so that bit 32 applies to all axes beyond 31. However, it was not considered worth introducing the various tests required just for the sake of pandering to unrealistic possibilities.

In addition, wcssub() has a hard-coded limit of 32 coordinate elements (matching the *stat[]* bit mask), and likewise for tabs2p() (via a static helper function, tabvox()). While it would be a simple matter to generalise this by allocating memory from the heap, since tabvox() calls itself recursively and needs to be as fast as possible, again it was not considered worth pandering to unrealistic possibilities.

11   Example code, testing and verification

WCSLIB has an extensive test suite that also provides programming templates as well as demonstrations. Test programs, with names that indicate the main WCSLIB routine under test, reside in ./{C,Fortran}/test and each contains a brief description of its purpose.

The high- and middle-level test programs are more instructive for applications programming, while the low-level tests are important for verifying the integrity of the mathematical routines.

- High level:
  twcstab provides an example of high-level applications programming using WCSLIB and CFITSIO. It constructs an input FITS test file, specifically for testing TAB coordinates, partly using `wcstab.keyrec`, and then extracts the coordinate description from it following the steps outlined in wcshdr.h.

  tpih1 and *tpih2* verify wcspih(). The first prints the contents of the structs returned by wcspih() using wcsprt() and the second uses *cpgsbox()* to draw coordinate graticules. Input for these comes from a FITS WCS test header implemented as a list of keyrecords, `wcs.keyrec`, one keyrecord per line, together with a program, *tofits*, that compiles these into a valid FITS file.

  tbth1 tests wcsbth() by reading a test header and printing the resulting wcsprm structs. In the process it also tests wcsfix().

  tfitshdr also uses `wcs.keyrec` to test the generic FITS header parsing routine.

  twcsfix sets up a wcsprm struct containing various non-standard constructs and then invokes wcsfix() to translate them all to standard usage.

  twcslint tests the syntax checker for FITS WCS keyrecords (wcsware -l) on a specially constructed header riddled with invalid entries.

  tdis3 uses wcsware to test the handling of different types of distortion functions encoded in a set of test FITS headers.

- Middle level:
  twcs tests closure of wcss2p() and wcsp2s() for a number of selected projections. *twcsmix* verifies wcsmix() on the $1°$ grid of celestial longitude and latitude for a number of selected projections. It plots a test grid for each projection and indicates the location of successful and failed solutions. *tdis2* and *twcssub* test the extraction of a coordinate description for a subimage from a wcsprm struct by wcssub().

  tunits tests wcsutrne(), wcsunitse() and wcsulexe(), the units specification translator, converter and parser, either interactively or using a list of units specifications contained in units_test.

  twcscompare tests particular aspects of the comparison routine, wcscompare().

- Low level:
  tdis1, *tlin*, *tlog*, *tprj1*, *tspc*, *tsph*, *tspx*, and ttab1 test "closure" of the respective routines. Closure tests apply the forward and reverse transformations in sequence and compare the result with the original value. Ideally, the result should agree exactly, but because of floating point rounding errors there is usually a small discrepancy so it is only required to agree within a "closure tolerance".

  tprj1 tests for closure separately for longitude and latitude except at the poles where it only tests for closure in latitude. Note that closure in longitude does not deal with angular displacements on the sky. This is appropriate for many projections such as the cylindricals where circumpolar parallels are projected at the same length as the equator. On the other hand, *tsph* does test for closure in angular displacement.

  The tolerance for reporting closure discrepancies is set at 10^{-10} degree for most projections; this is slightly less than 3 microarcsec. The worst case closure figure is reported for each projection and this is

usually better than the reporting tolerance by several orders of magnitude. *tprj1* and *tsph* test closure at all points on the $1°$ grid of native longitude and latitude and to within $5°$ of any latitude of divergence for those projections that cannot represent the full sphere. Closure is also tested at a sequence of points close to the reference point (*tprj1*) or pole (*tsph*).

Closure has been verified at all test points for SUN workstations. However, non-closure may be observed for other machines near native latitude $-90°$ for the zenithal, cylindrical and conic equal area projections (**ZEA**, **CEA** and **COE**), and near divergent latitudes of projections such as the azimuthal perspective and stereographic projections (**AZP** and **STG**). Rounding errors may also carry points between faces of the quad-cube projections (**CSC**, **QSC**, and **TSC**). Although such excursions may produce long lists of non-closure points, this is not necessarily indicative of a fundamental problem.

Note that the inverse of the COBE quad-qube projection (**CSC**) is a polynomial approximation and its closure tolerance is intrinsically poor.

Although tests for closure help to verify the internal consistency of the routines they do not verify them in an absolute sense. This is partly addressed by *tcel1*, *tcel2*, *tprj2*, *ttab2* and *ttab3* which plot graticules for visual inspection of scaling, orientation, and other macroscopic characteristics of the projections.

There are also a number of other special-purpose test programs that are not automatically exercised by the test suite.

12 WCSLIB Fortran wrappers

The Fortran subdirectory contains wrappers, written in C, that allow Fortran programs to use WCSLIB. The wrappers have no associated C header files, nor C function prototypes, as they are only meant to be called by Fortran code. Hence the C code must be consulted directly to determine the argument lists. This resides in files with names of the form `*_f`.c. However, there are associated Fortran `INCLUDE` files that declare function return types and various parameter definitions. There are also `BLOCK DATA` modules, in files with names of the form `*_data`.f, used solely to initialise error message strings.

A prerequisite for using the wrappers is an understanding of the usage of the associated C routines, in particular the data structures they are based on. The principle difficulty in creating the wrappers was the need to manage these C structs from within Fortran, particularly as they contain pointers to allocated memory, pointers to C functions, and other structs that themselves contain similar entities.

To this end, routines have been provided to set and retrieve values of the various structs, for example `WCSPUT` and `WCSGET` for the wcsprm struct, and `CELPUT` and `CELGET` for the celprm struct. These must be used in conjunction with wrappers on the routines provided to manage the structs in C, for example `WCSINIT`, `WCSSUB`, `WCSCOPY`, `WCSFREE`, and `WCSPRT` which wrap wcsinit(), wcssub(), wcscopy(), wcsfree(), and wcsprt().

Compilers (e.g. gfortran) may warn of inconsistent usage of the third argument in the various `*PUT` and `*GET` routines, and as of gfortran 10, these warnings have been promoted to errors. Thus, type-specific variants are provided for each of the `*PUT` routines, `*PTI`, `*PTD`, and `*PTC` for *int*, *double*, or *char*[], and likewise `*GTI`, `*GTD`, and `*GTC` for the `*GET` routines. While, for brevity, we will here continue to refer to the `*PUT` and `*GET` routines, as compilers are generally becoming stricter, use of the type-specific variants is recommended.

The various `*PUT` and `*GET` routines are based on codes defined in Fortran include files (`*`.inc). If your Fortran compiler does not support the `INCLUDE` statement then you will need to include these manually in your code as necessary. Codes are defined as parameters with names like `WCS_CRPIX` which refers to wcsprm::crpix (if your Fortran compiler does not support long symbolic names then you will need to rename these).

The include files also contain parameters, such as `WCSLEN`, that define the length of an `INTEGER` array that must be declared to hold the struct. This length may differ for different platforms depending on how the C compiler aligns data within the structs. A test program for the C library, *twcs*, prints the size of the struct in *sizeof(int)* units and the values in the Fortran include files must equal or exceed these. On some platforms, such as Suns, it is

important that the start of the INTEGER array be **_aligned on a DOUBLE PRECISION boundary_**, otherwise a mysterious BUS error may result. This may be achieved via an EQUIVALENCE with a DOUBLE PRECISION variable, or by sequencing variables in a COMMON block so that the INTEGER array follows immediately after a DOUBLE PRECISION variable.

The *PUT routines set only one element of an array at a time; the final one or two integer arguments of these routines specify 1-relative array indices (N.B. not 0-relative as in C). The one exception is the prjprm::pv array.

The *PUT routines also reset the *flag* element to signal that the struct needs to be reinitialized. Therefore, if you wanted to set wcsprm::flag itself to -1 prior to the first call to WCSINIT, for example, then that WCSPUT must be the last one before the call.

The *GET routines retrieve whole arrays at a time and expect array arguments of the appropriate length where necessary. Note that they do not initialize the structs, i.e. via wcsset(), prjset(), or whatever.

A basic coding fragment is

```
        INTEGER   LNGIDX, STATUS
        CHARACTER CTYPE1*72

        INCLUDE 'wcs.inc'

*       WCSLEN is defined as a parameter in wcs.inc.
        INTEGER   WCS(WCSLEN)
        DOUBLE PRECISION DUMMY
        EQUIVALENCE (WCS, DUMMY)

*       Allocate memory and set default values for 2 axes.
        STATUS = WCSPTI (WCS, WCS_FLAG, -1, 0, 0)
        STATUS = WCSINI (2, WCS)

*       Set CRPIX1, and CRPIX2; WCS_CRPIX is defined in wcs.inc.
        STATUS = WCSPTD (WCS, WCS_CRPIX, 512D0, 1, 0)
        STATUS = WCSPTD (WCS, WCS_CRPIX, 512D0, 2, 0)

*       Set PC1_2 to 5.0 (I = 1, J = 2).
        STATUS = WCSPTD (WCS, WCS_PC, 5D0, 1, 2)

*       Set CTYPE1 to 'RA---SIN'; N.B. must be given as CHARACTER*72.
        CTYPE1 = 'RA---SIN'
        STATUS = WCSPTC (WCS, WCS_CTYPE, CTYPE1, 1, 0)

*       Use an alternate method to set CTYPE2.
        STATUS = WCSPTC (WCS, WCS_CTYPE, 'DEC--SIN'//CHAR(0), 2, 0)

*       Set PV1_3 to -1.0 (I = 1, M = 3).
        STATUS = WCSPTD (WCS, WCS_PV, -1D0, 1, 3)

        etc.

*       Initialize.
        STATUS = WCSSET (WCS)
        IF (STATUS.NE.0) THEN
          CALL FLUSH (6)
          STATUS = WCSPERR (WCS, 'EXAMPLE: '//CHAR(0))
        ENDIF

*       Find the "longitude" axis.
        STATUS = WCSGTI (WCS, WCS_LNG, LNGIDX)

*       Free memory.
        STATUS = WCSFREE (WCS)
```

Refer to the various Fortran test programs for further programming examples. In particular, *twcs* and *twcsmix* show how to retrieve elements of the celprm and prjprm structs contained within the wcsprm struct.

Treatment of `CHARACTER` arguments in wrappers such as `SPCTYPE`, `SPECX`, and `WCSSPTR`, depends on whether they are given or returned. Where a `CHARACTER` variable is returned, its length must match the declared length in the definition of the C wrapper. The terminating null character in the C string, and all following it up to the declared length, are replaced with blanks. If the Fortran `CHARACTER` variable were shorter than the declared length, an out-of-bounds memory access error would result. If longer, the excess, uninitialized, characters could contain garbage.

If the `CHARACTER` argument is given, a null-terminated `CHARACTER` variable may be provided as input, e.g. constructed using the Fortran `CHAR(0)` intrinsic as in the example code above. The wrapper makes a character-by-character copy, searching for a NULL character in the process. If it finds one, the copy terminates early, resulting in a valid C string. In this case any trailing blanks before the `NULL` character are preserved if it makes sense to do so, such as in setting a prefix for use by the `*PERR` wrappers, such as `WCSPERR` in the example above. If a NULL is not found, then the `CHARACTER` argument must be at least as long as the declared length, and any trailing blanks are stripped off. Should a `CHARACTER` argument exceed the declared length, the excess characters are ignored.

There is one exception to the above caution regarding `CHARACTER` arguments. The `WCSLIB_VERSION` wrapper is unusual in that it provides for the length of its `CHARACTER` argument to be specified, and only so many characters as fit within that length are returned.

Note that the data type of the third argument to the `*PUT` (or `*PTI`, `*PTD`, or `*PTC`) and `*GET` (or `*GTI`, `*GTD`, or `*GTC`) routines differs depending on the data type of the corresponding C struct member, be it *int*, *double*, or *char*[]. It is essential that the Fortran data type match that of the C struct for *int* and *double* types, and be a `CHARACTER` variable of the correct length for *char*[] types, or else be null-terminated, as in the coding example above. As a further example, in the two equivalent calls

```
      STATUS = PRJGET (PRJ, PRJ_NAME, NAME)
      STATUS = PRJGTC (PRJ, PRJ_NAME, NAME)
```

which return a character string, `NAME` must be a `CHARACTER` variable of length 40, as declared in the prjprm struct, no less and no more, the comments above pertaining to wrappers that contain `CHARACTER` arguments also applying here. However, a few exceptions have been made to simplify coding. The relevant `*PUT` (or `*PTC`) wrappers allow unterminated `CHARACTER` variables of less than the declared length for the following: `prjprm::code` (3 characters), `spcprm::type` (4 characters), `spcprm::code` (3 characters), and `fitskeyid::name` (8 characters). It doesn't hurt to specify longer `CHARACTER` variables, but the trailing characters will be ignored. Notwithstanding this simplification, the length of the corresponding variables in the `*GET` (or `*GTC`) wrappers must match the length declared in the struct.

When calling wrappers for C functions that print to *stdout*, such as `WCSPRT`, and `WCSPERR`, or that may print to *stderr*, such as `WCSPIH`, `WCSBTH`, `WCSULEXE`, or `WCSUTRNE`, it may be necessary to flush the Fortran I/O buffers beforehand so that the output appears in the correct order. The wrappers for these functions do call `fflush(↩ NULL)`, but depending on the particular system, this may not succeed in flushing the Fortran I/O buffers. Most Fortran compilers provide the non-standard intrinsic `FLUSH()`, which is called with unit number 6 to flush *stdout* (as in the example above), and unit 0 for *stderr*.

A basic assumption made by the wrappers is that an `INTEGER` variable is no less than half the size of a `DOUBLE PRECISION`.

13 PGSBOX

`PGSBOX`, which is provided as a separate part of WCSLIB, is a [PGPLOT](#) routine (PGPLOT being a Fortran graphics library) that draws and labels curvilinear coordinate grids. Example `PGSBOX` grids can be seen at `http↩ ://www.atnf.csiro.au/people/Mark.Calabretta/WCS/PGSBOX/index.html`.

The prologue to pgsbox.f contains usage instructions. pgtest.f and cpgtest.c serve as test and demonstration programs in Fortran and C and also as well- documented examples of usage.

PGSBOX requires a separate routine, `EXTERNAL NLFUNC`, to define the coordinate transformation. Fortran subroutine PGCRFN (pgcrfn.f) is provided to define separable pairs of non-linear coordinate systems. Linear, logarithmic and power-law axis types are currently defined; further types may be added as required. A C function, *pgwcsl↩ _()*, with Fortran-like interface defines an `NLFUNC` that interfaces to WCSLIB 4.x for PGSBOX to draw celestial coordinate grids.

[PGPLOT](#) is implemented as a Fortran library with a set of C wrapper routines that are generated by a software tool. However, PGSBOX has a more complicated interface than any of the standard PGPLOT routines, especially in having an `EXTERNAL` function in its argument list. Consequently, PGSBOX is implemented in Fortran but with a hand-coded C wrapper, *cpgsbox()*.

As an example, in this suite the C test/demo program, *cpgtest*, calls the C wrapper, *cpgsbox()*, passing it a pointer to *pgwcsl_()*. In turn, *cpgsbox()* calls PGSBOX, which invokes *pgwcsl_()* as an `EXTERNAL` subroutine. In this sequence, a complicated C struct defined by *cpgtest* is passed through PGSBOX to *pgwcsl_()* as an `INTEGER` array.

While there are no formal standards for calling Fortran from C, there are some fairly well established conventions. Nevertheless, it's possible that you may need to modify the code if you use a combination of Fortran and C compilers with linkage conventions that differ from that of the GNU compilers, gcc and g77.

14 WCSLIB version numbers

The full WCSLIB/PGSBOX version number is composed of three integers in fields separated by periods:

- **Major**: the first number changes only when the ABI changes, a rare occurence (and the API never changes). Typically, the ABI changes when the contents of a struct change. For example, the contents of the *linprm* struct changed between 4.25.1 and 5.0.

  In practical terms, this number becomes the major version number of the WCSLIB sharable library, **libwcs.↩ so.**<*major*>. To avoid possible segmentation faults or bus errors that may arise from the altered ABI, the dynamic (runtime) linker will not allow an application linked to a sharable library with a particular major version number to run with that of a different major version number.

  Application code must be recompiled and relinked to use a newer version of the WCSLIB sharable library with a new major version number.

- **Minor**: the second number changes when existing code is changed, whether due to added functionality or bug fixes. This becomes the minor version number of the WCSLIB sharable library, **libwcs.↩ so.**<*major*>.<*minor*>.

  Because the ABI remains unchanged, older applications can use the new sharable library without needing to be recompiled, thus obtaining the benefit of bug fixes, speed enhancements, etc.

  Application code written subsequently to use the added functionality would, of course, need to be recompiled.

- **Patch**: the third number, which is often omitted, indicates a change to the build procedures, documentation, or test suite. It may also indicate changes to the utility applications (*wcsware*, *HPXcvt*, etc.), including the addition of new ones.

  However, the library itself, including the definitions in the header files, remains unaltered, so there is no point in recompiling applications.

The following describes what happens (or should happen) when WCSLIB's installation procedures are used on a typical Linux system using the GNU gcc compiler and GNU linker.

The sharable library should be installed as libwcs.so.<*major*>.<*minor*>, say libwcs.so.5.4 for concreteness, and a number of symbolic links created as follows:

```
libwcs.so      -> libwcs.so.5
libwcs.so.5    -> libwcs.so.5.4
libwcs.so.5.4
```

When an application is linked using '-lwcs', the linker finds libwcs.so and the symlinks lead it to libwcs.so.5.4. However, that library's SONAME is actually 'libwcs.so.5', by virtue of linker options used when the sharable library was created, as can be seen by running

```
readelf -d libwcs.so.5.4
```

Thus, when an application that was compiled and linked to libwcs.so.5.4 begins execution, the dynamic linker, ld.so, will attempt to bind it to libwcs.so.5, as can be seen by running

```
ldd <application>
```

Later, when WCSLIB 5.5 is installed, the library symbolic links will become

```
libwcs.so      -> libwcs.so.5
libwcs.so.5    -> libwcs.so.5.5
libwcs.so.5.4
libwcs.so.5.5
```

Thus, even without being recompiled, existing applications will link automatically to libwcs.so.5.5 at runtime. In fact, libwcs.so.5.4 would no longer be used and could be deleted.

If WCSLIB 6.0 were to be installed at some later time, then the libwcs.so.6 libraries would be used for new compilations. However, the libwcs.so.5 libraries must be left in place for existing executables that still require them:

```
libwcs.so      -> libwcs.so.6
libwcs.so.6    -> libwcs.so.6.0
libwcs.so.6.0
libwcs.so.5    -> libwcs.so.5.5
libwcs.so.5.5
```

15 Deprecated List

Global celini_errmsg

    Added for backwards compatibility, use cel_errmsg directly now instead.

Global celprt_errmsg

    Added for backwards compatibility, use cel_errmsg directly now instead.

Global cels2x_errmsg

    Added for backwards compatibility, use cel_errmsg directly now instead.

Global celset_errmsg

    Added for backwards compatibility, use cel_errmsg directly now instead.

Global celx2s_errmsg

    Added for backwards compatibility, use cel_errmsg directly now instead.

Global cylfix_errmsg

    Added for backwards compatibility, use wcsfix_errmsg directly now instead.

Global FITSHDR_CARD

Added for backwards compatibility, use *FITSHDR_KEYREC* instead.

Global lincpy_errmsg

Added for backwards compatibility, use lin_errmsg directly now instead.

Global linfree_errmsg

Added for backwards compatibility, use lin_errmsg directly now instead.

Global linini_errmsg

Added for backwards compatibility, use lin_errmsg directly now instead.

Global linp2x_errmsg

Added for backwards compatibility, use lin_errmsg directly now instead.

Global linprt_errmsg

Added for backwards compatibility, use lin_errmsg directly now instead.

Global linset_errmsg

Added for backwards compatibility, use lin_errmsg directly now instead.

Global linx2p_errmsg

Added for backwards compatibility, use lin_errmsg directly now instead.

Global prjini_errmsg

Added for backwards compatibility, use prj_errmsg directly now instead.

Global prjprt_errmsg

Added for backwards compatibility, use prj_errmsg directly now instead.

Global prjs2x_errmsg

Added for backwards compatibility, use prj_errmsg directly now instead.

Global prjset_errmsg

Added for backwards compatibility, use prj_errmsg directly now instead.

Global prjx2s_errmsg

Added for backwards compatibility, use prj_errmsg directly now instead.

Global spcini_errmsg

Added for backwards compatibility, use spc_errmsg directly now instead.

Global spcprt_errmsg

Added for backwards compatibility, use spc_errmsg directly now instead.

Global spcs2x_errmsg

Added for backwards compatibility, use spc_errmsg directly now instead.

Global spcset_errmsg

Added for backwards compatibility, use spc_errmsg directly now instead.

Global spcx2s_errmsg

Added for backwards compatibility, use spc_errmsg directly now instead.

Global tabcpy_errmsg

Added for backwards compatibility, use tab_errmsg directly now instead.

Global tabfree_errmsg

Added for backwards compatibility, use tab_errmsg directly now instead.

Global tabini_errmsg

Added for backwards compatibility, use tab_errmsg directly now instead.

Global tabprt_errmsg

Added for backwards compatibility, use tab_errmsg directly now instead.

Global tabs2x_errmsg

Added for backwards compatibility, use tab_errmsg directly now instead.

Global tabset_errmsg

Added for backwards compatibility, use tab_errmsg directly now instead.

Global tabx2s_errmsg

Added for backwards compatibility, use tab_errmsg directly now instead.

Global wcscopy_errmsg

Added for backwards compatibility, use wcs_errmsg directly now instead.

Global wcsfree_errmsg

Added for backwards compatibility, use wcs_errmsg directly now instead.

Global wcsini_errmsg

Added for backwards compatibility, use wcs_errmsg directly now instead.

Global wcsmix_errmsg

Added for backwards compatibility, use wcs_errmsg directly now instead.

Global wcsp2s_errmsg

Added for backwards compatibility, use wcs_errmsg directly now instead.

Global wcsprt_errmsg

Added for backwards compatibility, use wcs_errmsg directly now instead.

Global wcss2p_errmsg

Added for backwards compatibility, use wcs_errmsg directly now instead.

Global wcsset_errmsg

Added for backwards compatibility, use wcs_errmsg directly now instead.

Global wcssub_errmsg

Added for backwards compatibility, use wcs_errmsg directly now instead.

16 Data Structure Index

16.1 Data Structures

Here are the data structures with brief descriptions:

17 File Index

17.1 File List

Here is a list of all files with brief descriptions:

18 Data Structure Documentation

18.1 auxprm Struct Reference

Additional auxiliary parameters.

```
#include <wcs.h>
```

Data Fields

- double rsun_ref
- double dsun_obs
- double crln_obs
- double hgln_obs
- double hglt_obs

18.1.1 Detailed Description

The **auxprm** struct holds auxiliary coordinate system information of a specialist nature. It is anticipated that this struct will expand in future to accomodate additional parameters.

All members of this struct are to be set by the user.

18.1.2 Field Documentation

18.1.2.1   rsun_ref `double auxprm::rsun_ref`

(Given, auxiliary) Reference radius of the Sun used in coordinate calculations (m).

18.1.2.2   dsun_obs `double auxprm::dsun_obs`

(Given, auxiliary) Distance between the centre of the Sun and the observer (m).

18.1.2.3   crln_obs `double auxprm::crln_obs`

(Given, auxiliary) Carrington heliographic longitude of the observer (deg).

18.1.2.4   hgln_obs `double auxprm::hgln_obs`

(Given, auxiliary) Stonyhurst heliographic longitude of the observer (deg).

18.1.2.5   hglt_obs `double auxprm::hglt_obs`

(Given, auxiliary) Heliographic latitude (Carrington or Stonyhurst) of the observer (deg).

18.2   celprm Struct Reference

Celestial transformation parameters.

`#include <cel.h>`

Data Fields

- int flag
- int offset
- double phi0
- double theta0
- double ref [4]
- struct prjprm prj
- double euler [5]
- int latpreq
- int isolat
- struct wcserr ∗ err
- void ∗ padding

18.2.1   Detailed Description

The **celprm** struct contains information required to transform celestial coordinates. It consists of certain members that must be set by the user (*given*) and others that are set by the WCSLIB routines (*returned*). Some of the latter are supplied for informational purposes and others are for internal use only.

Returned **celprm** struct members must not be modified by the user.

18.2.2 Field Documentation

18.2.2.1 flag `int celprm::flag`

(Given and returned) This flag must be set to zero whenever any of the following **celprm** struct members are set or changed:

- celprm::offset,

- celprm::phi0,

- celprm::theta0,

- celprm::ref[4],

- celprm::prj:

    - prjprm::code,

    - prjprm::r0,

    - prjprm::pv[],

    - prjprm::phi0,

    - prjprm::theta0.

This signals the initialization routine, celset(), to recompute the returned members of the **celprm** struct. celset() will reset flag to indicate that this has been done.

18.2.2.2 offset `int celprm::offset`

(*Given*) If true (non-zero), an offset will be applied to (x, y) to force (x, y) = (0,0) at the fiducial point, (ϕ_0, θ_0). Default is 0 (false).

18.2.2.3 phi0 `double celprm::phi0`

(*Given*) The native longitude, ϕ_0 [deg], and ...

18.2.2.4 theta0 `double celprm::theta0`

(*Given*) ... the native latitude, θ_0 [deg], of the fiducial point, i.e. the point whose celestial coordinates are given in celprm::ref[1:2]. If undefined (set to a magic value by prjini()) the initialization routine, celset(), will set this to a projection-specific default.

18.2.2.5 ref `double celprm::ref`

(*Given*) The first pair of values should be set to the celestial longitude and latitude of the fiducial point [deg] - typically right ascension and declination. These are given by the **CRVAL**i a keywords in FITS.

(Given and returned) The second pair of values are the native longitude, ϕ_p [deg], and latitude, θ_p [deg], of the celestial pole (the latter is the same as the celestial latitude of the native pole, δ_p) and these are given by the FITS keywords **LONPOLE**a and **LATPOLE**a (or by **PV**i_**2**a and **PV**i_**3**a attached to the longitude axis which take precedence if defined).

LONPOLEa defaults to ϕ_0 (see above) if the celestial latitude of the fiducial point of the projection is greater than or equal to the native latitude, otherwise ϕ_0 + 180 [deg]. (This is the condition for the celestial latitude to increase in the same direction as the native latitude at the fiducial point.) ref[2] may be set to **UNDEFINED** (from wcsmath.h) or 999.0 to indicate that the correct default should be substituted.

θ_p, the native latitude of the celestial pole (or equally the celestial latitude of the native pole, δ_p) is often determined uniquely by **CRVAL**ia and **LONPOLE**a in which case **LATPOLE**a is ignored. However, in some circumstances there are two valid solutions for θ_p and **LATPOLE**a is used to choose between them. **LATPOLE**a is set in ref[3] and the solution closest to this value is used to reset ref[3]. It is therefore legitimate, for example, to set ref[3] to +90.0 to choose the more northerly solution - the default if the **LATPOLE**a keyword is omitted from the FITS header. For the special case where the fiducial point of the projection is at native latitude zero, its celestial latitude is zero, and **LONPOLE**a = $\pm$ 90.0 then the celestial latitude of the native pole is not determined by the first three reference values and **LATPOLE**a specifies it completely.

The returned value, celprm::latpreq, specifies how **LATPOLE**a was actually used.

18.2.2.6 prj `struct prjprm celprm::prj`

(Given and returned) Projection parameters described in the prologue to prj.h.

18.2.2.7 euler `double celprm::euler`

(*Returned*) Euler angles and associated intermediaries derived from the coordinate reference values. The first three values are the Z-, X-, and Z'-Euler angles [deg], and the remaining two are the cosine and sine of the X-Euler angle.

18.2.2.8 latpreq `int celprm::latpreq`

(*Returned*) For informational purposes, this indicates how the **LATPOLE**a keyword was used

- 0: Not required, θ_p (== δ_p) was determined uniquely by the **CRVAL**ia and **LONPOLE**a keywords.

- 1: Required to select between two valid solutions of θ_p.

- 2: θ_p was specified solely by **LATPOLE**a.

18.2.2.9 isolat `int celprm::isolat`

(*Returned*) True if the spherical rotation preserves the magnitude of the latitude, which occurs iff the axes of the native and celestial coordinates are coincident. It signals an opportunity to cache intermediate calculations common to all elements in a vector computation.

18.2.2.10 err `struct wcserr * celprm::err`

(*Returned*) If enabled, when an error status is returned, this struct contains detailed information about the error, see wcserr_enable().

18.2.2.11 padding `void * celprm::padding`

(An unused variable inserted for alignment purposes only.)

Global variable: const char ∗cel_errmsg[] - Status return messages Status messages to match the status value returned from each function.

18.3 disprm Struct Reference

Distortion parameters.

`#include <dis.h>`

Data Fields

- int flag
- int naxis
- char(∗ dtype)[72]
- int ndp
- int ndpmax
- struct dpkey ∗ dp
- double ∗ maxdis
- double totdis
- int ∗ docorr
- int ∗ Nhat
- int ∗∗ axmap
- double ∗∗ offset
- double ∗∗ scale
- int ∗∗ iparm
- double ∗∗ dparm
- int i_naxis
- int ndis
- struct wcserr ∗ err
- int(∗∗ disp2x)(DISP2X_ARGS)
- int(∗∗ disx2p)(DISX2P_ARGS)
- double ∗ tmpmem
- int m_flag
- int m_naxis
- char(∗ m_dtype)[72]
- struct dpkey ∗ m_dp
- double ∗ m_maxdis

18.3.1 Detailed Description

The **disprm** struct contains all of the information required to apply a set of distortion functions. It consists of certain members that must be set by the user (*given*) and others that are set by the WCSLIB routines (*returned*). While the addresses of the arrays themselves may be set by disinit() if it (optionally) allocates memory, their contents must be set by the user.

18.3.2   Field Documentation

18.3.2.1   flag   `int disprm::flag`

(Given and returned) This flag must be set to zero whenever any of the following members of the **disprm** struct are set or modified:

- disprm::naxis,

- disprm::dtype,

- disprm::ndp,

- disprm::dp.

This signals the initialization routine, disset(), to recompute the returned members of the **disprm** struct. disset() will reset flag to indicate that this has been done.

PLEASE NOTE: flag must be set to -1 when disinit() is called for the first time for a particular **disprm** struct in order to initialize memory management. It must ONLY be used on the first initialization otherwise memory leaks may result.

18.3.2.2   naxis   `int disprm::naxis`

(Given or returned) Number of pixel and world coordinate elements.

If disinit() is used to initialize the **disprm** struct (as would normally be the case) then it will set naxis from the value passed to it as a function argument. The user should not subsequently modify it.

18.3.2.3   dtype   `disprm::dtype`

(*Given*) Pointer to the first element of an array of char[72] containing the name of the distortion function for each axis.

18.3.2.4   ndp   `int disprm::ndp`

(*Given*) The number of entries in the disprm::dp[] array.

18.3.2.5   ndpmax   `int disprm::ndpmax`

(*Given*) The length of the disprm::dp[] array.

ndpmax will be set by disinit() if it allocates memory for disprm::dp[], otherwise it must be set by the user. See also disndp().

18.3.2.6 dp `struct dpkey disprm::dp`

(*Given*) Address of the first element of an array of length ndpmax of dpkey structs.

As a FITS header parser encounters each **DP**`ja` or **DQ**`ia` keyword it should load it into a dpkey struct in the array and increment ndp. However, note that a single **disprm** struct must hold only **DP**`ja` or **DQ**`ia` keyvalues, not both. disset() interprets them as required by the particular distortion function.

18.3.2.7 maxdis `double * disprm::maxdis`

(*Given*) Pointer to the first element of an array of double specifying the maximum absolute value of the distortion for each axis computed over the whole image.

It is not necessary to reset the **disprm** struct (via disset()) when disprm::maxdis is changed.

18.3.2.8 totdis `double disprm::totdis`

(*Given*) The maximum absolute value of the combination of all distortion functions specified as an offset in pixel coordinates computed over the whole image.

It is not necessary to reset the **disprm** struct (via disset()) when disprm::totdis is changed.

18.3.2.9 docorr `int * disprm::docorr`

(*Returned*) Pointer to the first element of an array of int containing flags that indicate the mode of correction for each axis.

If docorr is zero, the distortion function returns the corrected coordinates directly. Any other value indicates that the distortion function computes a correction to be added to pixel coordinates (prior distortion) or intermediate pixel coordinates (sequent distortion).

18.3.2.10 Nhat `int * disprm::Nhat`

(*Returned*) Pointer to the first element of an array of int containing the number of coordinate axes that form the independent variables of the distortion function for each axis.

18.3.2.11 axmap `int ** disprm::axmap`

(*Returned*) Pointer to the first element of an array of int∗ containing pointers to the first elements of the axis mapping arrays for each axis.

An axis mapping associates the independent variables of a distortion function with the 0-relative image axis number. For example, consider an image with a spectrum on the first axis (axis 0), followed by RA (axis 1), Dec (axis2), and time (axis 3) axes. For a distortion in (RA,Dec) and no distortion on the spectral or time axes, the axis mapping arrays, axmap[j][], would be
```
j=0: [-1, -1, -1, -1]   ...no  distortion on spectral axis,
  1: [ 1,  2, -1, -1]   ...RA  distortion depends on RA and Dec,
  2: [ 2,  1, -1, -1]   ...Dec distortion depends on Dec and RA,
  3: [-1, -1, -1, -1]   ...no  distortion on time axis,
```

where -1 indicates that there is no corresponding independent variable.

18.3.2.12 offset `double ** disprm::offset`

(*Returned*) Pointer to the first element of an array of double∗ containing pointers to the first elements of arrays of offsets used to renormalize the independent variables of the distortion function for each axis.

The offsets are subtracted from the independent variables before scaling.

18.3.2.13 scale `double ** disprm::scale`

(*Returned*) Pointer to the first element of an array of double∗ containing pointers to the first elements of arrays of scales used to renormalize the independent variables of the distortion function for each axis.

The scale is applied to the independent variables after the offsets are subtracted.

18.3.2.14 iparm `int ** disprm::iparm`

(*Returned*) Pointer to the first element of an array of int∗ containing pointers to the first elements of the arrays of integer distortion parameters for each axis.

18.3.2.15 dparm `double ** disprm::dparm`

(*Returned*) Pointer to the first element of an array of double∗ containing pointers to the first elements of the arrays of floating point distortion parameters for each axis.

18.3.2.16 i_naxis `int disprm::i_naxis`

(*Returned*) Dimension of the internal arrays (normally equal to naxis).

18.3.2.17 ndis `int disprm::ndis`

(*Returned*) The number of distortion functions.

18.3.2.18 err `struct wcserr * disprm::err`

(*Returned*) If enabled, when an error status is returned, this struct contains detailed information about the error, see wcserr_enable().

18.3.2.19 disp2x `int(** disprm::disp2x)(DISP2X_ARGS)`

(For internal use only.)

18.3.2.20 disx2p `int(** disprm::disx2p)(DISX2P_ARGS)`

(For internal use only.)

18.3.2.21 tmpmem `double * disprm::tmpmem`

(For internal use only.)

18.3.2.22 m_flag `int disprm::m_flag`

(For internal use only.)

18.3.2.23 m_naxis `int disprm::m_naxis`

(For internal use only.)

18.3.2.24 m_dtype `disprm::m_dtype`

(For internal use only.)

18.3.2.25 m_dp `double ** disprm::m_dp`

(For internal use only.)

18.3.2.26 m_maxdis `double * disprm::m_maxdis`

(For internal use only.)

18.4 dpkey Struct Reference

Store for **DP**`ja` and **DQ**`ia` keyvalues.

`#include <dis.h>`

Data Fields

- char field [72]
- int j
- int type
- union {
    int i
    double f
  } value

18.4.1 Detailed Description

The **dpkey** struct is used to pass the parsed contents of **DP**`ja` or **DQ**`ia` keyrecords to disset() via the disprm struct. A disprm struct must hold only **DP**`ja` or **DQ**`ia` keyvalues, not both.

All members of this struct are to be set by the user.

18.4.2 Field Documentation

18.4.2.1 field `char dpkey::field`

(*Given*) The full field name of the record, including the keyword name. Note that the colon delimiter separating the field name and the value in record-valued keyvalues is not part of the field name. For example, in the following:
`DP3A = 'AXIS.1: 2'`

the full record field name is "DP3A**.AXIS.**1", and the record's value is 2.

18.4.2.2 j `int dpkey::j`

(*Given*) Axis number (1-relative), i.e. the j in **DP**`ja` or i in **DQ**`ia`.

18.4.2.3 type `int dpkey::type`

(*Given*) The data type of the record's value

- 0: Integer (stored as an int),

- 1: Floating point (stored as a double).

18.4.2.4 i `int dpkey::i`

18.4.2.5 f `double dpkey::f`

18.4.2.6 value `union dpkey::value`

(*Given*) A union comprised of

- dpkey::i,

- dpkey::f,

the record's value.

18.5 fitskey Struct Reference

Keyword/value information.

`#include <fitshdr.h>`

Data Fields

- int keyno
- int keyid
- int status
- char keyword [12]
- int type
- int padding
- union {
    int i
    int64 k
    int l [8]
    double f
    double c [2]
    char s [72]
  } keyvalue

- int ulen
- char comment [84]

18.5.1 Detailed Description

fitshdr() returns an array of **fitskey** structs, each of which contains the result of parsing one FITS header keyrecord. All members of the **fitskey** struct are returned by fitshdr(), none are given by the user.

18.5.2 Field Documentation

18.5.2.1 keyno `int fitskey::keyno`

(*Returned*) Keyrecord number (1-relative) in the array passed as input to fitshdr(). This will be negated if the keyword matched any specified in the keyids[] index.

18.5.2.2 keyid `int fitskey::keyid`

(*Returned*) Index into the first entry in keyids[] with which the keyrecord matches, else -1.

18.5.2.3 status `int fitskey::status`

(*Returned*) Status flag bit-vector for the header keyrecord employing the following bit masks defined as preprocessor macros:

- FITSHDR_KEYWORD: Illegal keyword syntax.

- FITSHDR_KEYVALUE: Illegal keyvalue syntax.

- FITSHDR_COMMENT: Illegal keycomment syntax.

- FITSHDR_KEYREC: Illegal keyrecord, e.g. an **END** keyrecord with trailing text.

- FITSHDR_TRAILER: Keyrecord following a valid **END** keyrecord.

The header keyrecord is syntactically correct if no bits are set.

18.5.2.4 keyword `char fitskey::keyword`

(*Returned*) Keyword name, null-filled for keywords of less than eight characters (trailing blanks replaced by nulls).

Use
```
sprintf(dst, "%.8s", keyword)
```
to copy it to a character array with null-termination, or
```
sprintf(dst, "%8.8s", keyword)
```
to blank-fill to eight characters followed by null-termination.

18.5.2.5 type `int fitskey::type`

(*Returned*) Keyvalue data type:

- 0: No keyvalue (both the value and type are undefined).

- 1: Logical, represented as int.

- 2: 32-bit signed integer.

- 3: 64-bit signed integer (see below).

- 4: Very long integer (see below).

- 5: Floating point (stored as double).

- 6: Integer complex (stored as double[2]).

- 7: Floating point complex (stored as double[2]).

- 8: String.

- 8+10∗n: Continued string (described below and in fitshdr() note 2).

A negative type indicates that a syntax error was encountered when attempting to parse a keyvalue of the particular type.

Comments on particular data types:

- 64-bit signed integers lie in the range
```
(-9223372036854775808 <= int64 <  -2147483648) ||
         (+2147483647 <  int64 <= +9223372036854775807)
```
  A native 64-bit data type may be defined via preprocessor macro WCSLIB_INT64 defined in wcsconfig.h, e.g. as 'long long int'; this will be typedef'd to 'int64' here. If WCSLIB_INT64 is not set, then int64 is typedef'd to int[3] instead and fitskey::keyvalue is to be computed as
```
((keyvalue.k[2]) * 1000000000 +
  keyvalue.k[1]) * 1000000000 +
  keyvalue.k[0]
```
  and may reported via
```
if (keyvalue.k[2]) {
  printf("%d%09d%09d", keyvalue.k[2], abs(keyvalue.k[1]),
                       abs(keyvalue.k[0]));
} else {
  printf("%d%09d", keyvalue.k[1], abs(keyvalue.k[0]));
}
```
  where keyvalue.k[0] and keyvalue.k[1] range from -999999999 to +999999999.

- Very long integers, up to 70 decimal digits in length, are encoded in keyvalue.l as an array of int[8], each of which stores 9 decimal digits. fitskey::keyvalue is to be computed as
```
(((((((keyvalue.l[7]) * 1000000000 +
       keyvalue.l[6]) * 1000000000 +
       keyvalue.l[5]) * 1000000000 +
       keyvalue.l[4]) * 1000000000 +
       keyvalue.l[3]) * 1000000000 +
       keyvalue.l[2]) * 1000000000 +
       keyvalue.l[1]) * 1000000000 +
       keyvalue.l[0]
```

- Continued strings are not reconstructed, they remain split over successive **fitskey** structs in the keys[] array returned by fitshdr(). fitskey::keyvalue data type, 8 + 10n, indicates the segment number, n, in the continuation.

18.5.2.6 padding `int fitskey::padding`

(An unused variable inserted for alignment purposes only.)

18.5.2.7 i `int fitskey::i`

(*Returned*) Logical (fitskey::type == 1) and 32-bit signed integer (fitskey::type == 2) data types in the fitskey::keyvalue union.

18.5.2.8 k `int64 fitskey::k`

(*Returned*) 64-bit signed integer (fitskey::type == 3) data type in the fitskey::keyvalue union.

18.5.2.9 l `int fitskey::l`

(*Returned*) Very long integer (fitskey::type == 4) data type in the fitskey::keyvalue union.

18.5.2.10 f `double fitskey::f`

(*Returned*) Floating point (fitskey::type == 5) data type in the fitskey::keyvalue union.

18.5.2.11 c `double fitskey::c`

(*Returned*) Integer and floating point complex (fitskey::type == 6 || 7) data types in the fitskey::keyvalue union.

18.5.2.12 s `char fitskey::s`

(*Returned*) Null-terminated string (fitskey::type == 8) data type in the fitskey::keyvalue union.

18.5.2.13 keyvalue `union fitskey::keyvalue`

(*Returned*) A union comprised of

- fitskey::i,
- fitskey::k,
- fitskey::l,
- fitskey::f,
- fitskey::c,
- fitskey::s,

used by the **fitskey** struct to contain the value associated with a keyword.

18.5.2.14 ulen `int fitskey::ulen`

(*Returned*) Where a keycomment contains a units string in the standard form, e.g. [m/s], the ulen member indicates its length, inclusive of square brackets. Otherwise ulen is zero.

18.5.2.15 comment `char fitskey::comment`

(*Returned*) Keycomment, i.e. comment associated with the keyword or, for keyrecords rejected because of syntax errors, the compete keyrecord itself with null-termination.

Comments are null-terminated with trailing spaces removed. Leading spaces are also removed from keycomments (i.e. those immediately following the '**/**' character), but not from **COMMENT** or **HISTORY** keyrecords or keyrecords without a value indicator ("**=** " in columns 9-80).

18.6 fitskeyid Struct Reference

Keyword indexing.

`#include <fitshdr.h>`

Data Fields

- char name [12]
- int count
- int idx [2]

18.6.1 Detailed Description

fitshdr() uses the **fitskeyid** struct to return indexing information for specified keywords. The struct contains three members, the first of which, fitskeyid::name, must be set by the user with the remainder returned by fitshdr().

18.6.2 Field Documentation

18.6.2.1 name `char fitskeyid::name`

(*Given*) Name of the required keyword. This is to be set by the user; the '.' character may be used for wildcarding. Trailing blanks will be replaced with nulls.

18.6.2.2 count `int fitskeyid::count`

(*Returned*) The number of matches found for the keyword.

18.6.2.3 idx `int fitskeyid::idx`

(*Returned*) Indices into keys[], the array of [fitskey](#) structs returned by [fitshdr()](#). Note that these are 0-relative array indices, not keyrecord numbers.

If the keyword is found in the header the first index will be set to the array index of its first occurrence, otherwise it will be set to -1.

If multiples of the keyword are found, the second index will be set to the array index of its last occurrence, otherwise it will be set to -1.

18.7   linprm Struct Reference

Linear transformation parameters.

`#include <lin.h>`

Data Fields

- int [flag](#)
- int [naxis](#)
- double ∗ [crpix](#)
- double ∗ [pc](#)
- double ∗ [cdelt](#)
- struct [disprm](#) ∗ [dispre](#)
- struct [disprm](#) ∗ [disseq](#)
- double ∗ [piximg](#)
- double ∗ [imgpix](#)
- int [i_naxis](#)
- int [unity](#)
- int [affine](#)
- int [simple](#)
- struct [wcserr](#) ∗ [err](#)
- double ∗ [tmpcrd](#)
- int [m_flag](#)
- int [m_naxis](#)
- double ∗ [m_crpix](#)
- double ∗ [m_pc](#)
- double ∗ [m_cdelt](#)
- struct [disprm](#) ∗ [m_dispre](#)
- struct [disprm](#) ∗ [m_disseq](#)

18.7.1   Detailed Description

The **linprm** struct contains all of the information required to perform a linear transformation. It consists of certain members that must be set by the user (*given*) and others that are set by the WCSLIB routines (*returned*).

18.7.2   Field Documentation

18.7.2.1 flag `int linprm::flag`

(Given and returned) This flag must be set to zero whenever any of the following members of the **linprm** struct are set or modified:

- linprm::naxis (q.v., not normally set by the user),

- linprm::pc,

- linprm::cdelt,

- linprm::dispre.

- linprm::disseq.

This signals the initialization routine, linset(), to recompute the returned members of the **linprm** struct. linset() will reset flag to indicate that this has been done.

PLEASE NOTE: flag should be set to -1 when lininit() is called for the first time for a particular **linprm** struct in order to initialize memory management. It must ONLY be used on the first initialization otherwise memory leaks may result.

18.7.2.2 naxis `int linprm::naxis`

(Given or returned) Number of pixel and world coordinate elements.

If lininit() is used to initialize the **linprm** struct (as would normally be the case) then it will set naxis from the value passed to it as a function argument. The user should not subsequently modify it.

18.7.2.3 crpix `double * linprm::crpix`

(*Given*) Pointer to the first element of an array of double containing the coordinate reference pixel, **CRPIX**ja.

It is not necessary to reset the **linprm** struct (via linset()) when linprm::crpix is changed.

18.7.2.4 pc `double * linprm::pc`

(*Given*) Pointer to the first element of the **PC**i_ja (pixel coordinate) transformation matrix. The expected order is
```
struct linprm lin;
lin.pc = {PC1_1, PC1_2, PC2_1, PC2_2};
```

This may be constructed conveniently from a 2-D array via
```
double m[2][2] = {{PC1_1, PC1_2},
                  {PC2_1, PC2_2}};
```

which is equivalent to
```
double m[2][2];
m[0][0] = PC1_1;
m[0][1] = PC1_2;
m[1][0] = PC2_1;
m[1][1] = PC2_2;
```

The storage order for this 2-D array is the same as for the 1-D array, whence
```
lin.pc = *m;
```

would be legitimate.

18.7.2.5 cdelt `double * linprm::cdelt`

(*Given*) Pointer to the first element of an array of double containing the coordinate increments, **CDELT**ia.

18.7.2.6 dispre `struct disprm * linprm::dispre`

(*Given*) Pointer to a disprm struct holding parameters for prior distortion functions, or a null (0x0) pointer if there are none.

Function lindist() may be used to assign a disprm pointer to a **linprm** struct, allowing it to take control of any memory allocated for it, as in the following example:

```
void add_distortion(struct linprm *lin)
{
  struct disprm *dispre;
  dispre = malloc(sizeof(struct disprm));
  dispre->flag = -1;
  lindist(1, lin, dispre, ndpmax);
    :
   (Set up dispre.)
    :
  return;
}
```

Here, after the distortion function parameters etc. are copied into dispre, dispre is assigned using lindist() which takes control of the allocated memory. It will be freed later when linfree() is invoked on the **linprm** struct.

Consider also the following erroneous code:

```
void bad_code(struct linprm *lin)
{
  struct disprm dispre;
  dispre.flag = -1;
  lindist(1, lin, &dispre, ndpmax);   // WRONG.
    :
  return;
}
```

Here, dispre is declared as a struct, rather than a pointer. When the function returns, dispre will go out of scope and its memory will most likely be reused, thereby trashing its contents. Later, a segfault will occur when linfree() tries to free dispre's stale address.

18.7.2.7 disseq `struct disprm * linprm::disseq`

(*Given*) Pointer to a disprm struct holding parameters for sequent distortion functions, or a null (0x0) pointer if there are none.

Refer to the comments and examples given for disprm::dispre.

18.7.2.8 piximg `double * linprm::piximg`

(*Returned*) Pointer to the first element of the matrix containing the product of the **CDELT**ia diagonal matrix and the **PC**i_ja matrix.

18.7.2.9 imgpix `double * linprm::imgpix`

(*Returned*) Pointer to the first element of the inverse of the linprm::piximg matrix.

18.7.2.10 i_naxis `int linprm::i_naxis`

(*Returned*) The dimension of linprm::piximg and linprm::imgpix (normally equal to naxis).

18.7.2.11   unity `int linprm::unity`

(*Returned*) True if the linear transformation matrix is unity.

18.7.2.12   affine `int linprm::affine`

(*Returned*) True if there are no distortions.

18.7.2.13   simple `int linprm::simple`

(*Returned*) True if unity and no distortions.

18.7.2.14   err `struct` wcserr `* linprm::err`

(*Returned*) If enabled, when an error status is returned, this struct contains detailed information about the error, see wcserr_enable().

18.7.2.15   tmpcrd `double * linprm::tmpcrd`

(For internal use only.)

18.7.2.16   m_flag `int linprm::m_flag`

(For internal use only.)

18.7.2.17   m_naxis `int linprm::m_naxis`

(For internal use only.)

18.7.2.18   m_crpix `double * linprm::m_crpix`

(For internal use only.)

18.7.2.19   m_pc `double * linprm::m_pc`

(For internal use only.)

18.7.2.20   m_cdelt `double * linprm::m_cdelt`

(For internal use only.)

18.7.2.21   m_dispre `struct` disprm `* linprm::m_dispre`

(For internal use only.)

18.7.2.22 m_disseq `struct disprm * linprm::m_disseq`

(For internal use only.)

18.8   prjprm Struct Reference

Projection parameters.

`#include <prj.h>`

Data Fields

- int flag
- char code [4]
- double r0
- double pv [PVN]
- double phi0
- double theta0
- int bounds
- char name [40]
- int category
- int pvrange
- int simplezen
- int equiareal
- int conformal
- int global
- int divergent
- double x0
- double y0
- struct wcserr ∗ err
- void ∗ padding
- double w [10]
- int m
- int n
- int(∗ prjx2s)(PRJX2S_ARGS)
- int(∗ prjs2x)(PRJS2X_ARGS)

18.8.1   Detailed Description

The **prjprm** struct contains all information needed to project or deproject native spherical coordinates. It consists of certain members that must be set by the user (*given*) and others that are set by the WCSLIB routines (*returned*). Some of the latter are supplied for informational purposes while others are for internal use only.

18.8.2   Field Documentation

18.8.2.1 flag `int prjprm::flag`

(Given and returned) This flag must be set to zero whenever any of the following **prjprm** struct members are set or changed:

- prjprm::code,

- prjprm::r0,

- prjprm::pv[],

- prjprm::phi0,

- prjprm::theta0.

This signals the initialization routine (prjset() or **???set()**) to recompute the returned members of the **prjprm** struct. flag will then be reset to indicate that this has been done.

Note that flag need not be reset when prjprm::bounds is changed.

18.8.2.2 code `char prjprm::code`

(*Given*) Three-letter projection code defined by the FITS standard.

18.8.2.3 r0 `double prjprm::r0`

(*Given*) The radius of the generating sphere for the projection, a linear scaling parameter. If this is zero, it will be reset to its default value of $180°/\pi$ (the value for FITS WCS).

18.8.2.4 pv `double prjprm::pv`

(*Given*) Projection parameters. These correspond to the **PV**i_ma keywords in FITS, so pv[0] is **PV**i_**0**a, pv[1] is **PV**i_**1**a, etc., where i denotes the latitude-like axis. Many projections use pv[1] (**PV**i_**1**a), some also use pv[2] (**PV**i_**2**a) and **SZP** uses pv[3] (**PV**i_**3**a). **ZPN** is currently the only projection that uses any of the others.

Usage of the pv[] array as it applies to each projection is described in the prologue to each trio of projection routines in prj.c.

18.8.2.5 phi0 `double prjprm::phi0`

(*Given*) The native longitude, ϕ_0 [deg], and ...

18.8.2.6 theta0 `double prjprm::theta0`

(*Given*) ... the native latitude, θ_0 [deg], of the reference point, i.e. the point (x, y) = (0,0). If undefined (set to a magic value by prjini()) the initialization routine will set this to a projection-specific default.

18.8.2.7 bounds `int prjprm::bounds`

(*Given*) Controls bounds checking. If bounds&1 then enable strict bounds checking for the spherical-to-Cartesian (s2x) transformation for the **AZP**, **SZP**, **TAN**, **SIN**, **ZPN**, and **COP** projections. If bounds&2 then enable strict bounds checking for the Cartesian-to-spherical transformation (x2s) for the **HPX** and XPH projections. If bounds&4 then the Cartesian- to-spherical transformations (x2s) will invoke prjbchk() to perform bounds checking on the computed native coordinates, with a tolerance set to suit each projection. bounds is set to 7 by prjini() by default which enables all checks. Zero it to disable all checking.

It is not necessary to reset the **prjprm** struct (via prjset() or **???set()**) when prjprm::bounds is changed.

The remaining members of the **prjprm** struct are maintained by the setup routines and must not be modified elsewhere:

18.8.2.8 name `char prjprm::name`

(*Returned*) Long name of the projection.

Provided for information only, not used by the projection routines.

18.8.2.9 category `int prjprm::category`

(*Returned*) Projection category matching the value of the relevant global variable:

- ZENITHAL,

- CYLINDRICAL,

- PSEUDOCYLINDRICAL,

- CONVENTIONAL,

- CONIC,

- POLYCONIC,

- QUADCUBE, and

- HEALPIX.

The category name may be identified via the prj_categories character array, e.g.
```
struct prjprm prj;
  ...
printf("%s\n", prj_categories[prj.category]);
```

Provided for information only, not used by the projection routines.

18.8.2.10 pvrange `int prjprm::pvrange`

(*Returned*) Range of projection parameter indices: 100 times the first allowed index plus the number of parameters, e.g. **TAN** is 0 (no parameters), **SZP** is 103 (1 to 3), and **ZPN** is 30 (0 to 29).

Provided for information only, not used by the projection routines.

18.8.2.11   simplezen `int prjprm::simplezen`

(*Returned*) True if the projection is a radially-symmetric zenithal projection.

Provided for information only, not used by the projection routines.

18.8.2.12   equiareal `int prjprm::equiareal`

(*Returned*) True if the projection is equal area.

Provided for information only, not used by the projection routines.

18.8.2.13   conformal `int prjprm::conformal`

(*Returned*) True if the projection is conformal.

Provided for information only, not used by the projection routines.

18.8.2.14   global `int prjprm::global`

(*Returned*) True if the projection can represent the whole sphere in a finite, non-overlapped mapping.

Provided for information only, not used by the projection routines.

18.8.2.15   divergent `int prjprm::divergent`

(*Returned*) True if the projection diverges in latitude.

Provided for information only, not used by the projection routines.

18.8.2.16   x0 `double prjprm::x0`

(*Returned*) The offset in x,and ...

18.8.2.17   y0 `double prjprm::y0`

(*Returned*) ... the offset in y used to force (x, y) = (0,0) at (ϕ_0, θ_0).

18.8.2.18   err `struct wcserr * prjprm::err`

(*Returned*) If enabled, when an error status is returned, this struct contains detailed information about the error, see wcserr_enable().

18.8.2.19   padding `void * prjprm::padding`

(An unused variable inserted for alignment purposes only.)

18.8.2.20 w `double prjprm::w`

(*Returned*) Intermediate floating-point values derived from the projection parameters, cached here to save recomputation.

Usage of the w[] array as it applies to each projection is described in the prologue to each trio of projection routines in prj.c.

18.8.2.21 m `int prjprm::m`

18.8.2.22 n `int prjprm::n`

(*Returned*) Intermediate integer value (used only for the **ZPN** and **HPX** projections).

18.8.2.23 prjx2s `prjprm::prjx2s`

(*Returned*) Pointer to the spherical projection ...

18.8.2.24 prjs2x `prjprm::prjs2x`

(*Returned*) ... and deprojection routines.

18.9 pscard Struct Reference

Store for **PS**i_ma keyrecords.

`#include <wcs.h>`

Data Fields

- int i
- int m
- char value [72]

18.9.1 Detailed Description

The **pscard** struct is used to pass the parsed contents of **PS**i_ma keyrecords to wcsset() via the wcsprm struct.

All members of this struct are to be set by the user.

18.9.2 Field Documentation

18.9.2.1   i `int pscard::i`

(*Given*) Axis number (1-relative), as in the FITS **PS**`i_ma` keyword.

18.9.2.2   m `int pscard::m`

(*Given*) Parameter number (non-negative), as in the FITS **PS**`i_ma` keyword.

18.9.2.3   value `char pscard::value`

(*Given*) Parameter value.

18.10   pvcard Struct Reference

Store for **PV**`i_ma` keyrecords.

`#include <wcs.h>`

Data Fields

- int [i](#)
- int [m](#)
- double [value](#)

18.10.1   Detailed Description

The **pvcard** struct is used to pass the parsed contents of **PV**`i_ma` keyrecords to [wcsset()](#) via the wcsprm struct.

All members of this struct are to be set by the user.

18.10.2   Field Documentation

18.10.2.1   i `int pvcard::i`

(*Given*) Axis number (1-relative), as in the FITS **PV**`i_ma` keyword. If i == 0, [wcsset()](#) will replace it with the latitude axis number.

18.10.2.2   m `int pvcard::m`

(*Given*) Parameter number (non-negative), as in the FITS **PV**`i_ma` keyword.

18.10.2.3   value `double pvcard::value`

(*Given*) Parameter value.

18.11 spcprm Struct Reference

Spectral transformation parameters.

```
#include <spc.h>
```

Data Fields

- int flag
- char type [8]
- char code [4]
- double crval
- double restfrq
- double restwav
- double pv [7]
- double w [6]
- int isGrism
- int padding1
- struct wcserr ∗ err
- void ∗ padding2
- int(∗ spxX2P)(SPX_ARGS)
- int(∗ spxP2S)(SPX_ARGS)
- int(∗ spxS2P)(SPX_ARGS)
- int(∗ spxP2X)(SPX_ARGS)

18.11.1 Detailed Description

The **spcprm** struct contains information required to transform spectral coordinates. It consists of certain members that must be set by the user (*given*) and others that are set by the WCSLIB routines (*returned*). Some of the latter are supplied for informational purposes while others are for internal use only.

18.11.2 Field Documentation

18.11.2.1 flag `int spcprm::flag`

(Given and returned) This flag must be set to zero whenever any of the following **spcprm** structure members are set or changed:

- spcprm::type,

- spcprm::code,

- spcprm::crval,

- spcprm::restfrq,

- spcprm::restwav,

- spcprm::pv[].

This signals the initialization routine, spcset(), to recompute the returned members of the **spcprm** struct. spcset() will reset flag to indicate that this has been done.

18.11.2.2   type `char spcprm::type`

(*Given*) Four-letter spectral variable type, e.g "ZOPT" for **CTYPE**ia = **'ZOPT−F2W'**. (Declared as char[8] for alignment reasons.)

18.11.2.3   code `char spcprm::code`

(*Given*) Three-letter spectral algorithm code, e.g "F2W" for **CTYPE**ia = **'ZOPT−F2W'**.

18.11.2.4   crval `double spcprm::crval`

(*Given*) Reference value (**CRVAL**ia), SI units.

18.11.2.5   restfrq `double spcprm::restfrq`

(*Given*) The rest frequency [Hz], and ...

18.11.2.6   restwav `double spcprm::restwav`

(*Given*) ... the rest wavelength in vacuo [m], only one of which need be given, the other should be set to zero. Neither are required if the X and S spectral variables are both wave-characteristic, or both velocity-characteristic, types.

18.11.2.7   pv `double spcprm::pv`

(*Given*) Grism parameters for '**GRI**' and '**GRA**' algorithm codes:

- 0: G, grating ruling density.

- 1: m, interference order.

- 2: α, angle of incidence [deg].

- 3: n_r, refractive index at the reference wavelength, λ_r.

- 4: n'_r, $dn/d\lambda$ at the reference wavelength, λ_r (/m).

- 5: ϵ, grating tilt angle [deg].

- 6: θ, detector tilt angle [deg].

The remaining members of the **spcprm** struct are maintained by spcset() and must not be modified elsewhere:

18.11.2.8   w `double spcprm::w`

(*Returned*) Intermediate values:

- 0: Rest frequency or wavelength (SI).

- 1: The value of the X-type spectral variable at the reference point (SI units).

- 2: dX/dS at the reference point (SI units).

The remainder are grism intermediates.

18.11.2.9 isGrism `int spcprm::isGrism`

(*Returned*) Grism coordinates?

- 0: no,

- 1: in vacuum,

- 2: in air.

18.11.2.10 padding1 `int spcprm::padding1`

(An unused variable inserted for alignment purposes only.)

18.11.2.11 err `struct` wcserr `* spcprm::err`

(*Returned*) If enabled, when an error status is returned, this struct contains detailed information about the error, see wcserr_enable().

18.11.2.12 padding2 `void * spcprm::padding2`

(An unused variable inserted for alignment purposes only.)

18.11.2.13 spxX2P `spcprm::spxX2P`

(*Returned*) The first and ...

18.11.2.14 spxP2S `spcprm::spxP2S`

(*Returned*) ... the second of the pointers to the transformation functions in the two-step algorithm chain $X \rightsquigarrow P \rightarrow S$ in the pixel-to-spectral direction where the non-linear transformation is from X to P. The argument list, SPX_ARGS, is defined in spx.h.

18.11.2.15 spxS2P `spcprm::spxS2P`

(*Returned*) The first and ...

18.11.2.16 spxP2X `spcprm::spxP2X`

(*Returned*) ... the second of the pointers to the transformation functions in the two-step algorithm chain $S \rightarrow P \rightsquigarrow X$ in the spectral-to-pixel direction where the non-linear transformation is from P to X. The argument list, SPX_ARGS, is defined in spx.h.

18.12 spxprm Struct Reference

Spectral variables and their derivatives.

```
#include <spx.h>
```

Data Fields

- double restfrq
- double restwav
- int wavetype
- int velotype
- double freq
- double afrq
- double ener
- double wavn
- double vrad
- double wave
- double vopt
- double zopt
- double awav
- double velo
- double beta
- double dfreqafrq
- double dafrqfreq
- double dfreqener
- double denerfreq
- double dfreqwavn
- double dwavnfreq
- double dfreqvrad
- double dvradfreq
- double dfreqwave
- double dwavefreq
- double dfreqawav
- double dawavfreq
- double dfreqvelo
- double dvelofreq
- double dwavevopt
- double dvoptwave
- double dwavezopt
- double dzoptwave
- double dwaveawav
- double dawavwave
- double dwavevelo
- double dvelowave
- double dawavvelo
- double dveloawav
- double dvelobeta
- double dbetavelo
- struct wcserr ∗ err
- void ∗ padding

18.12.1 Detailed Description

The **spxprm** struct contains the value of all spectral variables and their derivatives. It is used solely by specx() which constructs it from information provided via its function arguments.

This struct should be considered read-only, no members need ever be set nor should ever be modified by the user.

18.12.2 Field Documentation

18.12.2.1 restfrq `double spxprm::restfrq`

(*Returned*) Rest frequency [Hz].

18.12.2.2 restwav `double spxprm::restwav`

(*Returned*) Rest wavelength [m].

18.12.2.3 wavetype `int spxprm::wavetype`

(*Returned*) True if wave types have been computed, and ...

18.12.2.4 velotype `int spxprm::velotype`

(*Returned*) ... true if velocity types have been computed; types are defined below.

If one or other of spxprm::restfrq and spxprm::restwav is given (non-zero) then all spectral variables may be computed. If both are given, restfrq is used. If restfrq and restwav are both zero, only wave characteristic xor velocity type spectral variables may be computed depending on the variable given. These flags indicate what is available.

18.12.2.5 freq `double spxprm::freq`

(*Returned*) Frequency [Hz] (*wavetype*).

18.12.2.6 afrq `double spxprm::afrq`

(*Returned*) Angular frequency [rad/s] (*wavetype*).

18.12.2.7 ener `double spxprm::ener`

(*Returned*) Photon energy [J] (*wavetype*).

18.12.2.8 wavn `double spxprm::wavn`

(*Returned*) Wave number [/m] (*wavetype*).

18.12.2.9 vrad `double spxprm::vrad`

(*Returned*) Radio velocity [m/s] (*velotype*).

18.12.2.10 wave `double spxprm::wave`

(*Returned*) Vacuum wavelength [m] (*wavetype*).

18.12.2.11 vopt `double spxprm::vopt`

(*Returned*) Optical velocity [m/s] (*velotype*).

18.12.2.12 zopt `double spxprm::zopt`

(*Returned*) Redshift [dimensionless] (*velotype*).

18.12.2.13 awav `double spxprm::awav`

(*Returned*) Air wavelength [m] (*wavetype*).

18.12.2.14 velo `double spxprm::velo`

(*Returned*) Relativistic velocity [m/s] (*velotype*).

18.12.2.15 beta `double spxprm::beta`

(*Returned*) Relativistic beta [dimensionless] (*velotype*).

18.12.2.16 dfreqafrq `double spxprm::dfreqafrq`

(*Returned*) Derivative of frequency with respect to angular frequency [/rad] (constant, $= 1/2\pi$), and ...

18.12.2.17 dafrqfreq `double spxprm::dafrqfreq`

(*Returned*) ... vice versa [rad] (constant, $= 2\pi$, always available).

18.12.2.18 dfreqener `double spxprm::dfreqener`

(*Returned*) Derivative of frequency with respect to photon energy [/J/s] (constant, $= 1/h$), and ...

18.12.2.19 denerfreq `double spxprm::denerfreq`

(*Returned*) ... vice versa [Js] (constant, $= h$, Planck's constant, always available).

18.12.2.20 dfreqwavn `double spxprm::dfreqwavn`

(*Returned*) Derivative of frequency with respect to wave number [m/s] (constant, $= c$, the speed of light in vacuo), and ...

18.12.2.21 dwavnfreq `double spxprm::dwavnfreq`

(*Returned*) ... vice versa [s/m] (constant, $= 1/c$, always available).

18.12.2.22 dfreqvrad `double spxprm::dfreqvrad`

(*Returned*) Derivative of frequency with respect to radio velocity [/m], and ...

18.12.2.23 dvradfreq `double spxprm::dvradfreq`

(*Returned*) ... vice versa [m] (*wavetype* && *velotype*).

18.12.2.24 dfreqwave `double spxprm::dfreqwave`

(*Returned*) Derivative of frequency with respect to vacuum wavelength [/m/s], and ...

18.12.2.25 dwavefreq `double spxprm::dwavefreq`

(*Returned*) ... vice versa [m s] (*wavetype*).

18.12.2.26 dfreqawav `double spxprm::dfreqawav`

(*Returned*) Derivative of frequency with respect to air wavelength, [/m/s], and ...

18.12.2.27 dawavfreq `double spxprm::dawavfreq`

(*Returned*) ... vice versa [m s] (*wavetype*).

18.12.2.28 dfreqvelo `double spxprm::dfreqvelo`

(*Returned*) Derivative of frequency with respect to relativistic velocity [/m], and ...

18.12.2.29 dvelofreq `double spxprm::dvelofreq`

(*Returned*) ... vice versa [m] (*wavetype* && *velotype*).

18.12.2.30 dwavevopt `double spxprm::dwavevopt`

(*Returned*) Derivative of vacuum wavelength with respect to optical velocity [s], and ...

18.12.2.31 dvoptwave `double spxprm::dvoptwave`

(*Returned*) ... vice versa [/s] (*wavetype* && *velotype*).

18.12.2.32 dwavezopt `double spxprm::dwavezopt`

(*Returned*) Derivative of vacuum wavelength with respect to redshift [m], and ...

18.12.2.33 dzoptwave `double spxprm::dzoptwave`

(*Returned*) ... vice versa [/m] (*wavetype* && *velotype*).

18.12.2.34 dwaveawav `double spxprm::dwaveawav`

(*Returned*) Derivative of vacuum wavelength with respect to air wavelength [dimensionless], and ...

18.12.2.35 dawavwave `double spxprm::dawavwave`

(*Returned*) ... vice versa [dimensionless] (*wavetype*).

18.12.2.36 dwavevelo `double spxprm::dwavevelo`

(*Returned*) Derivative of vacuum wavelength with respect to relativistic velocity [s], and ...

18.12.2.37 dvelowave `double spxprm::dvelowave`

(*Returned*) ... vice versa [/s] (*wavetype* && *velotype*).

18.12.2.38 dawavvelo `double spxprm::dawavvelo`

(*Returned*) Derivative of air wavelength with respect to relativistic velocity [s], and ...

18.12.2.39 dveloawav `double spxprm::dveloawav`

(*Returned*) ... vice versa [/s] (*wavetype* && *velotype*).

18.12.2.40 dvelobeta `double spxprm::dvelobeta`

(*Returned*) Derivative of relativistic velocity with respect to relativistic beta [m/s] (constant, $= c$, the speed of light in vacuo), and ...

18.12.2.41 dbetavelo `double spxprm::dbetavelo`

(*Returned*) ... vice versa [s/m] (constant, $= 1/c$, always available).

18.12.2.42 err `struct wcserr * spxprm::err`

(*Returned*) If enabled, when an error status is returned, this struct contains detailed information about the error, see wcserr_enable().

18.12.2.43 padding `void * spxprm::padding`

(An unused variable inserted for alignment purposes only.)

Global variable: const char ∗spx_errmsg[] - Status return messages Error messages to match the status value returned from each function.

18.13 tabprm Struct Reference

Tabular transformation parameters.

`#include <tab.h>`

Data Fields

- int flag
- int M
- int ∗ K
- int ∗ map
- double ∗ crval
- double ∗∗ index
- double ∗ coord
- int nc
- int padding
- int ∗ sense
- int ∗ p0
- double ∗ delta
- double ∗ extrema
- struct wcserr ∗ err
- int m_flag
- int m_M
- int m_N
- int set_M
- int ∗ m_K
- int ∗ m_map
- double ∗ m_crval
- double ∗∗ m_index
- double ∗∗ m_indxs
- double ∗ m_coord

18.13.1 Detailed Description

The **tabprm** struct contains information required to transform tabular coordinates. It consists of certain members that must be set by the user (*given*) and others that are set by the WCSLIB routines (*returned*). Some of the latter are supplied for informational purposes while others are for internal use only.

18.13.2 Field Documentation

18.13.2.1 flag `int tabprm::flag`

(Given and returned) This flag must be set to zero whenever any of the following **tabprm** structure members are set or changed:

- tabprm::M (q.v., not normally set by the user),

- tabprm::K (q.v., not normally set by the user),

- tabprm::map,

- tabprm::crval,

- tabprm::index,

- tabprm::coord.

This signals the initialization routine, tabset(), to recompute the returned members of the **tabprm** struct. tabset() will reset flag to indicate that this has been done.

PLEASE NOTE: flag should be set to -1 when tabini() is called for the first time for a particular **tabprm** struct in order to initialize memory management. It must ONLY be used on the first initialization otherwise memory leaks may result.

18.13.2.2 M `int tabprm::M`

(Given or returned) Number of tabular coordinate axes.

If tabini() is used to initialize the **tabprm** struct (as would normally be the case) then it will set M from the value passed to it as a function argument. The user should not subsequently modify it.

18.13.2.3 K `int * tabprm::K`

(Given or returned) Pointer to the first element of a vector of length tabprm::M whose elements $(K_1, K_2, ... K_M)$ record the lengths of the axes of the coordinate array and of each indexing vector.

If tabini() is used to initialize the **tabprm** struct (as would normally be the case) then it will set K from the array passed to it as a function argument. The user should not subsequently modify it.

18.13.2.4 map `int * tabprm::map`

(*Given*) Pointer to the first element of a vector of length tabprm::M that defines the association between axis m in the M-dimensional coordinate array ($1 \leq m \leq M$) and the indices of the intermediate world coordinate and world coordinate arrays, x[] and world[], in the argument lists for tabx2s() and tabs2x().

When x[] and world[] contain the full complement of coordinate elements in image-order, as will usually be the case, then map[m-1] == i-1 for axis i in the N-dimensional image ($1 \leq i \leq N$). In terms of the FITS keywords

map[**PV**i_3a - 1] == i - 1.

However, a different association may result if x[], for example, only contains a (relevant) subset of intermediate world coordinate elements. For example, if M == 1 for an image with N > 1, it is possible to fill x[] with the relevant coordinate element with nelem set to 1. In this case map[0] = 0 regardless of the value of i.

18.13.2.5 crval `double * tabprm::crval`

(*Given*) Pointer to the first element of a vector of length [tabprm::M](#) whose elements contain the index value for the reference pixel for each of the tabular coordinate axes.

18.13.2.6 index `double ** tabprm::index`

(*Given*) Pointer to the first element of a vector of length [tabprm::M](#) of pointers to vectors of lengths $(K_1, K_2, ...K_M)$ of 0-relative indexes (see [tabprm::K](#)).

The address of any or all of these index vectors may be set to zero, i.e.
`index[m] == 0;`

this is interpreted as default indexing, i.e.
`index[m][k] = k;`

18.13.2.7 coord `double * tabprm::coord`

(*Given*) Pointer to the first element of the tabular coordinate array, treated as though it were defined as
`double coord[K_M]...[K_2][K_1][M];`

(see [tabprm::K](#)) i.e. with the M dimension varying fastest so that the M elements of a coordinate vector are stored contiguously in memory.

18.13.2.8 nc `int tabprm::nc`

(*Returned*) Total number of coordinate vectors in the coordinate array being the product $K_1 K_2 \ldots K_M$ (see [tabprm::K](#)).

18.13.2.9 padding `int tabprm::padding`

(An unused variable inserted for alignment purposes only.)

18.13.2.10 sense `int * tabprm::sense`

(*Returned*) Pointer to the first element of a vector of length [tabprm::M](#) whose elements indicate whether the corresponding indexing vector is monotonic increasing (+1), or decreasing (-1).

18.13.2.11 p0 `int * tabprm::p0`

(*Returned*) Pointer to the first element of a vector of length [tabprm::M](#) of interpolated indices into the coordinate array such that Υ_m, as defined in Paper III, is equal to (p0[m] + 1) + [tabprm::delta](#)[m].

18.13.2.12 delta `double * tabprm::delta`

(*Returned*) Pointer to the first element of a vector of length [tabprm::M](#) of interpolated indices into the coordinate array such that Υ_m, as defined in Paper III, is equal to ([tabprm::p0](#)[m] + 1) + delta[m].

18.13.2.13 extrema `double * tabprm::extrema`

(*Returned*) Pointer to the first element of an array that records the minimum and maximum value of each element of the coordinate vector in each row of the coordinate array, treated as though it were defined as
`double extrema[K_M]...[K_2][2][M]`

(see tabprm::K). The minimum is recorded in the first element of the compressed K_1 dimension, then the maximum. This array is used by the inverse table lookup function, tabs2x(), to speed up table searches.

18.13.2.14 err `struct wcserr * tabprm::err`

(*Returned*) If enabled, when an error status is returned, this struct contains detailed information about the error, see wcserr_enable().

18.13.2.15 m_flag `int tabprm::m_flag`

(For internal use only.)

18.13.2.16 m_M `int tabprm::m_M`

(For internal use only.)

18.13.2.17 m_N `int tabprm::m_N`

(For internal use only.)

18.13.2.18 set_M `int tabprm::set_M`

(For internal use only.)

18.13.2.19 m_K `int tabprm::m_K`

(For internal use only.)

18.13.2.20 m_map `int tabprm::m_map`

(For internal use only.)

18.13.2.21 m_crval `int tabprm::m_crval`

(For internal use only.)

18.13.2.22 m_index `int tabprm::m_index`

(For internal use only.)

18.13.2.23 m_indxs `int tabprm::m_indxs`

(For internal use only.)

18.13.2.24 m_coord `int tabprm::m_coord`

(For internal use only.)

18.14 wcserr Struct Reference

Error message handling.

`#include <wcserr.h>`

Data Fields

- int status
- int line_no
- const char ∗ function
- const char ∗ file
- char ∗ msg

18.14.1 Detailed Description

The **wcserr** struct contains the numeric error code, a textual description of the error, and information about the function, source file, and line number where the error was generated.

18.14.2 Field Documentation

18.14.2.1 status `int wcserr::status`

Numeric status code associated with the error, the meaning of which depends on the function that generated it. See the documentation for the particular function.

18.14.2.2 line_no `int wcserr::line_no`

Line number where the error occurred as given by the __LINE__ preprocessor macro.

const char ∗function Name of the function where the error occurred.

const char ∗file Name of the source file where the error occurred as given by the __FILE__ preprocessor macro.

18.14.2.3 function `const char* wcserr::function`

18.14.2.4 file `const char* wcserr::file`

18.14.2.5 msg `char * wcserr::msg`

Informative error message.

18.15 wcsprm Struct Reference

Coordinate transformation parameters.

`#include <wcs.h>`

Data Fields

- int flag
- int naxis
- double ∗ crpix
- double ∗ pc
- double ∗ cdelt
- double ∗ crval
- char(∗ cunit)[72]
- char(∗ ctype)[72]
- double lonpole
- double latpole
- double restfrq
- double restwav
- int npv
- int npvmax
- struct pvcard ∗ pv
- int nps
- int npsmax
- struct pscard ∗ ps
- double ∗ cd
- double ∗ crota
- int altlin
- int velref
- char alt [4]
- int colnum
- int ∗ colax
- char(∗ cname)[72]
- double ∗ crder
- double ∗ csyer
- double ∗ czphs
- double ∗ cperi
- char wcsname [72]
- char timesys [72]
- char trefpos [72]
- char trefdir [72]
- char plephem [72]
- char timeunit [72]

- char dateref [72]
- double mjdref [2]
- double timeoffs
- char dateobs [72]
- char datebeg [72]
- char dateavg [72]
- char dateend [72]
- double mjdobs
- double mjdbeg
- double mjdavg
- double mjdend
- double jepoch
- double bepoch
- double tstart
- double tstop
- double xposure
- double telapse
- double timsyer
- double timrder
- double timedel
- double timepixr
- double obsgeo [6]
- char obsorbit [72]
- char radesys [72]
- double equinox
- char specsys [72]
- char ssysobs [72]
- double velosys
- double zsource
- char ssyssrc [72]
- double velangl
- struct auxprm ∗ aux
- int ntab
- int nwtb
- struct tabprm ∗ tab
- struct wtbarr ∗ wtb
- char lngtyp [8]
- char lattyp [8]
- int lng
- int lat
- int spec
- int cubeface
- int ∗ types
- struct linprm lin
- struct celprm cel
- struct spcprm spc
- struct wcserr ∗ err
- int m_flag
- int m_naxis
- double ∗ m_crpix
- double ∗ m_pc
- double ∗ m_cdelt
- double ∗ m_crval
- char(∗ m_cunit)[72]
- char((∗ m_ctype)[72]

- struct pvcard ∗ m_pv
- struct pscard ∗ m_ps
- double ∗ m_cd
- double ∗ m_crota
- int ∗ m_colax
- char(∗ m_cname)[72]
- double ∗ m_crder
- double ∗ m_csyer
- double ∗ m_czphs
- double ∗ m_cperi
- struct auxprm ∗ m_aux
- struct tabprm ∗ m_tab
- struct wtbarr ∗ m_wtb

18.15.1 Detailed Description

The **wcsprm** struct contains information required to transform world coordinates. It consists of certain members that must be set by the user (*given*) and others that are set by the WCSLIB routines (*returned*). While the addresses of the arrays themselves may be set by wcsinit() if it (optionally) allocates memory, their contents must be set by the user.

Some parameters that are given are not actually required for transforming coordinates. These are described as "auxiliary"; the struct simply provides a place to store them, though they may be used by wcshdo() in constructing a FITS header from a **wcsprm** struct. Some of the returned values are supplied for informational purposes and others are for internal use only as indicated.

In practice, it is expected that a WCS parser would scan the FITS header to determine the number of coordinate axes. It would then use wcsinit() to allocate memory for arrays in the **wcsprm** struct and set default values. Then as it reread the header and identified each WCS keyrecord it would load the value into the relevant **wcsprm** array element. This is essentially what wcspih() does - refer to the prologue of wcshdr.h. As the final step, wcsset() is invoked, either directly or indirectly, to set the derived members of the **wcsprm** struct. wcsset() strips off trailing blanks in all string members and null-fills the character array.

18.15.2 Field Documentation

18.15.2.1 flag `int wcsprm::flag`

(Given and returned) This flag must be set to zero whenever any of the following **wcsprm** struct members are set or changed:

- wcsprm::naxis (q.v., not normally set by the user),

- wcsprm::crpix,

- wcsprm::pc,

- wcsprm::cdelt,

- wcsprm::crval,

- wcsprm::cunit,

- wcsprm::ctype,

- wcsprm::lonpole,

- wcsprm::latpole,

- wcsprm::restfrq,

- wcsprm::restwav,

- wcsprm::npv,

- wcsprm::pv,

- wcsprm::nps,

- wcsprm::ps,

- wcsprm::cd,

- wcsprm::crota,

- wcsprm::altlin,

- wcsprm::ntab,

- wcsprm::nwtb,

- wcsprm::tab,

- wcsprm::wtb.

This signals the initialization routine, wcsset(), to recompute the returned members of the linprm, celprm, spcprm, and tabprm structs. wcsset() will reset flag to indicate that this has been done.

PLEASE NOTE: flag should be set to -1 when wcsinit() is called for the first time for a particular **wcsprm** struct in order to initialize memory management. It must ONLY be used on the first initialization otherwise memory leaks may result.

18.15.2.2 naxis `int wcsprm::naxis`

(Given or returned) Number of pixel and world coordinate elements.

If wcsinit() is used to initialize the linprm struct (as would normally be the case) then it will set naxis from the value passed to it as a function argument. The user should not subsequently modify it.

18.15.2.3 crpix `double * wcsprm::crpix`

(*Given*) Address of the first element of an array of double containing the coordinate reference pixel, **CRPIX**ja.

18.15.2.4 pc `double * wcsprm::pc`

(*Given*) Address of the first element of the **PC**i_ja (pixel coordinate) transformation matrix. The expected order is
```
struct wcsprm wcs;
wcs.pc = {PC1_1, PC1_2, PC2_1, PC2_2};
```

This may be constructed conveniently from a 2-D array via
```
double m[2][2] = {{PC1_1, PC1_2},
                  {PC2_1, PC2_2}};
```

which is equivalent to
```
double m[2][2];
m[0][0] = PC1_1;
m[0][1] = PC1_2;
m[1][0] = PC2_1;
m[1][1] = PC2_2;
```

The storage order for this 2-D array is the same as for the 1-D array, whence
```
wcs.pc = *m;
```

would be legitimate.

18.15.2.5 cdelt `double * wcsprm::cdelt`

(*Given*) Address of the first element of an array of double containing the coordinate increments, **CDELT**ia.

18.15.2.6 crval `double * wcsprm::crval`

(*Given*) Address of the first element of an array of double containing the coordinate reference values, **CRVAL**ia.

18.15.2.7 cunit `wcsprm::cunit`

(*Given*) Address of the first element of an array of char[72] containing the **CUNIT**ia keyvalues which define the units of measurement of the **CRVAL**ia, **CDELT**ia, and **CD**i_ja keywords.

As **CUNIT**ia is an optional header keyword, cunit[][72] may be left blank but otherwise is expected to contain a standard units specification as defined by WCS Paper I. Utility function wcsutrn(), described in wcsunits.h, is available to translate commonly used non-standard units specifications but this must be done as a separate step before invoking wcsset().

For celestial axes, if cunit[][72] is not blank, wcsset() uses wcsunits() to parse it and scale cdelt[], crval[], and cd[][∗] to degrees. It then resets cunit[][72] to "deg".

For spectral axes, if cunit[][72] is not blank, wcsset() uses wcsunits() to parse it and scale cdelt[], crval[], and cd[][∗] to SI units. It then resets cunit[][72] accordingly.

wcsset() ignores cunit[][72] for other coordinate types; cunit[][72] may be used to label coordinate values.

These variables accomodate the longest allowed string-valued FITS keyword, being limited to 68 characters, plus the null-terminating character.

18.15.2.8 ctype `wcsprm::ctype`

(*Given*) Address of the first element of an array of char[72] containing the coordinate axis types, **CTYPE**ia.

The ctype[][72] keyword values must be in upper case and there must be zero or one pair of matched celestial axis types, and zero or one spectral axis. The ctype[][72] strings should be padded with blanks on the right and null-terminated so that they are at least eight characters in length.

These variables accomodate the longest allowed string-valued FITS keyword, being limited to 68 characters, plus the null-terminating character.

18.15.2.9 lonpole `double wcsprm::lonpole`

(Given and returned) The native longitude of the celestial pole, ϕ_p, given by **LONPOLE**a [deg] or by **PVi_2a** [deg] attached to the longitude axis which takes precedence if defined, and ...

18.15.2.10 latpole `double wcsprm::latpole`

(Given and returned) ... the native latitude of the celestial pole, θ_p, given by **LATPOLE**a [deg] or by **PVi_3a** [deg] attached to the longitude axis which takes precedence if defined.

lonpole and latpole may be left to default to values set by wcsinit() (see celprm::ref), but in any case they will be reset by wcsset() to the values actually used. Note therefore that if the **wcsprm** struct is reused without resetting them, whether directly or via wcsinit(), they will no longer have their default values.

18.15.2.11 restfrq `double wcsprm::restfrq`

(*Given*) The rest frequency [Hz], and/or ...

18.15.2.12 restwav `double wcsprm::restwav`

(*Given*) ... the rest wavelength in vacuo [m], only one of which need be given, the other should be set to zero.

18.15.2.13 npv `int wcsprm::npv`

(*Given*) The number of entries in the [wcsprm::pv](#)[] array.

18.15.2.14 npvmax `int wcsprm::npvmax`

(Given or returned) The length of the [wcsprm::pv](#)[] array.

npvmax will be set by [wcsinit()](#) if it allocates memory for [wcsprm::pv](#)[], otherwise it must be set by the user. See also [wcsnpv()](#).

18.15.2.15 pv `struct` [pvcard](#) `* wcsprm::pv`

(*Given*) Address of the first element of an array of length npvmax of pvcard structs.

As a FITS header parser encounters each **PV**`i_ma` keyword it should load it into a pvcard struct in the array and increment npv. [wcsset()](#) interprets these as required.

Note that, if they were not given, [wcsset()](#) resets the entries for **PVi_1a**, **PVi_2a**, **PVi_3a**, and **PVi_4a** for longitude axis **i** to match phi_0 and theta_0 (the native longitude and latitude of the reference point), **LONPOLE**a and **LATPOLE**a respectively.

18.15.2.16 nps `int wcsprm::nps`

(*Given*) The number of entries in the [wcsprm::ps](#)[] array.

18.15.2.17 npsmax `int wcsprm::npsmax`

(Given or returned) The length of the [wcsprm::ps](#)[] array.

npsmax will be set by [wcsinit()](#) if it allocates memory for [wcsprm::ps](#)[], otherwise it must be set by the user. See also [wcsnps()](#).

18.15.2.18 ps `struct` [pscard](#) `* wcsprm::ps`

(*Given*) Address of the first element of an array of length npsmax of pscard structs.

As a FITS header parser encounters each **PS**`i_ma` keyword it should load it into a pscard struct in the array and increment nps. [wcsset()](#) interprets these as required (currently no **PS**`i_ma` keyvalues are recognized).

18.15.2.19 cd `double * wcsprm::cd`

(*Given*) For historical compatibility, the **wcsprm** struct supports two alternate specifications of the linear transformation matrix, those associated with the **CD**`i_ja` keywords, and ...

18.15.2.20 crota `double * wcsprm::crota`

(*Given*) ... those associated with the **CROTA**`i` keywords. Although these may not formally co-exist with **PC**`i_ja`, the approach taken here is simply to ignore them if given in conjunction with **PC**`i_ja`.

18.15.2.21 altlin `int wcsprm::altlin`

(*Given*) altlin is a bit flag that denotes which of the **PC**`i_ja`, **CD**`i_ja` and **CROTA**`i` keywords are present in the header:

- Bit 0: **PC**`i_ja` is present.

- Bit 1: **CD**`i_ja` is present.

  Matrix elements in the IRAF convention are equivalent to the product **CD**`i_ja` = **CDELT**`ia` ∗ **PC**`i_ja`, but the defaults differ from that of the **PC**`i_ja` matrix. If one or more **CD**`i_ja` keywords are present then all unspecified **CD**`i_ja` default to zero. If no **CD**`i_ja` (or **CROTA**`i`) keywords are present, then the header is assumed to be in **PC**`i_ja` form whether or not any **PC**`i_ja` keywords are present since this results in an interpretation of **CDELT**`ia` consistent with the original FITS specification.

  While **CD**`i_ja` may not formally co-exist with **PC**`i_ja`, it may co-exist with **CDELT**`ia` and **CROTA**`i` which are to be ignored.

- Bit 2: **CROTA**`i` is present.

  In the AIPS convention, **CROTA**`i` may only be associated with the latitude axis of a celestial axis pair. It specifies a rotation in the image plane that is applied AFTER the **CDELT**`ia`; any other **CROTA**`i` keywords are ignored.

  CROTA`i` may not formally co-exist with **PC**`i_ja`.

  CROTA`i` and **CDELT**`ia` may formally co-exist with **CD**`i_ja` but if so are to be ignored.

- Bit 3: **PC**`i_ja` + **CDELT**`ia` was derived from **CD**`i_ja` by wcspcx().

  This bit is set by wcspcx() when it derives **PC**`i_ja` and **CDELT**`ia` from **CD**`i_ja` via an orthonormal decomposition. In particular, it signals wcsset() not to replace **PC**`i_ja` by a copy of **CD**`i_ja` with **CDELT**`ia` set to unity.

CD`i_ja` and **CROTA**`i` keywords, if found, are to be stored in the wcsprm::cd and wcsprm::crota arrays which are dimensioned similarly to wcsprm::pc and wcsprm::cdelt. FITS header parsers should use the following procedure:

- Whenever a **PC**`i_ja` keyword is encountered:
  `altlin |= 1;`

- Whenever a **CD**`i_ja` keyword is encountered:
  `altlin |= 2;`

- Whenever a **CROTA**`i` keyword is encountered:
  `altlin |= 4;`

If none of these bits are set the **PC**`i_ja` representation results, i.e. wcsprm::pc and wcsprm::cdelt will be used as given.

These alternate specifications of the linear transformation matrix are translated immediately to **PC**`i_ja` by wcsset() and are invisible to the lower-level WCSLIB routines. In particular, unless bit 3 is also set, wcsset() resets wcsprm::cdelt to unity if **CD**`i_ja` is present (and no **PC**`i_ja`).

If **CROTA**`i` are present but none is associated with the latitude axis (and no **PC**`i_ja` or **CD**`i_ja`), then wcsset() reverts to a unity **PC**`i_ja` matrix.

18.15.2.22 velref `int wcsprm::velref`

(*Given*) AIPS velocity code **VELREF**, refer to spcaips().

It is not necessary to reset the **wcsprm** struct (via wcsset()) when wcsprm::velref is changed.

18.15.2.23 alt `char wcsprm::alt`

(Given, auxiliary) Character code for alternate coordinate descriptions (i.e. the '**a**' in keyword names such as **CTYPE**ia). This is blank for the primary coordinate description, or one of the 26 upper-case letters, A-Z.

An array of four characters is provided for alignment purposes, only the first is used.

It is not necessary to reset the **wcsprm** struct (via wcsset()) when wcsprm::alt is changed.

18.15.2.24 colnum `int wcsprm::colnum`

(Given, auxiliary) Where the coordinate representation is associated with an image-array column in a FITS binary table, this variable may be used to record the relevant column number.

It should be set to zero for an image header or pixel list.

It is not necessary to reset the **wcsprm** struct (via wcsset()) when wcsprm::colnum is changed.

18.15.2.25 colax `int * wcsprm::colax`

(Given, auxiliary) Address of the first element of an array of int recording the column numbers for each axis in a pixel list.

The array elements should be set to zero for an image header or image array in a binary table.

It is not necessary to reset the **wcsprm** struct (via wcsset()) when wcsprm::colax is changed.

18.15.2.26 cname `wcsprm::cname`

(Given, auxiliary) The address of the first element of an array of char[72] containing the coordinate axis names, **CNAME**ia.

These variables accomodate the longest allowed string-valued FITS keyword, being limited to 68 characters, plus the null-terminating character.

It is not necessary to reset the **wcsprm** struct (via wcsset()) when wcsprm::cname is changed.

18.15.2.27 crder `double * wcsprm::crder`

(Given, auxiliary) Address of the first element of an array of double recording the random error in the coordinate value, **CRDER**ia.

It is not necessary to reset the **wcsprm** struct (via wcsset()) when wcsprm::crder is changed.

18.15.2.28 csyer `double * wcsprm::csyer`

(Given, auxiliary) Address of the first element of an array of double recording the systematic error in the coordinate value, **CSYER**ia.

It is not necessary to reset the **wcsprm** struct (via wcsset()) when wcsprm::csyer is changed.

18.15.2.29 czphs `double * wcsprm::czphs`

(Given, auxiliary) Address of the first element of an array of double recording the time at the zero point of a phase axis, CZPHSia.

It is not necessary to reset the **wcsprm** struct (via wcsset()) when wcsprm::czphs is changed.

18.15.2.30 cperi `double * wcsprm::cperi`

(Given, auxiliary) Address of the first element of an array of double recording the period of a phase axis, CPERIia.

It is not necessary to reset the **wcsprm** struct (via wcsset()) when wcsprm::cperi is changed.

18.15.2.31 wcsname `char wcsprm::wcsname`

(Given, auxiliary) The name given to the coordinate representation, **WCSNAME**a. This variable accomodates the longest allowed string-valued FITS keyword, being limited to 68 characters, plus the null-terminating character.

It is not necessary to reset the **wcsprm** struct (via wcsset()) when wcsprm::wcsname is changed.

18.15.2.32 timesys `char wcsprm::timesys`

(Given, auxiliary) **TIMESYS** keyvalue, being the time scale (UTC, TAI, etc.) in which all other time-related auxiliary header values are recorded. Also defines the time scale for an image axis with **CTYPE**ia set to 'TIME'.

It is not necessary to reset the **wcsprm** struct (via wcsset()) when wcsprm::timesys is changed.

18.15.2.33 trefpos `char wcsprm::trefpos`

(Given, auxiliary) **TREFPOS** keyvalue, being the location in space where the recorded time is valid.

It is not necessary to reset the **wcsprm** struct (via wcsset()) when wcsprm::trefpos is changed.

18.15.2.34 trefdir `char wcsprm::trefdir`

(Given, auxiliary) **TREFDIR** keyvalue, being the reference direction used in calculating a pathlength delay.

It is not necessary to reset the **wcsprm** struct (via wcsset()) when wcsprm::trefdir is changed.

18.15.2.35 plephem `char wcsprm::plephem`

(Given, auxiliary) **PLEPHEM** keyvalue, being the Solar System ephemeris used for calculating a pathlength delay.

It is not necessary to reset the **wcsprm** struct (via wcsset()) when wcsprm::plephem is changed.

18.15.2.36 timeunit `char wcsprm::timeunit`

(Given, auxiliary) **TIMEUNIT** keyvalue, being the time units in which the following header values are expressed: **TSTART**, **TSTOP**, **TIMEOFFS**, **TIMSYER**, **TIMRDER**, **TIMEDEL**. It also provides the default value for **CUNIT**ia for time axes.

It is not necessary to reset the **wcsprm** struct (via wcsset()) when wcsprm::timeunit is changed.

18.15.2.37 dateref `char wcsprm::dateref`

(Given, auxiliary) **DATEREF** keyvalue, being the date of a reference epoch relative to which other time measurements refer.

It is not necessary to reset the **wcsprm** struct (via wcsset()) when wcsprm::dateref is changed.

18.15.2.38 mjdref `double wcsprm::mjdref`

(Given, auxiliary) **MJDREF** keyvalue, equivalent to **DATEREF** expressed as a Modified Julian Date (MJD = JD - 2400000.5). The value is given as the sum of the two-element vector, allowing increased precision.

It is not necessary to reset the **wcsprm** struct (via wcsset()) when wcsprm::mjdref is changed.

18.15.2.39 timeoffs `double wcsprm::timeoffs`

(Given, auxiliary) **TIMEOFFS** keyvalue, being a time offset, which may be used, for example, to provide a uniform clock correction for times referenced to **DATEREF**.

It is not necessary to reset the **wcsprm** struct (via wcsset()) when wcsprm::timeoffs is changed.

18.15.2.40 dateobs `char wcsprm::dateobs`

(Given, auxiliary) **DATE−OBS** keyvalue, being the date at the start of the observation unless otherwise explained in the **DATE−OBS** keycomment, in ISO format, *yyyy-mm-dd*T*hh:mm:ss*.

It is not necessary to reset the **wcsprm** struct (via wcsset()) when wcsprm::dateobs is changed.

18.15.2.41 datebeg `char wcsprm::datebeg`

(Given, auxiliary) **DATE−BEG** keyvalue, being the date at the start of the observation in ISO format, *yyyy-mm-dd*T*hh:mm:ss*.

It is not necessary to reset the **wcsprm** struct (via wcsset()) when wcsprm::datebeg is changed.

18.15.2.42 dateavg `char wcsprm::dateavg`

(Given, auxiliary) **DATE−AVG** keyvalue, being the date at a representative mid-point of the observation in ISO format, *yyyy-mm-dd*T*hh:mm:ss*.

It is not necessary to reset the **wcsprm** struct (via wcsset()) when wcsprm::dateavg is changed.

18.15.2.43   dateend `char wcsprm::dateend`

(Given, auxiliary) **DATE−END** keyvalue, baing the date at the end of the observation in ISO format, *yyyy-mm-dd**T**hh:mm:ss*.

It is not necessary to reset the **wcsprm** struct (via wcsset()) when wcsprm::dateend is changed.

18.15.2.44   mjdobs `double wcsprm::mjdobs`

(Given, auxiliary) **MJD−OBS** keyvalue, equivalent to **DATE−OBS** expressed as a Modified Julian Date (MJD = JD - 2400000.5).

It is not necessary to reset the **wcsprm** struct (via wcsset()) when wcsprm::mjdobs is changed.

18.15.2.45   mjdbeg `double wcsprm::mjdbeg`

(Given, auxiliary) **MJD−BEG** keyvalue, equivalent to **DATE−BEG** expressed as a Modified Julian Date (MJD = JD - 2400000.5).

It is not necessary to reset the **wcsprm** struct (via wcsset()) when wcsprm::mjdbeg is changed.

18.15.2.46   mjdavg `double wcsprm::mjdavg`

(Given, auxiliary) **MJD−AVG** keyvalue, equivalent to **DATE−AVG** expressed as a Modified Julian Date (MJD = JD - 2400000.5).

It is not necessary to reset the **wcsprm** struct (via wcsset()) when wcsprm::mjdavg is changed.

18.15.2.47   mjdend `double wcsprm::mjdend`

(Given, auxiliary) **MJD−END** keyvalue, equivalent to **DATE−END** expressed as a Modified Julian Date (MJD = JD - 2400000.5).

It is not necessary to reset the **wcsprm** struct (via wcsset()) when wcsprm::mjdend is changed.

18.15.2.48   jepoch `double wcsprm::jepoch`

(Given, auxiliary) J**EPOCH** keyvalue, equivalent to **DATE−OBS** expressed as a Julian epoch.

It is not necessary to reset the **wcsprm** struct (via wcsset()) when wcsprm::jepoch is changed.

18.15.2.49   bepoch `double wcsprm::bepoch`

(Given, auxiliary) B**EPOCH** keyvalue, equivalent to **DATE−OBS** expressed as a Besselian epoch

It is not necessary to reset the **wcsprm** struct (via wcsset()) when wcsprm::bepoch is changed.

18.15.2.50 tstart `double wcsprm::tstart`

(Given, auxiliary) **TSTART** keyvalue, equivalent to **DATE−BEG** expressed as a time in units of **TIMEUNIT** relative to **DATEREF**+**TIMEOFFS**.

It is not necessary to reset the **wcsprm** struct (via wcsset()) when wcsprm::tstart is changed.

18.15.2.51 tstop `double wcsprm::tstop`

(Given, auxiliary) **TSTOP** keyvalue, equivalent to **DATE−END** expressed as a time in units of **TIMEUNIT** relative to **DATEREF**+**TIMEOFFS**.

It is not necessary to reset the **wcsprm** struct (via wcsset()) when wcsprm::tstop is changed.

18.15.2.52 xposure `double wcsprm::xposure`

(Given, auxiliary) **XPOSURE** keyvalue, being the effective exposure time in units of **TIMEUNIT**.

It is not necessary to reset the **wcsprm** struct (via wcsset()) when wcsprm::xposure is changed.

18.15.2.53 telapse `double wcsprm::telapse`

(Given, auxiliary) **TELAPSE** keyvalue, equivalent to the elapsed time between **DATE−BEG** and **DATE−END**, in units of **TIMEUNIT**.

It is not necessary to reset the **wcsprm** struct (via wcsset()) when wcsprm::telapse is changed.

18.15.2.54 timsyer `double wcsprm::timsyer`

(Given, auxiliary) **TIMSYER** keyvalue, being the absolute error of the time values, in units of **TIMEUNIT**.

It is not necessary to reset the **wcsprm** struct (via wcsset()) when wcsprm::timsyer is changed.

18.15.2.55 timrder `double wcsprm::timrder`

(Given, auxiliary) **TIMRDER** keyvalue, being the accuracy of time stamps relative to each other, in units of **TIMEUNIT**.

It is not necessary to reset the **wcsprm** struct (via wcsset()) when wcsprm::timrder is changed.

18.15.2.56 timedel `double wcsprm::timedel`

(Given, auxiliary) **TIMEDEL** keyvalue, being the resolution of the time stamps.

It is not necessary to reset the **wcsprm** struct (via wcsset()) when wcsprm::timedel is changed.

18.15.2.57 timepixr `double wcsprm::timepixr`

(Given, auxiliary) **TIMEPIXR** keyvalue, being the relative position of the time stamps in binned time intervals, a value between 0.0 and 1.0.

It is not necessary to reset the **wcsprm** struct (via wcsset()) when wcsprm::timepixr is changed.

18.15.2.58 obsgeo `double wcsprm::obsgeo`

(Given, auxiliary) Location of the observer in a standard terrestrial reference frame. The first three give ITRS Cartesian coordinates **OBSGEO−X** [m], **OBSGEO−Y** [m], **OBSGEO−Z** [m], and the second three give **OBSGEO−L** [deg], **OBSGEO−B** [deg], **OBSGEO−H** [m], which are related through a standard transformation.

It is not necessary to reset the **wcsprm** struct (via wcsset()) when wcsprm::obsgeo is changed.

18.15.2.59 obsorbit `char wcsprm::obsorbit`

(Given, auxiliary) **OBSORBIT** keyvalue, being the URI, URL, or name of an orbit ephemeris file giving spacecraft coordinates relating to **TREFPOS**.

It is not necessary to reset the **wcsprm** struct (via wcsset()) when wcsprm::obsorbit is changed.

18.15.2.60 radesys `char wcsprm::radesys`

(Given, auxiliary) The equatorial or ecliptic coordinate system type, **RADESYS**a.

It is not necessary to reset the **wcsprm** struct (via wcsset()) when wcsprm::radesys is changed.

18.15.2.61 equinox `double wcsprm::equinox`

(Given, auxiliary) The equinox associated with dynamical equatorial or ecliptic coordinate systems, **EQUINOX**a (or **EPOCH** in older headers). Not applicable to ICRS equatorial or ecliptic coordinates.

It is not necessary to reset the **wcsprm** struct (via wcsset()) when wcsprm::equinox is changed.

18.15.2.62 specsys `char wcsprm::specsys`

(Given, auxiliary) Spectral reference frame (standard of rest), **SPECSYS**a.

It is not necessary to reset the **wcsprm** struct (via wcsset()) when wcsprm::specsys is changed.

18.15.2.63 ssysobs `char wcsprm::ssysobs`

(Given, auxiliary) The spectral reference frame in which there is no differential variation in the spectral coordinate across the field-of-view, **SSYSOBS**a.

It is not necessary to reset the **wcsprm** struct (via wcsset()) when wcsprm::ssysobs is changed.

18.15.2.64 velosys `double wcsprm::velosys`

(Given, auxiliary) The relative radial velocity [m/s] between the observer and the selected standard of rest in the direction of the celestial reference coordinate, **VELOSYS**a.

It is not necessary to reset the **wcsprm** struct (via wcsset()) when wcsprm::velosys is changed.

18.15.2.65 zsource `double wcsprm::zsource`

(Given, auxiliary) The redshift, **ZSOURCE**a, of the source.

It is not necessary to reset the **wcsprm** struct (via wcsset()) when wcsprm::zsource is changed.

18.15.2.66 ssyssrc `char wcsprm::ssyssrc`

(Given, auxiliary) The spectral reference frame (standard of rest), **SSYSSRC**a, in which wcsprm::zsource was measured.

It is not necessary to reset the **wcsprm** struct (via wcsset()) when wcsprm::ssyssrc is changed.

18.15.2.67 velangl `double wcsprm::velangl`

(Given, auxiliary) The angle [deg] that should be used to decompose an observed velocity into radial and transverse components.

It is not necessary to reset the **wcsprm** struct (via wcsset()) when wcsprm::velangl is changed.

18.15.2.68 aux `struct auxprm * wcsprm::aux`

(Given, auxiliary) This struct holds auxiliary coordinate system information of a specialist nature. While these parameters may be widely recognized within particular fields of astronomy, they differ from the above auxiliary parameters in not being defined by any of the FITS WCS standards. Collecting them together in a separate struct that is allocated only when required helps to control bloat in the size of the **wcsprm** struct.

18.15.2.69 ntab `int wcsprm::ntab`

(*Given*) See wcsprm::tab.

18.15.2.70 nwtb `int wcsprm::nwtb`

(*Given*) See wcsprm::wtb.

18.15.2.71 tab `struct tabprm * wcsprm::tab`

(*Given*) Address of the first element of an array of ntab tabprm structs for which memory has been allocated. These are used to store tabular transformation parameters.

Although technically wcsprm::ntab and tab are "given", they will normally be set by invoking wcstab(), whether directly or indirectly.

The tabprm structs contain some members that must be supplied and others that are derived. The information to be supplied comes primarily from arrays stored in one or more FITS binary table extensions. These arrays, referred to here as "wcstab arrays", are themselves located by parameters stored in the FITS image header.

18.15.2.72 wtb `struct wtbarr * wcsprm::wtb`

(*Given*) Address of the first element of an array of nwtb wtbarr structs for which memory has been allocated. These are used in extracting wcstab arrays from a FITS binary table.

Although technically wcsprm::nwtb and wtb are "given", they will normally be set by invoking wcstab(), whether directly or indirectly.

18.15.2.73 lngtyp `char wcsprm::lngtyp`

(*Returned*) Four-character WCS celestial longitude and ...

18.15.2.74 lattyp `char wcsprm::lattyp`

(*Returned*) ... latitude axis types. e.g. "RA", "DEC", "GLON", "GLAT", etc. extracted from '**RA-**', '**DEC-**', '**GLON**', '**GLAT**', etc. in the first four characters of **CTYPE**ia but with trailing dashes removed. (Declared as char[8] for alignment reasons.)

18.15.2.75 lng `int wcsprm::lng`

(*Returned*) Index for the longitude coordinate, and ...

18.15.2.76 lat `int wcsprm::lat`

(*Returned*) ... index for the latitude coordinate, and ...

18.15.2.77 spec `int wcsprm::spec`

(*Returned*) ... index for the spectral coordinate in the imgcrd[][] and world[][] arrays in the API of wcsp2s(), wcss2p() and wcsmix().

These may also serve as indices into the pixcrd[][] array provided that the **PC**i_ja matrix does not transpose axes.

18.15.2.78 cubeface `int wcsprm::cubeface`

(*Returned*) Index into the pixcrd[][] array for the **CUBEFACE** axis. This is used for quadcube projections where the cube faces are stored on a separate axis (see wcs.h).

18.15.2.79 types `int * wcsprm::types`

(*Returned*) Address of the first element of an array of int containing a four-digit type code for each axis.

- First digit (i.e. 1000s):
    - 0: Non-specific coordinate type.
    - 1: Stokes coordinate.
    - 2: Celestial coordinate (including **CUBEFACE**).
    - 3: Spectral coordinate.
    - 4: Time coordinate.

- Second digit (i.e. 100s):
    - 0: Linear axis.
    - 1: Quantized axis (**STOKES**, **CUBEFACE**).
    - 2: Non-linear celestial axis.
    - 3: Non-linear spectral axis.
    - 4: Logarithmic axis.
    - 5: Tabular axis.

- Third digit (i.e. 10s):
    - 0: Group number, e.g. lookup table number, being an index into the tabprm array (see above).

- The fourth digit is used as a qualifier depending on the axis type.
    - For celestial axes:
        * 0: Longitude coordinate.
        * 1: Latitude coordinate.
        * 2: **CUBEFACE** number.
    - For lookup tables: the axis number in a multidimensional table.

CTYPE`ia` in "4-3" form with unrecognized algorithm code will have its type set to -1 and generate an error.

18.15.2.80 lin `struct `linprm` wcsprm::lin`

(*Returned*) Linear transformation parameters (usage is described in the prologue to lin.h).

18.15.2.81 cel `struct `celprm` wcsprm::cel`

(*Returned*) Celestial transformation parameters (usage is described in the prologue to cel.h).

18.15.2.82 spc `struct `spcprm` wcsprm::spc`

(*Returned*) Spectral transformation parameters (usage is described in the prologue to spc.h).

18.15.2.83 err `struct `wcserr` * wcsprm::err`

(*Returned*) If enabled, when an error status is returned, this struct contains detailed information about the error, see wcserr_enable().

18.15.2.84 m_flag `int wcsprm::m_flag`

(For internal use only.)

18.15.2.85 m_naxis `int wcsprm::m_naxis`

(For internal use only.)

18.15.2.86 m_crpix `double * wcsprm::m_crpix`

(For internal use only.)

18.15.2.87 m_pc `double * wcsprm::m_pc`

(For internal use only.)

18.15.2.88 m_cdelt `double * wcsprm::m_cdelt`

(For internal use only.)

18.15.2.89 m_crval `double * wcsprm::m_crval`

(For internal use only.)

18.15.2.90 m_cunit `wcsprm::m_cunit`

(For internal use only.)

18.15.2.91 m_ctype `wcsprm::m_ctype`

(For internal use only.)

18.15.2.92 m_pv `struct pvcard * wcsprm::m_pv`

(For internal use only.)

18.15.2.93 m_ps `struct pscard * wcsprm::m_ps`

(For internal use only.)

18.15.2.94 m_cd `double * wcsprm::m_cd`

(For internal use only.)

18.15.2.95 m_crota `double * wcsprm::m_crota`

(For internal use only.)

18.15.2.96 m_colax `int * wcsprm::m_colax`

(For internal use only.)

18.15.2.97 m_cname `wcsprm::m_cname`

(For internal use only.)

18.15.2.98 m_crder `double * wcsprm::m_crder`

(For internal use only.)

18.15.2.99 m_csyer `double * wcsprm::m_csyer`

(For internal use only.)

18.15.2.100 m_czphs `double * wcsprm::m_czphs`

(For internal use only.)

18.15.2.101 m_cperi `double * wcsprm::m_cperi`

(For internal use only.)

18.15.2.102 m_aux `struct auxprm* wcsprm::m_aux`

18.15.2.103 m_tab `struct tabprm * wcsprm::m_tab`

(For internal use only.)

18.15.2.104 m_wtb `struct wtbarr * wcsprm::m_wtb`

(For internal use only.)

18.16 wtbarr Struct Reference

Extraction of coordinate lookup tables from BINTABLE.

`#include <getwcstab.h>`

Data Fields

- int i
- int m
- int kind
- char extnam [72]
- int extver
- int extlev
- char ttype [72]
- long row
- int ndim
- int ∗ dimlen
- double ∗∗ arrayp

18.16.1 Detailed Description

Function wcstab(), which is invoked automatically by wcspih(), sets up an array of **wtbarr** structs to assist in extracting coordinate lookup tables from a binary table extension (BINTABLE) and copying them into the tabprm structs stored in wcsprm. Refer to the usage notes for wcspih() and wcstab() in wcshdr.h, and also the prologue to tab.h.

For C++ usage, because of a name space conflict with the **wtbarr** typedef defined in CFITSIO header fitsio.h, the **wtbarr** struct is renamed to **wtbarr_s** by preprocessor macro substitution with scope limited to **wtbarr.h** itself, and similarly in wcs.h.

18.16.2 Field Documentation

18.16.2.1 i `int wtbarr::i`

(*Given*) Image axis number.

18.16.2.2 m `int wtbarr::m`

(*Given*) wcstab array axis number for index vectors.

18.16.2.3 kind `int wtbarr::kind`

(*Given*) Character identifying the wcstab array type:

- c: coordinate array,

- i: index vector.

18.16.2.4 extnam `char wtbarr::extnam`

(*Given*) **EXTNAME** identifying the binary table extension.

18.16.2.5 extver `int wtbarr::extver`

(*Given*) **EXTVER** identifying the binary table extension.

18.16.2.6 extlev `int wtbarr::extlev`

(*Given*) **EXTLEV** identifying the binary table extension.

18.16.2.7 ttype `char wtbarr::ttype`

(*Given*) **TTYPE**n identifying the column of the binary table that contains the wcstab array.

18.16.2.8 row `long wtbarr::row`

(*Given*) Table row number.

18.16.2.9 ndim `int wtbarr::ndim`

(*Given*) Expected dimensionality of the wcstab array.

18.16.2.10 dimlen `int * wtbarr::dimlen`

(*Given*) Address of the first element of an array of int of length ndim into which the wcstab array axis lengths are to be written.

18.16.2.11 arrayp `double ** wtbarr::arrayp`

(*Given*) Pointer to an array of double which is to be allocated by the user and into which the wcstab array is to be written.

19 File Documentation

19.1 cel.h File Reference

`#include "prj.h"`

Data Structures

- struct celprm
    Celestial transformation parameters.

Macros

- #define CELLEN (sizeof(struct celprm)/sizeof(int))

    Size of the celprm struct in int units.
- #define celini_errmsg cel_errmsg

    Deprecated.
- #define celprt_errmsg cel_errmsg

    Deprecated.
- #define celset_errmsg cel_errmsg

    Deprecated.
- #define celx2s_errmsg cel_errmsg

    Deprecated.
- #define cels2x_errmsg cel_errmsg

    Deprecated.

Enumerations

- enum cel_errmsg_enum {
  CELERR_SUCCESS = 0 , CELERR_NULL_POINTER = 1 , CELERR_BAD_PARAM = 2 , CELERR_BAD_COORD_TRANS
  = 3 ,
  CELERR_ILL_COORD_TRANS = 4 , CELERR_BAD_PIX = 5 , CELERR_BAD_WORLD = 6 }

Functions

- int celini (struct celprm ∗cel)

    Default constructor for the celprm struct.
- int celfree (struct celprm ∗cel)

    Destructor for the celprm struct.
- int celsize (const struct celprm ∗cel, int sizes[2])

    Compute the size of a celprm struct.
- int celprt (const struct celprm ∗cel)

    Print routine for the celprm struct.
- int celperr (const struct celprm ∗cel, const char ∗prefix)

    Print error messages from a celprm struct.
- int celset (struct celprm ∗cel)

    Setup routine for the celprm struct.
- int celx2s (struct celprm ∗cel, int nx, int ny, int sxy, int sll, const double x[ ], const double y[ ], double phi[ ],
  double theta[ ], double lng[ ], double lat[ ], int stat[ ])

    Pixel-to-world celestial transformation.
- int cels2x (struct celprm ∗cel, int nlng, int nlat, int sll, int sxy, const double lng[ ], const double lat[ ], double
  phi[ ], double theta[ ], double x[ ], double y[ ], int stat[ ])

    World-to-pixel celestial transformation.

Variables

- const char ∗ cel_errmsg [ ]

19.1.1 Detailed Description

Routines in this suite implement the part of the FITS World Coordinate System (WCS) standard that deals with celestial coordinates, as described in

```
"Representations of world coordinates in FITS",
Greisen, E.W., & Calabretta, M.R. 2002, A&A, 395, 1061 (WCS Paper I)
"Representations of celestial coordinates in FITS",
Calabretta, M.R., & Greisen, E.W. 2002, A&A, 395, 1077 (WCS Paper II)
```

These routines define methods to be used for computing celestial world coordinates from intermediate world coordinates (a linear transformation of image pixel coordinates), and vice versa. They are based on the celprm struct which contains all information needed for the computations. This struct contains some elements that must be set by the user, and others that are maintained by these routines, somewhat like a C++ class but with no encapsulation.

Routine celini() is provided to initialize the celprm struct with default values, celfree() reclaims any memory that may have been allocated to store an error message, celsize() computes its total size including allocated memory, and celprt() prints its contents.

celerr() prints the error message(s), if any, stored in a celprm struct and the prjprm struct that it contains.

A setup routine, celset(), computes intermediate values in the celprm struct from parameters in it that were supplied by the user. The struct always needs to be set up by celset() but it need not be called explicitly - refer to the explanation of celprm::flag.

celx2s() and cels2x() implement the WCS celestial coordinate transformations. In fact, they are high level driver routines for the lower level spherical coordinate rotation and projection routines described in sph.h and prj.h.

19.1.2 Macro Definition Documentation

19.1.2.1 CELLEN `#define CELLEN (sizeof(struct celprm)/sizeof(int))`

Size of the celprm struct in *int* units, used by the Fortran wrappers.

19.1.2.2 celini_errmsg `#define celini_errmsg cel_errmsg`

Deprecated Added for backwards compatibility, use cel_errmsg directly now instead.

19.1.2.3 celprt_errmsg `#define celprt_errmsg cel_errmsg`

Deprecated Added for backwards compatibility, use cel_errmsg directly now instead.

19.1.2.4 celset_errmsg `#define celset_errmsg cel_errmsg`

Deprecated Added for backwards compatibility, use cel_errmsg directly now instead.

19.1.2.5 celx2s_errmsg `#define celx2s_errmsg cel_errmsg`

Deprecated Added for backwards compatibility, use cel_errmsg directly now instead.

19.1.2.6 cels2x_errmsg `#define cels2x_errmsg cel_errmsg`

Deprecated Added for backwards compatibility, use cel_errmsg directly now instead.

19.1.3 Enumeration Type Documentation

19.1.3.1 cel_errmsg_enum `enum cel_errmsg_enum`

Enumerator

CELERR_SUCCESS	
CELERR_NULL_POINTER	
CELERR_BAD_PARAM	
CELERR_BAD_COORD_TRANS	
CELERR_ILL_COORD_TRANS	
CELERR_BAD_PIX	
CELERR_BAD_WORLD	

19.1.4 Function Documentation

19.1.4.1 celini() `int celini (`
            `struct celprm * cel )`

celini() sets all members of a celprm struct to default values. It should be used to initialize every celprm struct.

PLEASE NOTE: If the celprm struct has already been initialized, then before reinitializing, it celfree() should be used to free any memory that may have been allocated to store an error message. A memory leak may otherwise result.

Parameters

out	*cel*	Celestial transformation parameters.

Returns

Status return value:

- 0: Success.
- 1: Null celprm pointer passed.

19.1.4.2 celfree() `int celfree (`
`            struct celprm * cel )`

celfree() frees any memory that may have been allocated to store an error message in the celprm struct.

Parameters

in	*cel*	Celestial transformation parameters.

Returns

Status return value:

- 0: Success.
- 1: Null celprm pointer passed.

19.1.4.3 celsize() `int celsize (`
`            const struct celprm * cel,`
`            int sizes[2] )`

celsize() computes the full size of a celprm struct, including allocated memory.

Parameters

in	*cel*	Celestial transformation parameters. If NULL, the base size of the struct and the allocated size are both set to zero.
out	*sizes*	The first element is the base size of the struct as returned by sizeof(struct celprm). The second element is the total allocated size, in bytes. This figure includes memory allocated for the constituent struct, celprm::err. It is not an error for the struct not to have been set up via celset().

Returns

Status return value:

- 0: Success.

19.1.4.4   celprt() `int celprt (`
          `const struct celprm * cel )`

celprt() prints the contents of a celprm struct using wcsprintf(). Mainly intended for diagnostic purposes.

Parameters

in	*cel*	Celestial transformation parameters.

Returns

> Status return value:

- 0: Success.
- 1: Null celprm pointer passed.

19.1.4.5   celperr() `int celperr (`
          `const struct celprm * cel,`
          `const char * prefix )`

celperr() prints the error message(s), if any, stored in a celprm struct and the prjprm struct that it contains. If there are no errors then nothing is printed. It uses wcserr_prt(), q.v.

Parameters

in	*cel*	Coordinate transformation parameters.
in	*prefix*	If non-NULL, each output line will be prefixed with this string.

Returns

> Status return value:

- 0: Success.
- 1: Null celprm pointer passed.

19.1.4.6   celset() `int celset (`
          `struct celprm * cel )`

celset() sets up a celprm struct according to information supplied within it.

Note that this routine need not be called directly; it will be invoked by celx2s() and cels2x() if celprm::flag is anything other than a predefined magic value.

Parameters

in,out	*cel*	Celestial transformation parameters.

Returns

Status return value:

- 0: Success.
- 1: Null celprm pointer passed.
- 2: Invalid projection parameters.
- 3: Invalid coordinate transformation parameters.
- 4: Ill-conditioned coordinate transformation parameters.

For returns > 1, a detailed error message is set in celprm::err if enabled, see wcserr_enable().

19.1.4.7 celx2s() `int celx2s (`

```
struct celprm * cel,
int nx,
int ny,
int sxy,
int sll,
const double x[],
const double y[],
double phi[],
double theta[],
double lng[],
double lat[],
int stat[] )
```

celx2s() transforms (x, y) coordinates in the plane of projection to celestial coordinates (α, δ).

Parameters

in,out	*cel*	Celestial transformation parameters.
in	*nx,ny*	Vector lengths.
in	*sxy,sll*	Vector strides.
in	*x,y*	Projected coordinates in pseudo "degrees".
out	*phi,theta*	Longitude and latitude (ϕ, θ) in the native coordinate system of the projection [deg].
out	*lng,lat*	Celestial longitude and latitude (α, δ) of the projected point [deg].
out	*stat*	Status return value for each vector element: • 0: Success. • 1: Invalid value of (x, y).

Returns

Status return value:

- 0: Success.

- 1: Null celprm pointer passed.

- 2: Invalid projection parameters.

- 3: Invalid coordinate transformation parameters.

- 4: Ill-conditioned coordinate transformation parameters.

- 5: One or more of the (x, y) coordinates were invalid, as indicated by the stat vector.

For returns > 1, a detailed error message is set in celprm::err if enabled, see wcserr_enable().

19.1.4.8 cels2x() int cels2x (
        struct celprm * *cel,*
        int *nlng,*
        int *nlat,*
        int *sll,*
        int *sxy,*
        const double *lng[ ],*
        const double *lat[ ],*
        double *phi[ ],*
        double *theta[ ],*
        double *x[ ],*
        double *y[ ],*
        int *stat[ ]*)

cels2x() transforms celestial coordinates (α, δ) to (x, y) coordinates in the plane of projection.

Parameters

in,out	*cel*	Celestial transformation parameters.
in	*nlng,nlat*	Vector lengths.
in	*sll,sxy*	Vector strides.
in	*lng,lat*	Celestial longitude and latitude (α, δ) of the projected point [deg].
out	*phi,theta*	Longitude and latitude (ϕ, θ) in the native coordinate system of the projection [deg].
out	*x,y*	Projected coordinates in pseudo "degrees".

out	*stat*	Status return value for each vector element:
		
		    • 0: Success.
		
		    • 1: Invalid value of (α, δ).

Returns

Status return value:

- 0: Success.
- 1: Null celprm pointer passed.
- 2: Invalid projection parameters.
- 3: Invalid coordinate transformation parameters.
- 4: Ill-conditioned coordinate transformation parameters.
- 6: One or more of the (α, δ) coordinates were invalid, as indicated by the stat vector.

For returns > 1, a detailed error message is set in celprm::err if enabled, see wcserr_enable().

19.1.5 Variable Documentation

19.1.5.1 cel_errmsg `const char* cel_errmsg[] [extern]`

19.2 cel.h

[Go to the documentation of this file.](#)

```
1  /*============================================================================
2    WCSLIB 7.11 - an implementation of the FITS WCS standard.
3    Copyright (C) 1995-2022, Mark Calabretta
4
5    This file is part of WCSLIB.
6
7    WCSLIB is free software: you can redistribute it and/or modify it under the
8    terms of the GNU Lesser General Public License as published by the Free
9    Software Foundation, either version 3 of the License, or (at your option)
10   any later version.
11
12   WCSLIB is distributed in the hope that it will be useful, but WITHOUT ANY
13   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
14   FOR A PARTICULAR PURPOSE.  See the GNU Lesser General Public License for
15   more details.
16
17   You should have received a copy of the GNU Lesser General Public License
18   along with WCSLIB.  If not, see http://www.gnu.org/licenses.
19
20   Author: Mark Calabretta, Australia Telescope National Facility, CSIRO.
21   http://www.atnf.csiro.au/people/Mark.Calabretta
22   $Id: cel.h,v 7.11 2022/04/26 06:13:52 mcalabre Exp $
23  *============================================================================
24  *
25  * WCSLIB 7.11 - C routines that implement the FITS World Coordinate System
26  * (WCS) standard.  Refer to the README file provided with WCSLIB for an
27  * overview of the library.
28  *
29  *
30  * Summary of the cel routines
31  * ---------------------------
32  * Routines in this suite implement the part of the FITS World Coordinate
33  * System (WCS) standard that deals with celestial coordinates, as described in
34  *
35  =   "Representations of world coordinates in FITS",
36  =   Greisen, E.W., & Calabretta, M.R. 2002, A&A, 395, 1061 (WCS Paper I)
37  =
38  =   "Representations of celestial coordinates in FITS",
39  =   Calabretta, M.R., & Greisen, E.W. 2002, A&A, 395, 1077 (WCS Paper II)
40  *
41  * These routines define methods to be used for computing celestial world
42  * coordinates from intermediate world coordinates (a linear transformation
43  * of image pixel coordinates), and vice versa.  They are based on the celprm
44  * struct which contains all information needed for the computations.  This
45  * struct contains some elements that must be set by the user, and others that
46  * are maintained by these routines, somewhat like a C++ class but with no
47  * encapsulation.
48  *
49  * Routine celini() is provided to initialize the celprm struct with default
50  * values, celfree() reclaims any memory that may have been allocated to store
51  * an error message, celsize() computes its total size including allocated
52  * memory, and celprt() prints its contents.
53  *
54  * celperr() prints the error message(s), if any, stored in a celprm struct and
55  * the prjprm struct that it contains.
56  *
57  * A setup routine, celset(), computes intermediate values in the celprm struct
58  * from parameters in it that were supplied by the user.  The struct always
59  * needs to be set up by celset() but it need not be called explicitly - refer
60  * to the explanation of celprm::flag.
61  *
62  * celx2s() and cels2x() implement the WCS celestial coordinate
63  * transformations.  In fact, they are high level driver routines for the lower
64  * level spherical coordinate rotation and projection routines described in
65  * sph.h and prj.h.
66  *
67  *
68  * celini() - Default constructor for the celprm struct
69  * ----------------------------------------------------
```

```
70 * celini() sets all members of a celprm struct to default values.  It should
71 * be used to initialize every celprm struct.
72 *
73 * PLEASE NOTE: If the celprm struct has already been initialized, then before
74 * reinitializing, it celfree() should be used to free any memory that may have
75 * been allocated to store an error message.  A memory leak may otherwise
76 * result.
77 *
78 * Returned:
79 *   cel       struct celprm*
80 *                       Celestial transformation parameters.
81 *
82 * Function return value:
83 *             int       Status return value:
84 *                         0: Success.
85 *                         1: Null celprm pointer passed.
86 *
87 *
88 * celfree() - Destructor for the celprm struct
89 * ---------------------------------------------
90 * celfree() frees any memory that may have been allocated to store an error
91 * message in the celprm struct.
92 *
93 * Given:
94 *   cel       struct celprm*
95 *                       Celestial transformation parameters.
96 *
97 * Function return value:
98 *             int       Status return value:
99 *                         0: Success.
100 *                         1: Null celprm pointer passed.
101 *
102 *
103 * celsize() - Compute the size of a celprm struct
104 * -----------------------------------------------
105 * celsize() computes the full size of a celprm struct, including allocated
106 * memory.
107 *
108 * Given:
109 *   cel       const struct celprm*
110 *                       Celestial transformation parameters.
111 *
112 *                       If NULL, the base size of the struct and the allocated
113 *                       size are both set to zero.
114 *
115 * Returned:
116 *   sizes     int[2]    The first element is the base size of the struct as
117 *                       returned by sizeof(struct celprm).  The second element
118 *                       is the total allocated size, in bytes.  This figure
119 *                       includes memory allocated for the constituent struct,
120 *                       celprm::err.
121 *
122 *                       It is not an error for the struct not to have been set
123 *                       up via celset().
124 *
125 * Function return value:
126 *             int       Status return value:
127 *                         0: Success.
128 *
129 *
130 * celprt() - Print routine for the celprm struct
131 * ----------------------------------------------
132 * celprt() prints the contents of a celprm struct using wcsprintf().  Mainly
133 * intended for diagnostic purposes.
134 *
135 * Given:
136 *   cel       const struct celprm*
137 *                       Celestial transformation parameters.
138 *
139 * Function return value:
140 *             int       Status return value:
141 *                         0: Success.
142 *                         1: Null celprm pointer passed.
143 *
144 *
145 * celperr() - Print error messages from a celprm struct
146 * -----------------------------------------------------
147 * celperr() prints the error message(s), if any, stored in a celprm struct and
148 * the prjprm struct that it contains.  If there are no errors then nothing is
149 * printed.  It uses wcserr_prt(), q.v.
150 *
151 * Given:
152 *   cel       const struct celprm*
153 *                       Coordinate transformation parameters.
154 *
155 *   prefix    const char *
156 *                       If non-NULL, each output line will be prefixed with
```

```
157 *                      this string.
158 *
159 * Function return value:
160 *            int       Status return value:
161 *                        0: Success.
162 *                        1: Null celprm pointer passed.
163 *
164 *
165 * celset() - Setup routine for the celprm struct
166 * ----------------------------------------------
167 * celset() sets up a celprm struct according to information supplied within
168 * it.
169 *
170 * Note that this routine need not be called directly; it will be invoked by
171 * celx2s() and cels2x() if celprm::flag is anything other than a predefined
172 * magic value.
173 *
174 * Given and returned:
175 *   cel       struct celprm*
176 *                      Celestial transformation parameters.
177 *
178 * Function return value:
179 *            int       Status return value:
180 *                        0: Success.
181 *                        1: Null celprm pointer passed.
182 *                        2: Invalid projection parameters.
183 *                        3: Invalid coordinate transformation parameters.
184 *                        4: Ill-conditioned coordinate transformation
185 *                           parameters.
186 *
187 *                      For returns > 1, a detailed error message is set in
188 *                      celprm::err if enabled, see wcserr_enable().
189 *
190 *
191 * celx2s() - Pixel-to-world celestial transformation
192 * --------------------------------------------------
193 * celx2s() transforms (x,y) coordinates in the plane of projection to
194 * celestial coordinates (lng,lat).
195 *
196 * Given and returned:
197 *   cel       struct celprm*
198 *                      Celestial transformation parameters.
199 *
200 * Given:
201 *   nx,ny     int       Vector lengths.
202 *
203 *   sxy,sll   int       Vector strides.
204 *
205 *   x,y       const double[]
206 *                      Projected coordinates in pseudo "degrees".
207 *
208 * Returned:
209 *   phi,theta double[]  Longitude and latitude (phi,theta) in the native
210 *                      coordinate system of the projection [deg].
211 *
212 *   lng,lat   double[]  Celestial longitude and latitude (lng,lat) of the
213 *                      projected point [deg].
214 *
215 *   stat      int[]     Status return value for each vector element:
216 *                        0: Success.
217 *                        1: Invalid value of (x,y).
218 *
219 * Function return value:
220 *            int       Status return value:
221 *                        0: Success.
222 *                        1: Null celprm pointer passed.
223 *                        2: Invalid projection parameters.
224 *                        3: Invalid coordinate transformation parameters.
225 *                        4: Ill-conditioned coordinate transformation
226 *                           parameters.
227 *                        5: One or more of the (x,y) coordinates were
228 *                           invalid, as indicated by the stat vector.
229 *
230 *                      For returns > 1, a detailed error message is set in
231 *                      celprm::err if enabled, see wcserr_enable().
232 *
233 *
234 * cels2x() - World-to-pixel celestial transformation
235 * --------------------------------------------------
236 * cels2x() transforms celestial coordinates (lng,lat) to (x,y) coordinates in
237 * the plane of projection.
238 *
239 * Given and returned:
240 *   cel       struct celprm*
241 *                      Celestial transformation parameters.
242 *
243 * Given:
```

```
244 *   nlng,nlat int      Vector lengths.
245 *
246 *   sll,sxy   int      Vector strides.
247 *
248 *   lng,lat   const double[]
249 *                      Celestial longitude and latitude (lng,lat) of the
250 *                      projected point [deg].
251 *
252 * Returned:
253 *   phi,theta double[]  Longitude and latitude (phi,theta) in the native
254 *                      coordinate system of the projection [deg].
255 *
256 *   x,y       double[]  Projected coordinates in pseudo "degrees".
257 *
258 *   stat      int[]     Status return value for each vector element:
259 *                          0: Success.
260 *                          1: Invalid value of (lng,lat).
261 *
262 * Function return value:
263 *             int       Status return value:
264 *                          0: Success.
265 *                          1: Null celprm pointer passed.
266 *                          2: Invalid projection parameters.
267 *                          3: Invalid coordinate transformation parameters.
268 *                          4: Ill-conditioned coordinate transformation
269 *                             parameters.
270 *                          6: One or more of the (lng,lat) coordinates were
271 *                             invalid, as indicated by the stat vector.
272 *
273 *                      For returns > 1, a detailed error message is set in
274 *                      celprm::err if enabled, see wcserr_enable().
275 *
276 *
277 * celprm struct - Celestial transformation parameters
278 * ---------------------------------------------------
279 * The celprm struct contains information required to transform celestial
280 * coordinates.  It consists of certain members that must be set by the user
281 * ("given") and others that are set by the WCSLIB routines ("returned").  Some
282 * of the latter are supplied for informational purposes and others are for
283 * internal use only.
284 *
285 * Returned celprm struct members must not be modified by the user.
286 *
287 *   int flag
288 *     (Given and returned) This flag must be set to zero whenever any of the
289 *     following celprm struct members are set or changed:
290 *
291 *       - celprm::offset,
292 *       - celprm::phi0,
293 *       - celprm::theta0,
294 *       - celprm::ref[4],
295 *       - celprm::prj:
296 *         - prjprm::code,
297 *         - prjprm::r0,
298 *         - prjprm::pv[],
299 *         - prjprm::phi0,
300 *         - prjprm::theta0.
301 *
302 *     This signals the initialization routine, celset(), to recompute the
303 *     returned members of the celprm struct.  celset() will reset flag to
304 *     indicate that this has been done.
305 *
306 *   int offset
307 *     (Given) If true (non-zero), an offset will be applied to (x,y) to
308 *     force (x,y) = (0,0) at the fiducial point, (phi_0,theta_0).
309 *     Default is 0 (false).
310 *
311 *   double phi0
312 *     (Given) The native longitude, phi_0 [deg], and ...
313 *
314 *   double theta0
315 *     (Given) ... the native latitude, theta_0 [deg], of the fiducial point,
316 *     i.e. the point whose celestial coordinates are given in
317 *     celprm::ref[1:2].  If undefined (set to a magic value by prjini()) the
318 *     initialization routine, celset(), will set this to a projection-specific
319 *     default.
320 *
321 *   double ref[4]
322 *     (Given) The first pair of values should be set to the celestial
323 *     longitude and latitude of the fiducial point [deg] - typically right
324 *     ascension and declination.  These are given by the CRVALia keywords in
325 *     FITS.
326 *
327 *     (Given and returned) The second pair of values are the native longitude,
328 *     phi_p [deg], and latitude, theta_p [deg], of the celestial pole (the
329 *     latter is the same as the celestial latitude of the native pole,
330 *     delta_p) and these are given by the FITS keywords LONPOLEa and LATPOLEa
```

```
331 *      (or by PVi_2a and PVi_3a attached to the longitude axis which take
332 *      precedence if defined).
333 *
334 *      LONPOLEa defaults to phi_0 (see above) if the celestial latitude of the
335 *      fiducial point of the projection is greater than or equal to the native
336 *      latitude, otherwise phi_0 + 180 [deg].  (This is the condition for the
337 *      celestial latitude to increase in the same direction as the native
338 *      latitude at the fiducial point.)  ref[2] may be set to UNDEFINED (from
339 *      wcsmath.h) or 999.0 to indicate that the correct default should be
340 *      substituted.
341 *
342 *      theta_p, the native latitude of the celestial pole (or equally the
343 *      celestial latitude of the native pole, delta_p) is often determined
344 *      uniquely by CRVALia and LONPOLEa in which case LATPOLEa is ignored.
345 *      However, in some circumstances there are two valid solutions for theta_p
346 *      and LATPOLEa is used to choose between them.  LATPOLEa is set in ref[3]
347 *      and the solution closest to this value is used to reset ref[3].  It is
348 *      therefore legitimate, for example, to set ref[3] to +90.0 to choose the
349 *      more northerly solution - the default if the LATPOLEa keyword is omitted
350 *      from the FITS header.  For the special case where the fiducial point of
351 *      the projection is at native latitude zero, its celestial latitude is
352 *      zero, and LONPOLEa = +/- 90.0 then the celestial latitude of the native
353 *      pole is not determined by the first three reference values and LATPOLEa
354 *      specifies it completely.
355 *
356 *      The returned value, celprm::latpreq, specifies how LATPOLEa was actually
357 *      used.
358 *
359 *   struct prjprm prj
360 *      (Given and returned) Projection parameters described in the prologue to
361 *      prj.h.
362 *
363 *   double euler[5]
364 *      (Returned) Euler angles and associated intermediaries derived from the
365 *      coordinate reference values.  The first three values are the Z-, X-, and
366 *      Z'-Euler angles [deg], and the remaining two are the cosine and sine of
367 *      the X-Euler angle.
368 *
369 *   int latpreq
370 *      (Returned) For informational purposes, this indicates how the LATPOLEa
371 *      keyword was used
372 *        - 0: Not required, theta_p (== delta_p) was determined uniquely by the
373 *             CRVALia and LONPOLEa keywords.
374 *        - 1: Required to select between two valid solutions of theta_p.
375 *        - 2: theta_p was specified solely by LATPOLEa.
376 *
377 *   int isolat
378 *      (Returned) True if the spherical rotation preserves the magnitude of the
379 *      latitude, which occurs iff the axes of the native and celestial
380 *      coordinates are coincident.  It signals an opportunity to cache
381 *      intermediate calculations common to all elements in a vector
382 *      computation.
383 *
384 *   struct wcserr *err
385 *      (Returned) If enabled, when an error status is returned, this struct
386 *      contains detailed information about the error, see wcserr_enable().
387 *
388 *   void *padding
389 *      (An unused variable inserted for alignment purposes only.)
390 *
391 * Global variable: const char *cel_errmsg[] - Status return messages
392 * -----------------------------------------------------------------
393 * Status messages to match the status value returned from each function.
394 *
395 *===========================================================================*/
396
397 #ifndef WCSLIB_CEL
398 #define WCSLIB_CEL
399
400 #include "prj.h"
401
402 #ifdef __cplusplus
403 extern "C" {
404 #endif
405
406
407 extern const char *cel_errmsg[];
408
409 enum cel_errmsg_enum {
410   CELERR_SUCCESS         = 0,     // Success.
411   CELERR_NULL_POINTER    = 1,          // Null celprm pointer passed.
412   CELERR_BAD_PARAM       = 2,     // Invalid projection parameters.
413   CELERR_BAD_COORD_TRANS = 3,      // Invalid coordinate transformation
414                                    // parameters.
415   CELERR_ILL_COORD_TRANS = 4,      // Ill-conditioned coordinated transformation
416                                    // parameters.
417   CELERR_BAD_PIX         = 5,      // One or more of the (x,y) coordinates were
```

```
418                             // invalid.
419    CELERR_BAD_WORLD       = 6    // One or more of the (lng,lat) coordinates
420                             // were invalid.
421 };
422
423 struct celprm {
424   // Initialization flag (see the prologue above).
425   //------------------------------------------------------------------------
426   int    flag;                    // Set to zero to force initialization.
427
428   // Parameters to be provided (see the prologue above).
429   //------------------------------------------------------------------------
430   int    offset;          // Force (x,y) = (0,0) at (phi_0,theta_0).
431   double phi0, theta0;                // Native coordinates of fiducial point.
432   double ref[4];              // Celestial coordinates of fiducial
433                                 // point and native coordinates of
434                                 // celestial pole.
435
436   struct prjprm prj;          // Projection parameters (see prj.h).
437
438   // Information derived from the parameters supplied.
439   //------------------------------------------------------------------------
440   double euler[5];          // Euler angles and functions thereof.
441   int    latpreq;                 // LATPOLEa requirement.
442   int    isolat;          // True if |latitude| is preserved.
443
444   // Error handling
445   //------------------------------------------------------------------------
446   struct wcserr *err;
447
448   // Private
449   //------------------------------------------------------------------------
450   void   *padding;                 // (Dummy inserted for alignment purposes.)
451 };
452
453 // Size of the celprm struct in int units, used by the Fortran wrappers.
454 #define CELLEN (sizeof(struct celprm)/sizeof(int))
455
456
457 int celini(struct celprm *cel);
458
459 int celfree(struct celprm *cel);
460
461 int celsize(const struct celprm *cel, int sizes[2]);
462
463 int celprt(const struct celprm *cel);
464
465 int celperr(const struct celprm *cel, const char *prefix);
466
467 int celset(struct celprm *cel);
468
469 int celx2s(struct celprm *cel, int nx, int ny, int sxy, int sll,
470           const double x[], const double y[],
471           double phi[], double theta[], double lng[], double lat[],
472           int stat[]);
473
474 int cels2x(struct celprm *cel, int nlng, int nlat, int sll, int sxy,
475           const double lng[], const double lat[],
476           double phi[], double theta[], double x[], double y[],
477           int stat[]);
478
479
480 // Deprecated.
481 #define celini_errmsg cel_errmsg
482 #define celprt_errmsg cel_errmsg
483 #define celset_errmsg cel_errmsg
484 #define celx2s_errmsg cel_errmsg
485 #define cels2x_errmsg cel_errmsg
486
487 #ifdef __cplusplus
488 }
489 #endif
490
491 #endif // WCSLIB_CEL
```

19.3   dis.h File Reference

Data Structures

- struct dpkey

     Store for DP ja and DQ ia keyvalues.

- struct disprm

  Distortion parameters.

Macros

- #define DISP2X_ARGS
- #define DISX2P_ARGS
- #define DPLEN (sizeof(struct dpkey)/sizeof(int))
- #define DISLEN (sizeof(struct disprm)/sizeof(int))

Enumerations

- enum dis_errmsg_enum {
  DISERR_SUCCESS = 0 , DISERR_NULL_POINTER = 1 , DISERR_MEMORY = 2 , DISERR_BAD_PARAM
  = 3 ,
  DISERR_DISTORT = 4 , DISERR_DEDISTORT = 5 }

Functions

- int disndp (int n)

  Memory allocation for $DPja$ and $DQia$.
- int dpfill (struct dpkey ∗dp, const char ∗keyword, const char ∗field, int j, int type, int i, double f)

  Fill the contents of a dpkey struct.
- int dpkeyi (const struct dpkey ∗dp)

  Get the data value in a dpkey struct as int.
- double dpkeyd (const struct dpkey ∗dp)

  Get the data value in a dpkey struct as double.
- int disini (int alloc, int naxis, struct disprm ∗dis)

  Default constructor for the disprm struct.
- int disinit (int alloc, int naxis, struct disprm ∗dis, int ndpmax)

  Default constructor for the disprm struct.
- int discpy (int alloc, const struct disprm ∗dissrc, struct disprm ∗disdst)

  Copy routine for the disprm struct.
- int disfree (struct disprm ∗dis)

  Destructor for the disprm struct.
- int dissize (const struct disprm ∗dis, int sizes[2])

  Compute the size of a disprm struct.
- int disprt (const struct disprm ∗dis)

  Print routine for the disprm struct.
- int disperr (const struct disprm ∗dis, const char ∗prefix)

  Print error messages from a disprm struct.
- int dishdo (struct disprm ∗dis)

  write FITS headers using TPD.
- int disset (struct disprm ∗dis)

  Setup routine for the disprm struct.
- int disp2x (struct disprm ∗dis, const double rawcrd[ ], double discrd[ ])

  Apply distortion function.
- int disx2p (struct disprm ∗dis, const double discrd[ ], double rawcrd[ ])

  Apply de-distortion function.
- int diswarp (struct disprm ∗dis, const double pixblc[ ], const double pixtrc[ ], const double pixsamp[ ], int
  ∗nsamp, double maxdis[ ], double ∗maxtot, double avgdis[ ], double ∗avgtot, double rmsdis[ ], double ∗rmstot)

  Compute measures of distortion.

Variables

- const char ∗ dis_errmsg [ ]

    Status return messages.

19.3.1 Detailed Description

Routines in this suite implement extensions to the FITS World Coordinate System (WCS) standard proposed by
```
"Representations of distortions in FITS world coordinate systems",
Calabretta, M.R. et al. (WCS Paper IV, draft dated 2004/04/22),
available from http://www.atnf.csiro.au/people/Mark.Calabretta
```

In brief, a distortion function may occupy one of two positions in the WCS algorithm chain. Prior distortions precede the linear transformation matrix, whether it be **PC**i_ja or **CD**i_ja, and sequent distortions follow it. WCS Paper IV defines FITS keywords used to specify parameters for predefined distortion functions. The following are used for prior distortions:
```
CPDISja   ...(string-valued, identifies the distortion function)
DPja      ...(record-valued, parameters)
CPERRja   ...(floating-valued, maximum value)
```

Their counterparts for sequent distortions are **CQDIS**ia, **DQ**ia, and **CQERR**ia. An additional floating-valued keyword, **DVERR**a, records the maximum value of the combined distortions.

DPja and **DQ**ia are "record-valued". Syntactically, the keyvalues are standard FITS strings, but they are to be interpreted in a special way. The general form is
```
DPja = '<field-specifier>: <float>'
```

where the field-specifier consists of a sequence of fields separated by periods, and the ': ' between the field-specifier and the floating-point value is part of the record syntax. For example:
```
DP1 = 'AXIS.1: 1'
```

Certain field-specifiers are defined for all distortion functions, while others are defined only for particular distortions. Refer to WCS Paper IV for further details. wcspih() parses all distortion keywords and loads them into a disprm struct for analysis by disset() which knows (or possibly does not know) how to interpret them. Of the Paper IV distortion functions, only the general Polynomial distortion is currently implemented here.

TPV - the TPV "projection":
The distortion function component of the **TPV** celestial "projection" is also supported. The **TPV** projection, originally proposed in a draft of WCS Paper II, consists of a **TAN** projection with sequent polynomial distortion, the coefficients of which are encoded in **PV**i_ma keyrecords. Full details may be found at the registry of FITS conventions:
```
http://fits.gsfc.nasa.gov/registry/tpvwcs/tpv.html
```

Internally, wcsset() changes **TPV** to a **TAN** projection, translates the **PV**i_ma keywords to **DQ**ia and loads them into a disprm struct. These **DQ**ia keyrecords have the form
```
DQia = 'TPV.m: <value>'
```

where i, a, m, and the value for each **DQ**ia match each **PV**i_ma. Consequently, WCSLIB would handle a FITS header containing these keywords, along with **CQDIS**ia = **'TPV'** and the required **DQ**ia.**NAXES** and **DQ**ia.**AXIS.**ihat keywords.

Note that, as defined, **TPV** assumes that **CD**i_ja is used to define the linear transformation. The section on historical idiosyncrasies (below) cautions about translating **CD**i_ja to **PC**i_ja plus **CDELT**ia in this case.

SIP - Simple Imaging Polynomial:
These routines also support the Simple Imaging Polynomial (**SIP**), whose design was influenced by early drafts of WCS Paper IV. It is described in detail in
```
http://fits.gsfc.nasa.gov/registry/sip.html
```

SIP, which is defined only as a prior distortion for 2-D celestial images, has the interesting feature that it records an approximation to the inverse polynomial distortion function. This is used by disx2p() to provide an initial estimate

for its more precise iterative inversion. The special-purpose keywords used by **SIP** are parsed and translated by wcspih() as follows:

```
 A_p_q = <value>   ->   DP1 = 'SIP.FWD.p_q: <value>'
AP_p_q = <value>   ->   DP1 = 'SIP.REV.p_q: <value>'
 B_p_q = <value>   ->   DP2 = 'SIP.FWD.p_q: <value>'
BP_p_q = <value>   ->   DP2 = 'SIP.REV.p_q: <value>'
A_DMAX = <value>   ->   DPERR1 = <value>
B_DMAX = <value>   ->   DPERR2 = <value>
```

SIP's **A_ORDER** and **B_ORDER** keywords are not used. WCSLIB would recognise a FITS header containing the above keywords, along with **CPDIS**ja = '**SIP**' and the required **DP**ja**.NAXES** keywords.

DSS - Digitized Sky Survey:

The Digitized Sky Survey resulted from the production of the Guide Star Catalogue for the Hubble Space Telescope. Plate solutions based on a polynomial distortion function were encoded in FITS using non-standard keywords. Sect. 5.2 of WCS Paper IV describes how **DSS** coordinates may be translated to a sequent Polynomial distortion using two auxiliary variables. That translation is based on optimising the non-distortion component of the plate solution.

Following Paper IV, wcspih() translates the non-distortion component of **DSS** coordinates to standard WCS keywords (**CRPIX**ja, **PC**i_ja, **CRVAL**ia, etc), and fills a wcsprm struct with their values. It encodes the **DSS** polynomial coefficients as

```
AMDXm = <value>   ->   DQ1 = 'AMD.m: <value>'
AMDYm = <value>   ->   DQ2 = 'AMD.m: <value>'
```

WCSLIB would recognise a FITS header containing the above keywords, along with **CQDIS**ia = '**DSS**' and the required **DQ**ia**.NAXES** keywords.

WAT - the **TNX** and **ZPX** "projections":

The **TNX** and **ZPX** "projections" add a polynomial distortion function to the standard **TAN** and **ZPN** projections respectively. Unusually, the polynomial may be expressed as the sum of Chebyshev or Legendre polynomials, or as a simple sum of monomials, as described in

```
http://fits.gsfc.nasa.gov/registry/tnx/tnx-doc.html
http://fits.gsfc.nasa.gov/registry/zpxwcs/zpx.html
```

The polynomial coefficients are encoded in special-purpose **WAT**i_n keywords as a set of continued strings, thus providing the name for this distortion type. **WAT**i_n are parsed and translated by wcspih() into the following set:

```
DQi = 'WAT.POLY: <value>'
DQi = 'WAT.XMIN: <value>'
DQi = 'WAT.XMAX: <value>'
DQi = 'WAT.YMIN: <value>'
DQi = 'WAT.YMAX: <value>'
DQi = 'WAT.CHBY.m_n: <value>'  or
DQi = 'WAT.LEGR.m_n: <value>'  or
DQi = 'WAT.MONO.m_n: <value>'
```

along with **CQDIS**ia = '**WAT**' and the required **DP**ja**.NAXES** keywords. For **ZPX**, the **ZPN** projection parameters are also encoded in **WAT**i_n, and wcspih() translates these to standard **PV**i_ma.

Note that, as defined, **TNX** and **ZPX** assume that **CD**i_ja is used to define the linear transformation. The section on historical idiosyncrasies (below) cautions about translating **CD**i_ja to **PC**i_ja plus **CDELT**ia in this case.

TPD - Template Polynomial Distortion:

The "Template Polynomial Distortion" (**TPD**) is a superset of the **TPV**, **SIP**, **DSS**, and **WAT** (**TNX** & **ZPX**) polynomial distortions that also supports 1-D usage and inversions. Like **TPV**, **SIP**, and **DSS**, the form of the polynomial is fixed (the "template") and only the coefficients for the required terms are set non-zero. **TPD** generalizes **TPV** in going to 9th degree, **SIP** by accomodating **TPV**'s linear and radial terms, and **DSS** in both respects. While in theory the degree of the **WAT** polynomial distortion in unconstrained, in practice it is limited to values that can be handled by **TPD**.

Within WCSLIB, **TPV**, **SIP**, **DSS**, and **WAT** are all implemented as special cases of **TPD**. Indeed, **TPD** was developed precisely for that purpose. **WAT** distortions expressed as the sum of Chebyshev or Legendre polynomials are expanded for **TPD** as a simple sum of monomials. Moreover, the general Polynomial distortion is translated and implemented internally as **TPD** whenever possible.

However, WCSLIB also recognizes '**TPD**' as a distortion function in its own right (i.e. a recognized value of **CPDIS**ja or **CQDIS**ia), for use as both prior and sequent distortions. Its **DP**ja and **DQ**ia keyrecords have the form

```
DPja = 'TPD.FWD.m: <value>'
DPja = 'TPD.REV.m: <value>'
```

for the forward and reverse distortion functions. Moreover, like the general Polynomial distortion, **TPD** supports auxiliary variables, though only as a linear transformation of pixel coordinates (p1,p2):

```
x = a0 + a1*p1 + a2*p2
y = b0 + b1*p1 + b2*p2
```

where the coefficients of the auxiliary variables (x,y) are recorded as

```
DPja = 'AUX.1.COEFF.0: a0'      ...default 0.0
DPja = 'AUX.1.COEFF.1: a1'      ...default 1.0
DPja = 'AUX.1.COEFF.2: a2'      ...default 0.0
DPja = 'AUX.2.COEFF.0: b0'      ...default 0.0
DPja = 'AUX.2.COEFF.1: b1'      ...default 0.0
DPja = 'AUX.2.COEFF.2: b2'      ...default 1.0
```

Though nowhere near as powerful, in typical applications **TPD** is considerably faster than the general Polynomial distortion. As **TPD** has a finite and not too large number of possible terms (60), the coefficients for each can be stored (by disset()) in a fixed location in the disprm::dparm[] array. A large part of the speedup then arises from evaluating the polynomial using Horner's scheme.

Separate implementations for polynomials of each degree, and conditionals for 1-D polynomials and 2-D polynomials with and without the radial variable, ensure that unused terms mostly do not impose a significant computational overhead.

The **TPD** terms are as follows

```
0: 1     4: xx     12: xxxx    24: xxxxxx    40: xxxxxxxx
         5: xy     13: xxxy    25: xxxxxy    41: xxxxxxxy
1: x     6: yy     14: xxyy    26: xxxxyy    42: xxxxxxyy
2: y               15: xyyy    27: xxxyyy    43: xxxxxyyy
3: r     7: xxx    16: yyyy    28: xxyyyy    44: xxxxyyyy
         8: xxy                29: xyyyyy    45: xxxyyyyy
         9: xyy    17: xxxxx   30: yyyyyy    46: xxyyyyyy
        10: yyy    18: xxxxy                 47: xyyyyyyy
        11: rrr    19: xxxyy   31: xxxxxxx   48: yyyyyyyy
                   20: xxyyy   32: xxxxxxy
                   21: xyyyy   33: xxxxxyy   49: xxxxxxxxx
                   22: yyyyy   34: xxxxyyy   50: xxxxxxxxy
                   23: rrrrr   35: xxxyyyy   51: xxxxxxxyy
                               36: xxyyyyy   52: xxxxxxyyy
                               37: xyyyyyy   53: xxxxxyyyy
                               38: yyyyyyy   54: xxxxyyyyy
                               39: rrrrrrr   55: xxxyyyyyy
                                             56: xxyyyyyyy
                                             57: xyyyyyyyy
                                             58: yyyyyyyyy
                                             59: rrrrrrrrr
```

where r = $\sqrt{(x^2 + y^2)}$. Note that even powers of r are excluded since they can be accomodated by powers of $(x^2 + y^2)$.

Note here that "x" refers to the axis to which the distortion function is attached, with "y" being the complementary axis. So, for example, with longitude on axis 1 and latitude on axis 2, for **TPD** attached to axis 1, "x" refers to axis 1 and "y" to axis 2. For **TPD** attached to axis 2, "x" refers to axis 2, and "y" to axis 1.

TPV uses all terms up to 39. The m in its **PV**i_ma keywords translates directly to the **TPD** coefficient number.

SIP uses all terms except for 0, 3, 11, 23, 39, and 59, with terms 1 and 2 only used for the inverse. Its **A**_p_q, etc. keywords must be translated using a map.

DSS uses terms 0, 1, 2, 4, 5, 6, 7, 8, 9, 10, 17, 19, and 21. The presence of a non-zero constant term arises through the use of auxiliary variables with origin offset from the reference point of the **TAN** projection. However, in the translation given by WCS Paper IV, the distortion polynomial is zero, or very close to zero, at the reference pixel itself. The mapping between **DSS**'s **AMDX**m (or **AMDY**m) keyvalues and **TPD** coefficients, while still simple, is not quite as straightforward as for **TPV** and **SIP**.

WAT uses all but the radial terms, namely 3, 11, 23, 39, and 59. While the mapping between **WAT**'s monomial coefficients and **TPD** is fairly simple, for its expression in terms of a sum of Chebyshev or Legendre polynomials it is much less so.

Historical idiosyncrasies:

In addition to the above, some historical distortion functions have further idiosyncrasies that must be taken into account when translating them to **TPD**.

WCS Paper IV specifies that a distortion function returns a correction to be added to pixel coordinates (prior distortion) or intermediate pixel coordinates (sequent distortion). The correction is meant to be small so that ignoring the distortion function, i.e. setting the correction to zero, produces a commensurately small error.

However, rather than an additive correction, some historical distortion functions (**TPV**, **DSS**) define a polynomial that returns the corrected coordinates directly.

The difference between the two approaches is readily accounted for simply by adding or subtracting 1 from the coefficient of the first degree term of the polynomial. However, it opens the way for considerable confusion.

Additional to the formalism of WCS Paper IV, both the Polynomial and **TPD** distortion functions recognise a keyword
`DPja = 'DOCORR: 0'`

which is meant to apply generally to indicate that the distortion function returns the corrected coordinates directly. Any other value for **DOCORR** (or its absence) indicates that the distortion function returns an additive correction.

WCS Paper IV also specifies that the independent variables of a distortion function are pixel coordinates (prior distortion) or intermediate pixel coordinates (sequent distortion).

On the contrary, the independent variables of the **SIP** polynomial are pixel coordinate offsets from the reference pixel. This is readily handled via the renormalisation parameters
`DPja = 'OFFSET.jhat: <value>'`

where the value corresponds to **CRPIX**`ja`.

Likewise, because **TPV**, **TNX**, and **ZPX** are defined in terms of **CD**`i_ja`, the independent variables of the polynomial are intermediate world coordinates rather than intermediate pixel coordinates. Because sequent distortions are always applied before **CDELT**`ia`, if **CD**`i_ja` is translated to **PC**`i_ja` plus **CDELT**`ia`, then either **CDELT**`ia` must be unity, or the distortion polynomial coefficients must be adjusted to account for the change of scale.

Summary of the dis routines:

These routines apply the distortion functions defined by the extension to the FITS WCS standard proposed in Paper IV. They are based on the [disprm](#) struct which contains all information needed for the computations. The struct contains some members that must be set by the user, and others that are maintained by these routines, somewhat like a C++ class but with no encapsulation.

[dpfill()](#), [dpkeyi()](#), and [dpkeyd()](#) are provided to manage the dpkey struct.

[disndp()](#), [disini()](#), [disinit()](#), [discpy()](#), and [disfree()](#) are provided to manage the [disprm](#) struct, [dissize()](#) computes its total size including allocated memory, and [disprt()](#) prints its contents.

[disperr()](#) prints the error message(s) (if any) stored in a [disprm](#) struct.

[wcshdo()](#) normally writes **SIP** and **TPV** headers in their native form if at all possible. However, [dishdo()](#) may be used to set a flag that tells it to write the header in the form of the **TPD** translation used internally.

A setup routine, [disset()](#), computes intermediate values in the [disprm](#) struct from parameters in it that were supplied by the user. The struct always needs to be set up by [disset()](#), though [disset()](#) need not be called explicitly - refer to the explanation of [disprm::flag](#).

[disp2x()](#) and [disx2p()](#) implement the WCS distortion functions, [disp2x()](#) using separate functions, such as dispoly() and tpd7(), to do the computation.

An auxiliary routine, [diswarp()](#), computes various measures of the distortion over a specified range of coordinates.

PLEASE NOTE:

19.3.2 Macro Definition Documentation

19.3.2.1 DISP2X_ARGS `#define DISP2X_ARGS`

Value:
```
int inverse, const int iparm[], const double dparm[], \
int ncrd, const double rawcrd[], double *discrd
```

19.3.2.2 DISX2P_ARGS `#define DISX2P_ARGS`

Value:
```
int inverse, const int iparm[], const double dparm[], \
int ncrd, const double discrd[], double *rawcrd
```

19.3.2.3 DPLEN `#define DPLEN (sizeof(struct dpkey)/sizeof(int))`

19.3.2.4 DISLEN `#define DISLEN (sizeof(struct disprm)/sizeof(int))`

19.3.3 Enumeration Type Documentation

19.3.3.1 dis_errmsg_enum `enum dis_errmsg_enum`

Enumerator

DISERR_SUCCESS	
DISERR_NULL_POINTER	
DISERR_MEMORY	
DISERR_BAD_PARAM	
DISERR_DISTORT	
DISERR_DEDISTORT	

19.3.4 Function Documentation

19.3.4.1 disndp() `int disndp (`
              `int n )`

disndp() sets or gets the value of NDPMAX (default 256). This global variable controls the maximum number of dpkey structs, for holding **DP**ja or **DQ**ia keyvalues, that disini() should allocate space for. It is also used by disinit() as the default value of ndpmax.

PLEASE NOTE: This function is not thread-safe.

Parameters

in	n	Value of NDPMAX; ignored if < 0. Use a value less than zero to get the current value.

Returns

    Current value of NDPMAX.

19.3.4.2 dpfill() `int dpfill (`
              `struct dpkey * dp,`
              `const char * keyword,`
              `const char * field,`
              `int j,`
              `int type,`
              `int i,`
              `double f )`

dpfill() is a utility routine to aid in filling the contents of the dpkey struct. No checks are done on the validity of the inputs.

WCS Paper IV specifies the syntax of a record-valued keyword as
`keyword = '<field-specifier>: <float>'`

However, some **DP**ja and **DQ**ia record values, such as those of **DP**ja**.NAXES** and **DP**ja**.AXIS.**j, are intrinsically integer-valued. While FITS header parsers are not expected to know in advance which of **DP**ja and **DQ**ia are integral and which are floating point, if the record's value parses as an integer (i.e. without decimal point or exponent), then preferably enter it into the dpkey struct as an integer. Either way, it doesn't matter as disset() accepts either data type for all record values.

Parameters

in,out	dp	Store for **DP**ja and **DQ**ia keyvalues.
in	keyword	
in	field	These arguments are concatenated with an intervening "." to construct the full record field name, i.e. including the keyword name, **DP**ja or **DQ**ia (but excluding the colon delimiter which is NOT part of the name). Either may be given as a NULL pointer. Set both NULL to omit setting this component of the struct.
in	j	Axis number (1-relative), i.e. the j in **DP**ja or i in **DQ**ia. Can be given as 0, in which case the axis number will be obtained from the keyword component of the field name which must either have been given or preset. If j is non-zero, and keyword was given, then the value of j will be used to fill in the axis number.
in	type	Data type of the record's value • 0: Integer,
		• 1: Floating point.
in	i	For type == 0, the integer value of the record.
in	f	For type == 1, the floating point value of the record.

Returns

> Status return value:
>
> • 0: Success.

19.3.4.3 dpkeyi() `int dpkeyi (`
            `const struct dpkey * dp )`

dpkeyi() returns the data value in a dpkey struct as an integer value.

Parameters

| in,out | *dp* | Parsed contents of a **DP**ja or **DQ**ia keyrecord. |

Returns

> The record's value as int.

19.3.4.4 dpkeyd() `double dpkeyd (`
            `const struct dpkey * dp )`

dpkeyd() returns the data value in a dpkey struct as a floating point value.

Parameters

| in,out | *dp* | Parsed contents of a **DP**ja or **DQ**ia keyrecord. |

Returns

> The record's value as double.

19.3.4.5 disini() `int disini (`
            `int alloc,`
            `int naxis,`
            `struct disprm * dis )`

disini() is a thin wrapper on **disinit**(). It invokes it with ndpmax set to -1 which causes it to use the value of the global variable NDPMAX. It is thereby potentially thread-unsafe if NDPMAX is altered dynamically via disndp(). Use **disinit**() for a thread-safe alternative in this case.

19.3.4.6 disinit() `int disinit (`
        `int alloc,`
        `int naxis,`
        `struct disprm * dis,`
        `int ndpmax )`

disinit() allocates memory for arrays in a disprm struct and sets all members of the struct to default values.

PLEASE NOTE: every disprm struct must be initialized by **disinit**(), possibly repeatedly. On the first invokation, and only the first invokation, disprm::flag must be set to -1 to initialize memory management, regardless of whether **disinit**() will actually be used to allocate memory.

Parameters

in	alloc	If true, allocate memory unconditionally for arrays in the disprm struct.
		If false, it is assumed that pointers to these arrays have been set by the user except if they are null pointers in which case memory will be allocated for them regardless. (In other words, setting alloc true saves having to initalize these pointers to zero.)
in	naxis	The number of world coordinate axes, used to determine array sizes.
in,out	dis	Distortion function parameters. Note that, in order to initialize memory management disprm::flag must be set to -1 when dis is initialized for the first time (memory leaks may result if it had already been initialized).
in	ndpmax	The number of **DP**ja or **DQ**ia keywords to allocate space for. If set to -1, the value of the global variable NDPMAX will be used. This is potentially thread-unsafe if disndp() is being used dynamically to alter its value.

Returns

Status return value:

- 0: Success.
- 1: Null disprm pointer passed.
- 2: Memory allocation failed.

For returns > 1, a detailed error message is set in disprm::err if enabled, see wcserr_enable().

19.3.4.7 discpy() `int discpy (`
        `int alloc,`
        `const struct disprm * dissrc,`
        `struct disprm * disdst )`

discpy() does a deep copy of one disprm struct to another, using disinit() to allocate memory unconditionally for its arrays if required. Only the "information to be provided" part of the struct is copied; a call to disset() is required to initialize the remainder.

Parameters

in	alloc	If true, allocate memory unconditionally for arrays in the destination. Otherwise, it is assumed that pointers to these arrays have been set by the user except if they are null pointers in which case memory will be allocated for them regardless.
in	dissrc	Struct to copy from.
in,out	disdst	Struct to copy to. disprm::flag should be set to -1 if disdst was not previously initialized (memory leaks may result if it was previously initialized).

Returns

Status return value:

- 0: Success.
- 1: Null disprm pointer passed.
- 2: Memory allocation failed.

For returns > 1, a detailed error message is set in disprm::err if enabled, see wcserr_enable().

19.3.4.8 disfree() `int disfree (`
`struct disprm * dis )`

disfree() frees memory allocated for the disprm arrays by disinit(). disinit() keeps a record of the memory it allocates and **disfree**() will only attempt to free this.

PLEASE NOTE: disfree() must not be invoked on a disprm struct that was not initialized by disinit().

Parameters

in	*dis*	Distortion function parameters.

Returns

Status return value:

- 0: Success.
- 1: Null disprm pointer passed.

19.3.4.9 dissize() `int dissize (`
`const struct disprm * dis,`
`int sizes[2] )`

dissize() computes the full size of a disprm struct, including allocated memory.

Parameters

in	*dis*	Distortion function parameters. If NULL, the base size of the struct and the allocated size are both set to zero.
out	*sizes*	The first element is the base size of the struct as returned by sizeof(struct disprm). The second element is the total allocated size, in bytes, assuming that the allocation was done by disini(). This figure includes memory allocated for members of constituent structs, such as disprm::dp. It is not an error for the struct not to have been set up via tabset(), which normally results in additional memory allocation.

Returns

Status return value:

- 0: Success.

19.3.4.10 disprt() `int disprt (`
`const struct` disprm `*` *dis* `)`

disprt() prints the contents of a disprm struct using wcsprintf(). Mainly intended for diagnostic purposes.

Parameters

in	*dis*	Distortion function parameters.

Returns

Status return value:

- 0: Success.
- 1: Null disprm pointer passed.

19.3.4.11 disperr() `int disperr (`
`const struct` disprm `*` *dis,*
`const char` `*` *prefix* `)`

disperr() prints the error message(s) (if any) stored in a disprm struct. If there are no errors then nothing is printed. It uses wcserr_prt(), q.v.

Parameters

in	*dis*	Distortion function parameters.
in	*prefix*	If non-NULL, each output line will be prefixed with this string.

Returns

Status return value:

- 0: Success.
- 1: Null disprm pointer passed.

19.3.4.12 dishdo() `int dishdo (`
`struct` disprm `*` *dis* `)`

dishdo() sets a flag that tells wcshdo() to write FITS headers in the form of the **TPD** translation used internally. Normally **SIP** and **TPV** would be written in their native form if at all possible.

Parameters

in,out	*dis*	Distortion function parameters.

Returns

Status return value:

- 0: Success.

- 1: Null disprm pointer passed.

- 3: No **TPD** translation.

19.3.4.13 disset() int disset (
                struct disprm * *dis*)

disset(), sets up the disprm struct according to information supplied within it - refer to the explanation of disprm::flag.

Note that this routine need not be called directly; it will be invoked by disp2x() and disx2p() if the disprm::flag is anything other than a predefined magic value.

Parameters

in,out	*dis*	Distortion function parameters.

Returns

Status return value:

- 0: Success.

- 1: Null disprm pointer passed.

- 2: Memory allocation failed.

- 3: Invalid parameter.

For returns > 1, a detailed error message is set in disprm::err if enabled, see wcserr_enable().

19.3.4.14 disp2x() int disp2x (
                struct disprm * *dis,*
                const double *rawcrd[],*
                double *discrd[]*)

disp2x() applies the distortion functions. By definition, the distortion is in the pixel-to-world direction.

Depending on the point in the algorithm chain at which it is invoked, **disp2x**() may transform pixel coordinates to corrected pixel coordinates, or intermediate pixel coordinates to corrected intermediate pixel coordinates, or image coordinates to corrected image coordinates.

19.3.4.15 disx2p() `int disx2p (`
            `struct` disprm `* dis,`
            `const double discrd[],`
            `double rawcrd[] )`

disx2p() applies the inverse of the distortion functions. By definition, the de-distortion is in the world-to-pixel direction.

Depending on the point in the algorithm chain at which it is invoked, **disx2p**() may transform corrected pixel coordinates to pixel coordinates, or corrected intermediate pixel coordinates to intermediate pixel coordinates, or corrected image coordinates to image coordinates.

disx2p() iteratively solves for the inverse using disp2x(). It assumes that the distortion is small and the functions are well-behaved, being continuous and with continuous derivatives. Also that, to first order in the neighbourhood of the solution, discrd[j] $\sim$= a + b∗rawcrd[j], i.e. independent of rawcrd[i], where i != j. This is effectively equivalent to assuming that the distortion functions are separable to first order. Furthermore, a is assumed to be small, and b close to unity.

If disprm::**disx2p**() is defined, then **disx2p**() uses it to provide an initial estimate for its more precise iterative inversion.

Parameters

in,out	*dis*	Distortion function parameters.
in	*discrd*	Array of coordinates.
out	*rawcrd*	Array of coordinates to which the inverse distortion functions have been applied.

Returns

Status return value:

- 0: Success.
- 1: Null disprm pointer passed.
- 2: Memory allocation failed.
- 3: Invalid parameter.
- 5: De-distort error.

For returns > 1, a detailed error message is set in disprm::err if enabled, see wcserr_enable().

19.3.4.16 diswarp() `int diswarp (`
            `struct` disprm `* dis,`
            `const double pixblc[],`
            `const double pixtrc[],`
            `const double pixsamp[],`
            `int * nsamp,`
            `double maxdis[],`
            `double * maxtot,`
            `double avgdis[],`
            `double * avgtot,`
            `double rmsdis[],`
            `double * rmstot )`

diswarp() computes various measures of the distortion over a specified range of coordinates.

For prior distortions, the measures may be interpreted simply as an offset in pixel coordinates. For sequent distortions, the interpretation depends on the nature of the linear transformation matrix (**PC**i_ja or **CD**i_ja). If the latter introduces a scaling, then the measures will also be scaled. Note also that the image domain, which is rectangular in pixel coordinates, may be rotated, skewed, and/or stretched in intermediate pixel coordinates, and in general cannot be defined using pixblc[] and pixtrc[].

PLEASE NOTE: the measures of total distortion may be essentially meaningless if there are multiple sequent distortions with different scaling.

See also linwarp().

Parameters

in,out	*dis*	Distortion function parameters.
in	*pixblc*	Start of the range of pixel coordinates (for prior distortions), or intermediate pixel coordinates (for sequent distortions). May be specified as a NULL pointer which is interpreted as (1,1,...).
in	*pixtrc*	End of the range of pixel coordinates (prior) or intermediate pixel coordinates (sequent).
in	*pixsamp*	If positive or zero, the increment on the particular axis, starting at pixblc[]. Zero is interpreted as a unit increment. pixsamp may also be specified as a NULL pointer which is interpreted as all zeroes, i.e. unit increments on all axes. If negative, the grid size on the particular axis (the absolute value being rounded to the nearest integer). For example, if pixsamp is (-128.0,-128.0,...) then each axis will be sampled at 128 points between pixblc[] and pixtrc[] inclusive. Use caution when using this option on non-square images.
out	*nsamp*	The number of pixel coordinates sampled. Can be specified as a NULL pointer if not required.
out	*maxdis*	For each individual distortion function, the maximum absolute value of the distortion. Can be specified as a NULL pointer if not required.
out	*maxtot*	For the combination of all distortion functions, the maximum absolute value of the distortion. Can be specified as a NULL pointer if not required.
out	*avgdis*	For each individual distortion function, the mean value of the distortion. Can be specified as a NULL pointer if not required.
out	*avgtot*	For the combination of all distortion functions, the mean value of the distortion. Can be specified as a NULL pointer if not required.
out	*rmsdis*	For each individual distortion function, the root mean square deviation of the distortion. Can be specified as a NULL pointer if not required.
out	*rmstot*	For the combination of all distortion functions, the root mean square deviation of the distortion. Can be specified as a NULL pointer if not required.

Returns

Status return value:

- 0: Success.

- 1: Null disprm pointer passed.

- 2: Memory allocation failed.

- 3: Invalid parameter.

- 4: Distort error.

19.3.5 Variable Documentation

19.3.5.1 dis_errmsg `const char * dis_errmsg[]` `[extern]`

Error messages to match the status value returned from each function.

19.4 dis.h

[Go to the documentation of this file.](#)
```
1 /*============================================================================
2   WCSLIB 7.11 - an implementation of the FITS WCS standard.
3   Copyright (C) 1995-2022, Mark Calabretta
4
5   This file is part of WCSLIB.
6
7   WCSLIB is free software: you can redistribute it and/or modify it under the
8   terms of the GNU Lesser General Public License as published by the Free
9   Software Foundation, either version 3 of the License, or (at your option)
10  any later version.
11
12  WCSLIB is distributed in the hope that it will be useful, but WITHOUT ANY
13  WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
14  FOR A PARTICULAR PURPOSE.  See the GNU Lesser General Public License for
15  more details.
16
17  You should have received a copy of the GNU Lesser General Public License
18  along with WCSLIB.  If not, see http://www.gnu.org/licenses.
19
20  Author: Mark Calabretta, Australia Telescope National Facility, CSIRO.
21  http://www.atnf.csiro.au/people/Mark.Calabretta
22  $Id: dis.h,v 7.11 2022/04/26 06:13:52 mcalabre Exp $
23 *============================================================================
24 *
25 * WCSLIB 7.11 - C routines that implement the FITS World Coordinate System
26 * (WCS) standard.  Refer to the README file provided with WCSLIB for an
27 * overview of the library.
28 *
29 *
30 * Summary of the dis routines
31 * ---------------------------
32 * Routines in this suite implement extensions to the FITS World Coordinate
33 * System (WCS) standard proposed by
34 *
35 =   "Representations of distortions in FITS world coordinate systems",
36 =   Calabretta, M.R. et al. (WCS Paper IV, draft dated 2004/04/22),
37 =   available from http://www.atnf.csiro.au/people/Mark.Calabretta
38 *
39 * In brief, a distortion function may occupy one of two positions in the WCS
40 * algorithm chain.  Prior distortions precede the linear transformation
41 * matrix, whether it be PCi_ja or CDi_ja, and sequent distortions follow it.
42 * WCS Paper IV defines FITS keywords used to specify parameters for predefined
43 * distortion functions.  The following are used for prior distortions:
44 *
45 =   CPDISja   ...(string-valued, identifies the distortion function)
46 =   DPja      ...(record-valued, parameters)
47 =   CPERRja   ...(floating-valued, maximum value)
48 *
49 * Their counterparts for sequent distortions are CQDISia, DQia, and CQERRia.
50 * An additional floating-valued keyword, DVERRa, records the maximum value of
51 * the combined distortions.
52 *
53 * DPja and DQia are "record-valued".  Syntactically, the keyvalues are
54 * standard FITS strings, but they are to be interpreted in a special way.
55 * The general form is
56 *
57 =   DPja = '<field-specifier>: <float>'
58 *
59 * where the field-specifier consists of a sequence of fields separated by
60 * periods, and the ': ' between the field-specifier and the floating-point
61 * value is part of the record syntax.  For example:
62 *
63 =   DP1 = 'AXIS.1: 1'
64 *
65 * Certain field-specifiers are defined for all distortion functions, while
66 * others are defined only for particular distortions.  Refer to WCS Paper IV
```

```
 67 * for further details.  wcspih() parses all distortion keywords and loads them
 68 * into a disprm struct for analysis by disset() which knows (or possibly does
 69 * not know) how to interpret them.  Of the Paper IV distortion functions, only
 70 * the general Polynomial distortion is currently implemented here.
 71 *
 72 * TPV - the TPV "projection":
 73 * ---------------------------
 74 * The distortion function component of the TPV celestial "projection" is also
 75 * supported.  The TPV projection, originally proposed in a draft of WCS Paper
 76 * II, consists of a TAN projection with sequent polynomial distortion, the
 77 * coefficients of which are encoded in PVi_ma keyrecords.  Full details may be
 78 * found at the registry of FITS conventions:
 79 *
 80 =   http://fits.gsfc.nasa.gov/registry/tpvwcs/tpv.html
 81 *
 82 * Internally, wcsset() changes TPV to a TAN projection, translates the PVi_ma
 83 * keywords to DQia and loads them into a disprm struct.  These DQia keyrecords
 84 * have the form
 85 *
 86 =   DQia = 'TPV.m: <value>'
 87 *
 88 * where i, a, m, and the value for each DQia match each PVi_ma.  Consequently,
 89 * WCSLIB would handle a FITS header containing these keywords, along with
 90 * CQDISia = 'TPV' and the required DQia.NAXES and DQia.AXIS.ihat keywords.
 91 *
 92 * Note that, as defined, TPV assumes that CDi_ja is used to define the linear
 93 * transformation.  The section on historical idiosyncrasies (below) cautions
 94 * about translating CDi_ja to PCi_ja plus CDELTia in this case.
 95 *
 96 * SIP - Simple Imaging Polynomial:
 97 * --------------------------------
 98 * These routines also support the Simple Imaging Polynomial (SIP), whose
 99 * design was influenced by early drafts of WCS Paper IV.  It is described in
100 * detail in
101 *
102 =   http://fits.gsfc.nasa.gov/registry/sip.html
103 *
104 * SIP, which is defined only as a prior distortion for 2-D celestial images,
105 * has the interesting feature that it records an approximation to the inverse
106 * polynomial distortion function.  This is used by disx2p() to provide an
107 * initial estimate for its more precise iterative inversion.   The
108 * special-purpose keywords used by SIP are parsed and translated by wcspih()
109 * as follows:
110 *
111 =     A_p_q = <value>   ->   DP1 = 'SIP.FWD.p_q: <value>'
112 =    AP_p_q = <value>   ->   DP1 = 'SIP.REV.p_q: <value>'
113 =     B_p_q = <value>   ->   DP2 = 'SIP.FWD.p_q: <value>'
114 =    BP_p_q = <value>   ->   DP2 = 'SIP.REV.p_q: <value>'
115 =    A_DMAX = <value>   ->   DPERR1 = <value>
116 =    B_DMAX = <value>   ->   DPERR2 = <value>
117 *
118 * SIP's A_ORDER and B_ORDER keywords are not used.  WCSLIB would recognise a
119 * FITS header containing the above keywords, along with CPDISja = 'SIP' and
120 * the required DPja.NAXES keywords.
121 *
122 * DSS - Digitized Sky Survey:
123 * ---------------------------
124 * The Digitized Sky Survey resulted from the production of the Guide Star
125 * Catalogue for the Hubble Space Telescope.  Plate solutions based on a
126 * polynomial distortion function were encoded in FITS using non-standard
127 * keywords.  Sect. 5.2 of WCS Paper IV describes how DSS coordinates may be
128 * translated to a sequent Polynomial distortion using two auxiliary variables.
129 * That translation is based on optimising the non-distortion component of the
130 * plate solution.
131 *
132 * Following Paper IV, wcspih() translates the non-distortion component of DSS
133 * coordinates to standard WCS keywords (CRPIXja, PCi_ja, CRVALia, etc), and
134 * fills a wcsprm struct with their values.  It encodes the DSS polynomial
135 * coefficients as
136 *
137 =     AMDXm = <value>   ->   DQ1 = 'AMD.m: <value>'
138 =     AMDYm = <value>   ->   DQ2 = 'AMD.m: <value>'
139 *
140 * WCSLIB would recognise a FITS header containing the above keywords, along
141 * with CQDISia = 'DSS' and the required DQia.NAXES keywords.
142 *
143 * WAT - the TNX and ZPX "projections":
144 * ------------------------------------
145 * The TNX and ZPX "projections" add a polynomial distortion function to the
146 * standard TAN and ZPN projections respectively.  Unusually, the polynomial
147 * may be expressed as the sum of Chebyshev or Legendre polynomials, or as a
148 * simple sum of monomials, as described in
149 *
150 =   http://fits.gsfc.nasa.gov/registry/tnx/tnx-doc.html
151 =   http://fits.gsfc.nasa.gov/registry/zpxwcs/zpx.html
152 *
153 * The polynomial coefficients are encoded in special-purpose WATi_n keywords
```

```
154 * as a set of continued strings, thus providing the name for this distortion
155 * type.  WATi_n are parsed and translated by wcspih() into the following set:
156 *
157 =    DQi = 'WAT.POLY: <value>'
158 =    DQi = 'WAT.XMIN: <value>'
159 =    DQi = 'WAT.XMAX: <value>'
160 =    DQi = 'WAT.YMIN: <value>'
161 =    DQi = 'WAT.YMAX: <value>'
162 =    DQi = 'WAT.CHBY.m_n: <value>'  or
163 =    DQi = 'WAT.LEGR.m_n: <value>'  or
164 =    DQi = 'WAT.MONO.m_n: <value>'
165 *
166 * along with CQDISia = 'WAT' and the required DPja.NAXES keywords.  For ZPX,
167 * the ZPN projection parameters are also encoded in WATi_n, and wcspih()
168 * translates these to standard PVi_ma.
169 *
170 * Note that, as defined, TNX and ZPX assume that CDi_ja is used to define the
171 * linear transformation.  The section on historical idiosyncrasies (below)
172 * cautions about translating CDi_ja to PCi_ja plus CDELTia in this case.
173 *
174 * TPD - Template Polynomial Distortion:
175 * -------------------------------------
176 * The "Template Polynomial Distortion" (TPD) is a superset of the TPV, SIP,
177 * DSS, and WAT (TNX & ZPX) polynomial distortions that also supports 1-D usage
178 * and inversions.  Like TPV, SIP, and DSS, the form of the polynomial is fixed
179 * (the "template") and only the coefficients for the required terms are set
180 * non-zero.  TPD generalizes TPV in going to 9th degree, SIP by accomodating
181 * TPV's linear and radial terms, and DSS in both respects.  While in theory
182 * the degree of the WAT polynomial distortion in unconstrained, in practice it
183 * is limited to values that can be handled by TPD.
184 *
185 * Within WCSLIB, TPV, SIP, DSS, and WAT are all implemented as special cases
186 * of TPD.  Indeed, TPD was developed precisely for that purpose.  WAT
187 * distortions expressed as the sum of Chebyshev or Legendre polynomials are
188 * expanded for TPD as a simple sum of monomials.  Moreover, the general
189 * Polynomial distortion is translated and implemented internally as TPD
190 * whenever possible.
191 *
192 * However, WCSLIB also recognizes 'TPD' as a distortion function in its own
193 * right (i.e. a recognized value of CPDISja or CQDISia), for use as both prior
194 * and sequent distortions.  Its DPja and DQia keyrecords have the form
195 *
196 =    DPja = 'TPD.FWD.m: <value>'
197 =    DPja = 'TPD.REV.m: <value>'
198 *
199 * for the forward and reverse distortion functions.  Moreover, like the
200 * general Polynomial distortion, TPD supports auxiliary variables, though only
201 * as a linear transformation of pixel coordinates (p1,p2):
202 *
203 =    x = a0 + a1*p1 + a2*p2
204 =    y = b0 + b1*p1 + b2*p2
205 *
206 * where the coefficients of the auxiliary variables (x,y) are recorded as
207 *
208 =    DPja = 'AUX.1.COEFF.0: a0'     ...default 0.0
209 =    DPja = 'AUX.1.COEFF.1: a1'     ...default 1.0
210 =    DPja = 'AUX.1.COEFF.2: a2'     ...default 0.0
211 =    DPja = 'AUX.2.COEFF.0: b0'     ...default 0.0
212 =    DPja = 'AUX.2.COEFF.1: b1'     ...default 0.0
213 =    DPja = 'AUX.2.COEFF.2: b2'     ...default 1.0
214 *
215 * Though nowhere near as powerful, in typical applications TPD is considerably
216 * faster than the general Polynomial distortion.  As TPD has a finite and not
217 * too large number of possible terms (60), the coefficients for each can be
218 * stored (by disset()) in a fixed location in the disprm::dparm[] array.  A
219 * large part of the speedup then arises from evaluating the polynomial using
220 * Horner's scheme.
221 *
222 * Separate implementations for polynomials of each degree, and conditionals
223 * for 1-D polynomials and 2-D polynomials with and without the radial
224 * variable, ensure that unused terms mostly do not impose a significant
225 * computational overhead.
226 *
227 * The TPD terms are as follows
228 *
229 =   0: 1     4: xx     12: xxxx    24: xxxxxx    40: xxxxxxxx
230 =            5: xy     13: xxxy    25: xxxxxy    41: xxxxxxxy
231 =   1: x     6: yy     14: xxyy    26: xxxxyy    42: xxxxxxyy
232 =   2: y               15: xyyy    27: xxxyyy    43: xxxxxyyy
233 =   3: r     7: xxx    16: yyyy    28: xxyyyy    44: xxxxyyyy
234 =           8: xxy               29: xyyyyy     45: xxxyyyyy
235 =           9: xyy    17: xxxxx   30: yyyyyy     46: xxyyyyyy
236 =          10: yyy    18: xxxxy                  47: xyyyyyyy
237 =          11: rrr    19: xxxyy   31: xxxxxx     48: yyyyyyyy
238 =                     20: xxyyy   32: xxxxxy
239 =                     21: xyyyy   33: xxxxyy     49: xxxxxxxxx
240 =                     22: yyyyy   34: xxxyyy     50: xxxxxxxxy
```

```
241 =                           23: rrrrr     35: xxxyyyy    51: xxxxxxxyy
242 =                                         36: xxyyyyy    52: xxxxxxyyy
243 =                                         37: xyyyyyy    53: xxxxxyyyy
244 =                                         38: yyyyyyy    54: xxxxyyyyy
245 =                                         39: rrrrrrr    55: xxxyyyyyy
246 =                                                        56: xxyyyyyyy
247 =                                                        57: xyyyyyyyy
248 =                                                        58: yyyyyyyyy
249 =                                                        59: rrrrrrrrr
250 *
251 * where r = sqrt(xx + yy).  Note that even powers of r are excluded since they
252 * can be accomodated by powers of (xx + yy).
253 *
254 * Note here that "x" refers to the axis to which the distortion function is
255 * attached, with "y" being the complementary axis.  So, for example, with
256 * longitude on axis 1 and latitude on axis 2, for TPD attached to axis 1, "x"
257 * refers to axis 1 and "y" to axis 2.  For TPD attached to axis 2, "x" refers
258 * to axis 2, and "y" to axis 1.
259 *
260 * TPV uses all terms up to 39.  The m in its PVi_ma keywords translates
261 * directly to the TPD coefficient number.
262 *
263 * SIP uses all terms except for 0, 3, 11, 23, 39, and 59, with terms 1 and 2
264 * only used for the inverse.  Its A_p_q, etc. keywords must be translated
265 * using a map.
266 *
267 * DSS uses terms 0, 1, 2, 4, 5, 6, 7, 8, 9, 10, 17, 19, and 21.  The presence
268 * of a non-zero constant term arises through the use of auxiliary variables
269 * with origin offset from the reference point of the TAN projection.  However,
270 * in the translation given by WCS Paper IV, the distortion polynomial is zero,
271 * or very close to zero, at the reference pixel itself.  The mapping between
272 * DSS's AMDXm (or AMDYm) keyvalues and TPD coefficients, while still simple,
273 * is not quite as straightforward as for TPV and SIP.
274 *
275 * WAT uses all but the radial terms, namely 3, 11, 23, 39, and 59.  While the
276 * mapping between WAT's monomial coefficients and TPD is fairly simple, for
277 * its expression in terms of a sum of Chebyshev or Legendre polynomials it is
278 * much less so.
279 *
280 * Historical idiosyncrasies:
281 * --------------------------
282 * In addition to the above, some historical distortion functions have further
283 * idiosyncrasies that must be taken into account when translating them to TPD.
284 *
285 * WCS Paper IV specifies that a distortion function returns a correction to be
286 * added to pixel coordinates (prior distortion) or intermediate pixel
287 * coordinates (sequent distortion).  The correction is meant to be small so
288 * that ignoring the distortion function, i.e. setting the correction to zero,
289 * produces a commensurately small error.
290 *
291 * However, rather than an additive correction, some historical distortion
292 * functions (TPV, DSS) define a polynomial that returns the corrected
293 * coordinates directly.
294 *
295 * The difference between the two approaches is readily accounted for simply by
296 * adding or subtracting 1 from the coefficient of the first degree term of the
297 * polynomial.  However, it opens the way for considerable confusion.
298 *
299 * Additional to the formalism of WCS Paper IV, both the Polynomial and TPD
300 * distortion functions recognise a keyword
301 *
302 =   DPja = 'DOCORR: 0'
303 *
304 * which is meant to apply generally to indicate that the distortion function
305 * returns the corrected coordinates directly.  Any other value for DOCORR (or
306 * its absence) indicates that the distortion function returns an additive
307 * correction.
308 *
309 * WCS Paper IV also specifies that the independent variables of a distortion
310 * function are pixel coordinates (prior distortion) or intermediate pixel
311 * coordinates (sequent distortion).
312 *
313 * On the contrary, the independent variables of the SIP polynomial are pixel
314 * coordinate offsets from the reference pixel.  This is readily handled via
315 * the renormalisation parameters
316 *
317 =   DPja = 'OFFSET.jhat: <value>'
318 *
319 * where the value corresponds to CRPIXja.
320 *
321 * Likewise, because TPV, TNX, and ZPX are defined in terms of CDi_ja, the
322 * independent variables of the polynomial are intermediate world coordinates
323 * rather than intermediate pixel coordinates.  Because sequent distortions
324 * are always applied before CDELTia, if CDi_ja is translated to PCi_ja plus
325 * CDELTia, then either CDELTia must be unity, or the distortion polynomial
326 * coefficients must be adjusted to account for the change of scale.
327 *
```

```
328 * Summary of the dis routines:
329 * ---------------------------
330 * These routines apply the distortion functions defined by the extension to
331 * the FITS WCS standard proposed in Paper IV.  They are based on the disprm
332 * struct which contains all information needed for the computations.  The
333 * struct contains some members that must be set by the user, and others that
334 * are maintained by these routines, somewhat like a C++ class but with no
335 * encapsulation.
336 *
337 * dpfill(), dpkeyi(), and dpkeyd() are provided to manage the dpkey struct.
338 *
339 * disndp(), disini(), disinit(), discpy(), and disfree() are provided to
340 * manage the disprm struct, dissize() computes its total size including
341 * allocated memory, and disprt() prints its contents.
342 *
343 * disperr() prints the error message(s) (if any) stored in a disprm struct.
344 *
345 * wcshdo() normally writes SIP and TPV headers in their native form if at all
346 * possible.  However, dishdo() may be used to set a flag that tells it to
347 * write the header in the form of the TPD translation used internally.
348 *
349 * A setup routine, disset(), computes intermediate values in the disprm struct
350 * from parameters supplied in it that were supplied by the user.  The struct always
351 * needs to be set up by disset(), though disset() need not be called
352 * explicitly - refer to the explanation of disprm::flag.
353 *
354 * disp2x() and disx2p() implement the WCS distortion functions, disp2x() using
355 * separate functions, such as dispoly() and tpd7(), to do the computation.
356 *
357 * An auxiliary routine, diswarp(), computes various measures of the distortion
358 * over a specified range of coordinates.
359 *
360 * PLEASE NOTE: Distortions are not yet handled by wcsbth(), or wcscompare().
361 *
362 *
363 * disndp() - Memory allocation for DPja and DQia
364 * ----------------------------------------------
365 * disndp() sets or gets the value of NDPMAX (default 256).  This global
366 * variable controls the maximum number of dpkey structs, for holding DPja or
367 * DQia keyvalues, that disini() should allocate space for.  It is also used by
368 * disinit() as the default value of ndpmax.
369 *
370 * PLEASE NOTE: This function is not thread-safe.
371 *
372 * Given:
373 *   n         int       Value of NDPMAX; ignored if < 0.  Use a value less
374 *                       than zero to get the current value.
375 *
376 * Function return value:
377 *             int       Current value of NDPMAX.
378 *
379 *
380 * dpfill() - Fill the contents of a dpkey struct
381 * ----------------------------------------------
382 * dpfill() is a utility routine to aid in filling the contents of the dpkey
383 * struct.  No checks are done on the validity of the inputs.
384 *
385 * WCS Paper IV specifies the syntax of a record-valued keyword as
386 *
387 =   keyword = '<field-specifier>: <float>'
388 *
389 * However, some DPja and DQia record values, such as those of DPja.NAXES and
390 * DPja.AXIS.j, are intrinsically integer-valued.  While FITS header parsers
391 * are not expected to know in advance which of DPja and DQia are integral and
392 * which are floating point, if the record's value parses as an integer (i.e.
393 * without decimal point or exponent), then preferably enter it into the dpkey
394 * struct as an integer.  Either way, it doesn't matter as disset() accepts
395 * either data type for all record values.
396 *
397 * Given and returned:
398 *   dp        struct dpkey*
399 *                       Store for DPja and DQia keyvalues.
400 *
401 * Given:
402 *   keyword   const char *
403 *   field     const char *
404 *                       These arguments are concatenated with an intervening
405 *                       "." to construct the full record field name, i.e.
406 *                       including the keyword name, DPja or DQia (but
407 *                       excluding the colon delimiter which is NOT part of the
408 *                       name). Either may be given as a NULL pointer.  Set
409 *                       both NULL to omit setting this component of the
410 *                       struct.
411 *
412 *   j         int       Axis number (1-relative), i.e. the j in DPja or
413 *                       i in DQia.  Can be given as 0, in which case the axis
414 *                       number will be obtained from the keyword component of
```

```
415 *                        the field name which must either have been given or
416 *                        preset.
417 *
418 *                        If j is non-zero, and keyword was given, then the
419 *                        value of j will be used to fill in the axis number.
420 *
421 *   type      int        Data type of the record's value
422 *                          0: Integer,
423 *                          1: Floating point.
424 *
425 *   i         int        For type == 0, the integer value of the record.
426 *
427 *   f         double     For type == 1, the floating point value of the record.
428 *
429 * Function return value:
430 *             int        Status return value:
431 *                          0: Success.
432 *
433 *
434 * dpkeyi() - Get the data value in a dpkey struct as int
435 * ------------------------------------------------------
436 * dpkeyi() returns the data value in a dpkey struct as an integer value.
437 *
438 * Given and returned:
439 *   dp        const struct dpkey *
440 *                        Parsed contents of a DPja or DQia keyrecord.
441 *
442 * Function return value:
443 *             int        The record's value as int.
444 *
445 *
446 * dpkeyd() - Get the data value in a dpkey struct as double
447 * --------------------------------------------------------
448 * dpkeyd() returns the data value in a dpkey struct as a floating point
449 * value.
450 *
451 * Given and returned:
452 *   dp        const struct dpkey *
453 *                        Parsed contents of a DPja or DQia keyrecord.
454 *
455 * Function return value:
456 *             double     The record's value as double.
457 *
458 *
459 * disini() - Default constructor for the disprm struct
460 * ----------------------------------------------------
461 * disini() is a thin wrapper on disinit().  It invokes it with ndpmax set
462 * to -1 which causes it to use the value of the global variable NDPMAX.  It
463 * is thereby potentially thread-unsafe if NDPMAX is altered dynamically via
464 * disndp().  Use disinit() for a thread-safe alternative in this case.
465 *
466 *
467 * disinit() - Default constructor for the disprm struct
468 * -----------------------------------------------------
469 * disinit() allocates memory for arrays in a disprm struct and sets all
470 * members of the struct to default values.
471 *
472 * PLEASE NOTE: every disprm struct must be initialized by disinit(), possibly
473 * repeatedly.  On the first invokation, and only the first invokation,
474 * disprm::flag must be set to -1 to initialize memory management, regardless
475 * of whether disinit() will actually be used to allocate memory.
476 *
477 * Given:
478 *   alloc     int        If true, allocate memory unconditionally for arrays in
479 *                        the disprm struct.
480 *
481 *                        If false, it is assumed that pointers to these arrays
482 *                        have been set by the user except if they are null
483 *                        pointers in which case memory will be allocated for
484 *                        them regardless.  (In other words, setting alloc true
485 *                        saves having to initalize these pointers to zero.)
486 *
487 *   naxis     int        The number of world coordinate axes, used to determine
488 *                        array sizes.
489 *
490 * Given and returned:
491 *   dis       struct disprm*
492 *                        Distortion function parameters.  Note that, in order
493 *                        to initialize memory management disprm::flag must be
494 *                        set to -1 when dis is initialized for the first time
495 *                        (memory leaks may result if it had already been
496 *                        initialized).
497 *
498 * Given:
499 *   ndpmax    int        The number of DPja or DQia keywords to allocate space
500 *                        for.  If set to -1, the value of the global variable
501 *                        NDPMAX will be used.  This is potentially
```

```
502 *                         thread-unsafe if disndp() is being used dynamically to
503 *                         alter its value.
504 *
505 * Function return value:
506 *             int        Status return value:
507 *                          0: Success.
508 *                          1: Null disprm pointer passed.
509 *                          2: Memory allocation failed.
510 *
511 *                         For returns > 1, a detailed error message is set in
512 *                         disprm::err if enabled, see wcserr_enable().
513 *
514 *
515 * discpy() - Copy routine for the disprm struct
516 * ---------------------------------------------
517 * discpy() does a deep copy of one disprm struct to another, using disinit()
518 * to allocate memory unconditionally for its arrays if required.  Only the
519 * "information to be provided" part of the struct is copied; a call to
520 * disset() is required to initialize the remainder.
521 *
522 * Given:
523 *   alloc      int        If true, allocate memory unconditionally for arrays in
524 *                         the destination.  Otherwise, it is assumed that
525 *                         pointers to these arrays have been set by the user
526 *                         except if they are null pointers in which case memory
527 *                         will be allocated for them regardless.
528 *
529 *   dissrc     const struct disprm*
530 *                         Struct to copy from.
531 *
532 * Given and returned:
533 *   disdst     struct disprm*
534 *                         Struct to copy to.  disprm::flag should be set to -1
535 *                         if disdst was not previously initialized (memory leaks
536 *                         may result if it was previously initialized).
537 *
538 * Function return value:
539 *             int        Status return value:
540 *                          0: Success.
541 *                          1: Null disprm pointer passed.
542 *                          2: Memory allocation failed.
543 *
544 *                         For returns > 1, a detailed error message is set in
545 *                         disprm::err if enabled, see wcserr_enable().
546 *
547 *
548 * disfree() - Destructor for the disprm struct
549 * --------------------------------------------
550 * disfree() frees memory allocated for the disprm arrays by disinit().
551 * disinit() keeps a record of the memory it allocates and disfree() will only
552 * attempt to free this.
553 *
554 * PLEASE NOTE: disfree() must not be invoked on a disprm struct that was not
555 * initialized by disinit().
556 *
557 * Given:
558 *   dis        struct disprm*
559 *                         Distortion function parameters.
560 *
561 * Function return value:
562 *             int        Status return value:
563 *                          0: Success.
564 *                          1: Null disprm pointer passed.
565 *
566 *
567 * dissize() - Compute the size of a disprm struct
568 * -----------------------------------------------
569 * dissize() computes the full size of a disprm struct, including allocated
570 * memory.
571 *
572 * Given:
573 *   dis        const struct disprm*
574 *                         Distortion function parameters.
575 *
576 *                         If NULL, the base size of the struct and the allocated
577 *                         size are both set to zero.
578 *
579 * Returned:
580 *   sizes      int[2]     The first element is the base size of the struct as
581 *                         returned by sizeof(struct disprm).  The second element
582 *                         is the total allocated size, in bytes, assuming that
583 *                         the allocation was done by disini().  This figure
584 *                         includes memory allocated for members of constituent
585 *                         structs, such as disprm::dp.
586 *
587 *                         It is not an error for the struct not to have been set
588 *                         up via tabset(), which normally results in additional
```

```
589 *                        memory allocation.
590 *
591 * Function return value:
592 *            int       Status return value:
593 *                        0: Success.
594 *
595 *
596 * disprt() - Print routine for the disprm struct
597 * ----------------------------------------------
598 * disprt() prints the contents of a disprm struct using wcsprintf().  Mainly
599 * intended for diagnostic purposes.
600 *
601 * Given:
602 *   dis       const struct disprm*
603 *                        Distortion function parameters.
604 *
605 * Function return value:
606 *            int       Status return value:
607 *                        0: Success.
608 *                        1: Null disprm pointer passed.
609 *
610 *
611 * disperr() - Print error messages from a disprm struct
612 * -----------------------------------------------------
613 * disperr() prints the error message(s) (if any) stored in a disprm struct.
614 * If there are no errors then nothing is printed.  It uses wcserr_prt(), q.v.
615 *
616 * Given:
617 *   dis       const struct disprm*
618 *                        Distortion function parameters.
619 *
620 *   prefix    const char *
621 *                        If non-NULL, each output line will be prefixed with
622 *                        this string.
623 *
624 * Function return value:
625 *            int       Status return value:
626 *                        0: Success.
627 *                        1: Null disprm pointer passed.
628 *
629 *
630 * dishdo() - write FITS headers using TPD
631 * ---------------------------------------
632 * dishdo() sets a flag that tells wcshdo() to write FITS headers in the form
633 * of the TPD translation used internally.  Normally SIP and TPV would be
634 * written in their native form if at all possible.
635 *
636 * Given and returned:
637 *   dis       struct disprm*
638 *                        Distortion function parameters.
639 *
640 * Function return value:
641 *            int       Status return value:
642 *                        0: Success.
643 *                        1: Null disprm pointer passed.
644 *                        3: No TPD translation.
645 *
646 *
647 * disset() - Setup routine for the disprm struct
648 * ----------------------------------------------
649 * disset(), sets up the disprm struct according to information supplied within
650 * it - refer to the explanation of disprm::flag.
651 *
652 * Note that this routine need not be called directly; it will be invoked by
653 * disp2x() and disx2p() if the disprm::flag is anything other than a
654 * predefined magic value.
655 *
656 * Given and returned:
657 *   dis       struct disprm*
658 *                        Distortion function parameters.
659 *
660 * Function return value:
661 *            int       Status return value:
662 *                        0: Success.
663 *                        1: Null disprm pointer passed.
664 *                        2: Memory allocation failed.
665 *                        3: Invalid parameter.
666 *
667 *                        For returns > 1, a detailed error message is set in
668 *                        disprm::err if enabled, see wcserr_enable().
669 *
670 *
671 * disp2x() - Apply distortion function
672 * ------------------------------------
673 * disp2x() applies the distortion functions.  By definition, the distortion
674 * is in the pixel-to-world direction.
675 *
```

```
676  * Depending on the point in the algorithm chain at which it is invoked,
677  * disp2x() may transform pixel coordinates to corrected pixel coordinates, or
678  * intermediate pixel coordinates to corrected intermediate pixel coordinates,
679  * or image coordinates to corrected image coordinates.
680  *
681  *
682  * Given and returned:
683  *   dis       struct disprm*
684  *                     Distortion function parameters.
685  *
686  * Given:
687  *   rawcrd    const double[naxis]
688  *                     Array of coordinates.
689  *
690  * Returned:
691  *   discrd    double[naxis]
692  *                     Array of coordinates to which the distortion functions
693  *                     have been applied.
694  *
695  * Function return value:
696  *             int       Status return value:
697  *                         0: Success.
698  *                         1: Null disprm pointer passed.
699  *                         2: Memory allocation failed.
700  *                         3: Invalid parameter.
701  *                         4: Distort error.
702  *
703  *                     For returns > 1, a detailed error message is set in
704  *                     disprm::err if enabled, see wcserr_enable().
705  *
706  *
707  * disx2p() - Apply de-distortion function
708  * ---------------------------------------
709  * disx2p() applies the inverse of the distortion functions.  By definition,
710  * the de-distortion is in the world-to-pixel direction.
711  *
712  * Depending on the point in the algorithm chain at which it is invoked,
713  * disx2p() may transform corrected pixel coordinates to pixel coordinates, or
714  * corrected intermediate pixel coordinates to intermediate pixel coordinates,
715  * or corrected image coordinates to image coordinates.
716  *
717  * disx2p() iteratively solves for the inverse using disp2x().  It assumes
718  * that the distortion is small and the functions are well-behaved, being
719  * continuous and with continuous derivatives.  Also that, to first order
720  * in the neighbourhood of the solution, discrd[j] ~= a + b*rawcrd[j], i.e.
721  * independent of rawcrd[i], where i != j.  This is effectively equivalent to
722  * assuming that the distortion functions are separable to first order.
723  * Furthermore, a is assumed to be small, and b close to unity.
724  *
725  * If disprm::disx2p() is defined, then disx2p() uses it to provide an initial
726  * estimate for its more precise iterative inversion.
727  *
728  * Given and returned:
729  *   dis       struct disprm*
730  *                     Distortion function parameters.
731  *
732  * Given:
733  *   discrd    const double[naxis]
734  *                     Array of coordinates.
735  *
736  * Returned:
737  *   rawcrd    double[naxis]
738  *                     Array of coordinates to which the inverse distortion
739  *                     functions have been applied.
740  *
741  * Function return value:
742  *             int       Status return value:
743  *                         0: Success.
744  *                         1: Null disprm pointer passed.
745  *                         2: Memory allocation failed.
746  *                         3: Invalid parameter.
747  *                         5: De-distort error.
748  *
749  *                     For returns > 1, a detailed error message is set in
750  *                     disprm::err if enabled, see wcserr_enable().
751  *
752  *
753  * diswarp() - Compute measures of distortion
754  * ------------------------------------------
755  * diswarp() computes various measures of the distortion over a specified range
756  * of coordinates.
757  *
758  * For prior distortions, the measures may be interpreted simply as an offset
759  * in pixel coordinates.  For sequent distortions, the interpretation depends
760  * on the nature of the linear transformation matrix (PCi_ja or CDi_ja).  If
761  * the latter introduces a scaling, then the measures will also be scaled.
762  * Note also that the image domain, which is rectangular in pixel coordinates,
```

```
763 * may be rotated, skewed, and/or stretched in intermediate pixel coordinates,
764 * and in general cannot be defined using pixblc[] and pixtrc[].
765 *
766 * PLEASE NOTE: the measures of total distortion may be essentially meaningless
767 * if there are multiple sequent distortions with different scaling.
768 *
769 * See also linwarp().
770 *
771 * Given and returned:
772 *   dis       struct disprm*
773 *                       Distortion function parameters.
774 *
775 * Given:
776 *   pixblc    const double[naxis]
777 *                       Start of the range of pixel coordinates (for prior
778 *                       distortions), or intermediate pixel coordinates (for
779 *                       sequent distortions).  May be specified as a NULL
780 *                       pointer which is interpreted as (1,1,...).
781 *
782 *   pixtrc    const double[naxis]
783 *                       End of the range of pixel coordinates (prior) or
784 *                       intermediate pixel coordinates (sequent).
785 *
786 *   pixsamp   const double[naxis]
787 *                       If positive or zero, the increment on the particular
788 *                       axis, starting at pixblc[].  Zero is interpreted as a
789 *                       unit increment.  pixsamp may also be specified as a
790 *                       NULL pointer which is interpreted as all zeroes, i.e.
791 *                       unit increments on all axes.
792 *
793 *                       If negative, the grid size on the particular axis (the
794 *                       absolute value being rounded to the nearest integer).
795 *                       For example, if pixsamp is (-128.0,-128.0,...) then
796 *                       each axis will be sampled at 128 points between
797 *                       pixblc[] and pixtrc[] inclusive.  Use caution when
798 *                       using this option on non-square images.
799 *
800 * Returned:
801 *   nsamp     int*      The number of pixel coordinates sampled.
802 *
803 *                       Can be specified as a NULL pointer if not required.
804 *
805 *   maxdis    double[naxis]
806 *                       For each individual distortion function, the
807 *                       maximum absolute value of the distortion.
808 *
809 *                       Can be specified as a NULL pointer if not required.
810 *
811 *   maxtot    double*   For the combination of all distortion functions, the
812 *                       maximum absolute value of the distortion.
813 *
814 *                       Can be specified as a NULL pointer if not required.
815 *
816 *   avgdis    double[naxis]
817 *                       For each individual distortion function, the
818 *                       mean value of the distortion.
819 *
820 *                       Can be specified as a NULL pointer if not required.
821 *
822 *   avgtot    double*   For the combination of all distortion functions, the
823 *                       mean value of the distortion.
824 *
825 *                       Can be specified as a NULL pointer if not required.
826 *
827 *   rmsdis    double[naxis]
828 *                       For each individual distortion function, the
829 *                       root mean square deviation of the distortion.
830 *
831 *                       Can be specified as a NULL pointer if not required.
832 *
833 *   rmstot    double*   For the combination of all distortion functions, the
834 *                       root mean square deviation of the distortion.
835 *
836 *                       Can be specified as a NULL pointer if not required.
837 *
838 * Function return value:
839 *             int       Status return value:
840 *                         0: Success.
841 *                         1: Null disprm pointer passed.
842 *                         2: Memory allocation failed.
843 *                         3: Invalid parameter.
844 *                         4: Distort error.
845 *
846 *
847 * disprm struct - Distortion parameters
848 * -------------------------------------
849 * The disprm struct contains all of the information required to apply a set of
```

```
850 * distortion functions.  It consists of certain members that must be set by
851 * the user ("given") and others that are set by the WCSLIB routines
852 * ("returned").  While the addresses of the arrays themselves may be set by
853 * disinit() if it (optionally) allocates memory, their contents must be set by
854 * the user.
855 *
856 *   int flag
857 *     (Given and returned) This flag must be set to zero whenever any of the
858 *     following members of the disprm struct are set or modified:
859 *
860 *       - disprm::naxis,
861 *       - disprm::dtype,
862 *       - disprm::ndp,
863 *       - disprm::dp.
864 *
865 *     This signals the initialization routine, disset(), to recompute the
866 *     returned members of the disprm struct.  disset() will reset flag to
867 *     indicate that this has been done.
868 *
869 *     PLEASE NOTE: flag must be set to -1 when disinit() is called for the
870 *     first time for a particular disprm struct in order to initialize memory
871 *     management.  It must ONLY be used on the first initialization otherwise
872 *     memory leaks may result.
873 *
874 *   int naxis
875 *     (Given or returned) Number of pixel and world coordinate elements.
876 *
877 *     If disinit() is used to initialize the disprm struct (as would normally
878 *     be the case) then it will set naxis from the value passed to it as a
879 *     function argument.  The user should not subsequently modify it.
880 *
881 *   char (*dtype)[72]
882 *     (Given) Pointer to the first element of an array of char[72] containing
883 *     the name of the distortion function for each axis.
884 *
885 *   int ndp
886 *     (Given) The number of entries in the disprm::dp[] array.
887 *
888 *   int ndpmax
889 *     (Given) The length of the disprm::dp[] array.
890 *
891 *     ndpmax will be set by disinit() if it allocates memory for disprm::dp[],
892 *     otherwise it must be set by the user.  See also disndp().
893 *
894 *   struct dpkey dp
895 *     (Given) Address of the first element of an array of length ndpmax of
896 *     dpkey structs.
897 *
898 *     As a FITS header parser encounters each DPja or DQia keyword it should
899 *     load it into a dpkey struct in the array and increment ndp.  However,
900 *     note that a single disprm struct must hold only DPja or DQia keyvalues,
901 *     not both.  disset() interprets them as required by the particular
902 *     distortion function.
903 *
904 *   double *maxdis
905 *     (Given) Pointer to the first element of an array of double specifying
906 *     the maximum absolute value of the distortion for each axis computed over
907 *     the whole image.
908 *
909 *     It is not necessary to reset the disprm struct (via disset()) when
910 *     disprm::maxdis is changed.
911 *
912 *   double totdis
913 *     (Given) The maximum absolute value of the combination of all distortion
914 *     functions specified as an offset in pixel coordinates computed over the
915 *     whole image.
916 *
917 *     It is not necessary to reset the disprm struct (via disset()) when
918 *     disprm::totdis is changed.
919 *
920 *   int *docorr
921 *     (Returned) Pointer to the first element of an array of int containing
922 *     flags that indicate the mode of correction for each axis.
923 *
924 *     If docorr is zero, the distortion function returns the corrected
925 *     coordinates directly.  Any other value indicates that the distortion
926 *     function computes a correction to be added to pixel coordinates (prior
927 *     distortion) or intermediate pixel coordinates (sequent distortion).
928 *
929 *   int *Nhat
930 *     (Returned) Pointer to the first element of an array of int containing
931 *     the number of coordinate axes that form the independent variables of the
932 *     distortion function for each axis.
933 *
934 *   int **axmap
935 *     (Returned) Pointer to the first element of an array of int* containing
936 *     pointers to the first elements of the axis mapping arrays for each axis.
```

```
937 *
938 *      An axis mapping associates the independent variables of a distortion
939 *      function with the 0-relative image axis number.  For example, consider
940 *      an image with a spectrum on the first axis (axis 0), followed by RA
941 *      (axis 1), Dec (axis2), and time (axis 3) axes.  For a distortion in
942 *      (RA,Dec) and no distortion on the spectral or time axes, the axis
943 *      mapping arrays, axmap[j][], would be
944 *
945 =         j=0: [-1, -1, -1, -1]   ...no  distortion on spectral axis,
946 =           1: [ 1,  2, -1, -1]   ...RA  distortion depends on RA and Dec,
947 =           2: [ 2,  1, -1, -1]   ...Dec distortion depends on Dec and RA,
948 =           3: [-1, -1, -1, -1]   ...no  distortion on time axis,
949 *
950 *      where -1 indicates that there is no corresponding independent
951 *      variable.
952 *
953 *    double **offset
954 *      (Returned) Pointer to the first element of an array of double*
955 *      containing pointers to the first elements of arrays of offsets used to
956 *      renormalize the independent variables of the distortion function for
957 *      each axis.
958 *
959 *      The offsets are subtracted from the independent variables before
960 *      scaling.
961 *
962 *    double **scale
963 *      (Returned) Pointer to the first element of an array of double*
964 *      containing pointers to the first elements of arrays of scales used to
965 *      renormalize the independent variables of the distortion function for
966 *      each axis.
967 *
968 *      The scale is applied to the independent variables after the offsets are
969 *      subtracted.
970 *
971 *    int **iparm
972 *      (Returned) Pointer to the first element of an array of int*
973 *      containing pointers to the first elements of the arrays of integer
974 *      distortion parameters for each axis.
975 *
976 *    double **dparm
977 *      (Returned) Pointer to the first element of an array of double*
978 *      containing pointers to the first elements of the arrays of floating
979 *      point distortion parameters for each axis.
980 *
981 *    int i_naxis
982 *      (Returned) Dimension of the internal arrays (normally equal to naxis).
983 *
984 *    int ndis
985 *      (Returned) The number of distortion functions.
986 *
987 *    struct wcserr *err
988 *      (Returned) If enabled, when an error status is returned, this struct
989 *      contains detailed information about the error, see wcserr_enable().
990 *
991 *    int (**disp2x)(DISP2X_ARGS)
992 *      (For internal use only.)
993 *    int (**disx2p)(DISX2P_ARGS)
994 *      (For internal use only.)
995 *    double *tmpmem
996 *      (For internal use only.)
997 *    int m_flag
998 *      (For internal use only.)
999 *    int m_naxis
1000 *      (For internal use only.)
1001 *    char (*m_dtype)[72]
1002 *      (For internal use only.)
1003 *    double **m_dp
1004 *      (For internal use only.)
1005 *    double *m_maxdis
1006 *      (For internal use only.)
1007 *
1008 *
1009 * dpkey struct - Store for DPja and DQia keyvalues
1010 * ------------------------------------------------
1011 * The dpkey struct is used to pass the parsed contents of DPja or DQia
1012 * keyrecords to disset() via the disprm struct.  A disprm struct must hold
1013 * only DPja or DQia keyvalues, not both.
1014 *
1015 * All members of this struct are to be set by the user.
1016 *
1017 *    char field[72]
1018 *      (Given) The full field name of the record, including the keyword name.
1019 *      Note that the colon delimiter separating the field name and the value in
1020 *      record-valued keyvalues is not part of the field name.  For example, in
1021 *      the following:
1022 *
1023 =        DP3A = 'AXIS.1: 2'
```

```
1024 *
1025 *     the full record field name is "DP3A.AXIS.1", and the record's value
1026 *     is 2.
1027 *
1028 *   int j
1029 *     (Given) Axis number (1-relative), i.e. the j in DPja or i in DQia.
1030 *
1031 *   int type
1032 *     (Given) The data type of the record's value
1033 *       - 0: Integer (stored as an int),
1034 *       - 1: Floating point (stored as a double).
1035 *
1036 *   union value
1037 *     (Given) A union comprised of
1038 *       - dpkey::i,
1039 *       - dpkey::f,
1040 *
1041 *     the record's value.
1042 *
1043 *
1044 * Global variable: const char *dis_errmsg[] - Status return messages
1045 * ----------------------------------------------------------------
1046 * Error messages to match the status value returned from each function.
1047 *
1048 *===========================================================================*/
1049
1050 #ifndef WCSLIB_DIS
1051 #define WCSLIB_DIS
1052
1053 #ifdef __cplusplus
1054 extern "C" {
1055 #endif
1056
1057
1058 extern const char *dis_errmsg[];
1059
1060 enum dis_errmsg_enum {
1061   DISERR_SUCCESS      = 0,       // Success.
1062   DISERR_NULL_POINTER = 1,    // Null disprm pointer passed.
1063   DISERR_MEMORY       = 2,  // Memory allocation failed.
1064   DISERR_BAD_PARAM    = 3,      // Invalid parameter value.
1065   DISERR_DISTORT      = 4,        // Distortion error.
1066   DISERR_DEDISTORT    = 5       // De-distortion error.
1067 };
1068
1069 // For use in declaring distortion function prototypes (= DISX2P_ARGS).
1070 #define DISP2X_ARGS int inverse, const int iparm[], const double dparm[], \
1071 int ncrd, const double rawcrd[], double *discrd
1072
1073 // For use in declaring de-distortion function prototypes (= DISP2X_ARGS).
1074 #define DISX2P_ARGS int inverse, const int iparm[], const double dparm[], \
1075 int ncrd, const double discrd[], double *rawcrd
1076
1077
1078 // Struct used for storing DPja and DQia keyvalues.
1079 struct dpkey {
1080   char field[72];           // Full record field name (no colon).
1081   int j;                        // Axis number, as in DPja (1-relative).
1082   int type;                  // Data type of value.
1083   union {
1084     int    i;                    // Integer record value.
1085     double f;                    // Floating point record value.
1086   } value;                   // Record value.
1087 };
1088
1089 // Size of the dpkey struct in int units, used by the Fortran wrappers.
1090 #define DPLEN (sizeof(struct dpkey)/sizeof(int))
1091
1092
1093 struct disprm {
1094   // Initialization flag (see the prologue above).
1095   //-----------------------------------------------------------------------
1096   int flag;                 // Set to zero to force initialization.
1097
1098   // Parameters to be provided (see the prologue above).
1099   //-----------------------------------------------------------------------
1100   int naxis;                         // The number of pixel coordinate elements,
1101                                      // given by NAXIS.
1102   char   (*dtype)[72];          // For each axis, the distortion type.
1103   int    ndp;                        // Number of DPja or DQia keywords, and the
1104   int    ndpmax;           // number for which space was allocated.
1105   struct dpkey *dp;            // DPja or DQia keyvalues (not both).
1106   double *maxdis;          // For each axis, the maximum distortion.
1107   double totdis;           // The maximum combined distortion.
1108
1109   // Information derived from the parameters supplied.
1110   //-----------------------------------------------------------------------
```

```
1111    int    *docorr;            // For each axis, the mode of correction.
1112    int    *Nhat;                      // For each axis, the number of coordinate
1113                                       // axes that form the independent variables
1114                                       // of the distortion function.
1115    int    **axmap;            // For each axis, the axis mapping array.
1116    double **offset;                   // For each axis, renormalization offsets.
1117    double **scale;            // For each axis, renormalization scales.
1118    int    **iparm;            // For each axis, the array of integer
1119                                       // distortion parameters.
1120    double **dparm;            // For each axis, the array of floating
1121                                       // point distortion parameters.
1122    int    i_naxis;                    // Dimension of the internal arrays.
1123    int    ndis;                          // The number of distortion functions.
1124
1125    // Error handling, if enabled.
1126    //------------------------------------------------------------------------
1127    struct wcserr *err;
1128
1129    // Private - the remainder are for internal use.
1130    //------------------------------------------------------------------------
1131    int (**disp2x)(DISP2X_ARGS); // For each axis, pointers to the
1132    int (**disx2p)(DISX2P_ARGS); // distortion function and its inverse.
1133
1134    double *tmpmem;
1135
1136    int    m_flag, m_naxis;  // The remainder are for memory management.
1137    char   (*m_dtype)[72];
1138    struct dpkey *m_dp;
1139    double *m_maxdis;
1140 };
1141
1142 // Size of the disprm struct in int units, used by the Fortran wrappers.
1143 #define DISLEN (sizeof(struct disprm)/sizeof(int))
1144
1145
1146 int disndp(int n);
1147
1148 int dpfill(struct dpkey *dp, const char *keyword, const char *field, int j,
1149           int type, int i, double f);
1150
1151 int    dpkeyi(const struct dpkey *dp);
1152
1153 double dpkeyd(const struct dpkey *dp);
1154
1155 int disini(int alloc, int naxis, struct disprm *dis);
1156
1157 int disinit(int alloc, int naxis, struct disprm *dis, int ndpmax);
1158
1159 int discpy(int alloc, const struct disprm *dissrc, struct disprm *disdst);
1160
1161 int disfree(struct disprm *dis);
1162
1163 int dissize(const struct disprm *dis, int sizes[2]);
1164
1165 int disprt(const struct disprm *dis);
1166
1167 int disperr(const struct disprm *dis, const char *prefix);
1168
1169 int dishdo(struct disprm *dis);
1170
1171 int disset(struct disprm *dis);
1172
1173 int disp2x(struct disprm *dis, const double rawcrd[], double discrd[]);
1174
1175 int disx2p(struct disprm *dis, const double discrd[], double rawcrd[]);
1176
1177 int diswarp(struct disprm *dis, const double pixblc[], const double pixtrc[],
1178            const double pixsamp[], int *nsamp,
1179            double maxdis[], double *maxtot,
1180            double avgdis[], double *avgtot,
1181            double rmsdis[], double *rmstot);
1182
1183 #ifdef __cplusplus
1184 }
1185 #endif
1186
1187 #endif // WCSLIB_DIS
```

19.5   fitshdr.h File Reference

```
#include "wcsconfig.h"
```

Data Structures

- struct fitskeyid

    Keyword indexing.

- struct fitskey

    Keyword/value information.

Macros

- #define FITSHDR_KEYWORD 0x01

    Flag bit indicating illegal keyword syntax.

- #define FITSHDR_KEYVALUE 0x02

    Flag bit indicating illegal keyvalue syntax.

- #define FITSHDR_COMMENT 0x04

    Flag bit indicating illegal keycomment syntax.

- #define FITSHDR_KEYREC 0x08

    Flag bit indicating illegal keyrecord.

- #define FITSHDR_CARD 0x08

    Deprecated.

- #define FITSHDR_TRAILER 0x10

    Flag bit indicating keyrecord following a valid END keyrecord.

- #define KEYIDLEN (sizeof(struct fitskeyid)/sizeof(int))
- #define KEYLEN (sizeof(struct fitskey)/sizeof(int))

Typedefs

- typedef int int64[3]

    64-bit signed integer data type.

Enumerations

- enum fitshdr_errmsg_enum {
  FITSHDRERR_SUCCESS = 0 , FITSHDRERR_NULL_POINTER = 1 , FITSHDRERR_MEMORY = 2 ,
  FITSHDRERR_FLEX_PARSER = 3 ,
  FITSHDRERR_DATA_TYPE = 4 }

Functions

- int fitshdr (const char header[ ], int nkeyrec, int nkeyids, struct fitskeyid keyids[ ], int ∗nreject, struct fitskey ∗∗keys)

    FITS header parser routine.

Variables

- const char ∗ fitshdr_errmsg [ ]

    Status return messages.

19.5.1 Detailed Description

The Flexible Image Transport System (FITS), is a data format widely used in astronomy for data interchange and archive. It is described in

```
"Definition of the Flexible Image Transport System (FITS), version 3.0",
Pence, W.D., Chiappetti, L., Page, C.G., Shaw, R.A., & Stobie, E. 2010,
A&A, 524, A42 - http://dx.doi.org/10.1051/0004-6361/201015362
```

See also http:
fitshdr() is a generic FITS header parser provided to handle keyrecords that are ignored by the WCS header parsers, wcspih() and wcsbth(). Typically the latter may be set to remove WCS keyrecords from a header leaving fitshdr() to handle the remainder.

19.5.2 Macro Definition Documentation

19.5.2.1 FITSHDR_KEYWORD `#define FITSHDR_KEYWORD 0x01`

Bit mask for the status flag bit-vector returned by fitshdr() indicating illegal keyword syntax.

19.5.2.2 FITSHDR_KEYVALUE `#define FITSHDR_KEYVALUE 0x02`

Bit mask for the status flag bit-vector returned by fitshdr() indicating illegal keyvalue syntax.

19.5.2.3 FITSHDR_COMMENT `#define FITSHDR_COMMENT 0x04`

Bit mask for the status flag bit-vector returned by fitshdr() indicating illegal keycomment syntax.

19.5.2.4 FITSHDR_KEYREC `#define FITSHDR_KEYREC 0x08`

Bit mask for the status flag bit-vector returned by fitshdr() indicating an illegal keyrecord, e.g. an END keyrecord with trailing text.

19.5.2.5 FITSHDR_CARD `#define FITSHDR_CARD 0x08`

Deprecated Added for backwards compatibility, use *FITSHDR_KEYREC* instead.

19.5.2.6 FITSHDR_TRAILER `#define FITSHDR_TRAILER 0x10`

Bit mask for the status flag bit-vector returned by fitshdr() indicating a keyrecord following a valid END keyrecord.

19.5.2.7 KEYIDLEN `#define KEYIDLEN (sizeof(struct fitskeyid)/sizeof(int))`

19.5.2.8 KEYLEN `#define KEYLEN (sizeof(struct fitskey)/sizeof(int))`

19.5.3 Typedef Documentation

19.5.3.1 int64 `int64`

64-bit signed integer data type defined via preprocessor macro WCSLIB_INT64 which may be defined in wcsconfig.h. For example

```
#define WCSLIB_INT64 long long int
```

This is typedef'd in fitshdr.h as

```
#ifdef WCSLIB_INT64
  typedef WCSLIB_INT64 int64;
#else
  typedef int int64[3];
#endif
```

See fitskey::type.

19.5.4 Enumeration Type Documentation

19.5.4.1 fitshdr_errmsg_enum `enum fitshdr_errmsg_enum`

Enumerator

FITSHDRERR_SUCCESS	
FITSHDRERR_NULL_POINTER	
FITSHDRERR_MEMORY	
FITSHDRERR_FLEX_PARSER	
FITSHDRERR_DATA_TYPE	

19.5.5 Function Documentation

19.5.5.1 fitshdr() `int fitshdr (`
```
        const char header[],
        int nkeyrec,
        int nkeyids,
        struct fitskeyid keyids[],
        int * nreject,
        struct fitskey ** keys )
```

fitshdr() parses a character array containing a FITS header, extracting all keywords and their values into an array of fitskey structs.

Parameters

in	*header*	Character array containing the (entire) FITS header, for example, as might be obtained conveniently via the CFITSIO routine fits_hdr2str().
		Each header "keyrecord" (formerly "card image") consists of exactly 80 7-bit ASCII printing characters in the range 0x20 to 0x7e (which excludes NUL, BS, TAB, LF, FF and CR) especially noting that the keyrecords are NOT null-terminated.
in	*nkeyrec*	Number of keyrecords in header[].
in	*nkeyids*	Number of entries in keyids[].
in,out	*keyids*	While all keywords are extracted from the header, keyids[] provides a convienient way of indexing them. The fitskeyid struct contains three members; fitskeyid::name must be set by the user while fitskeyid::count and fitskeyid::idx are returned by **fitshdr**(). All matched keywords will have their fitskey::keyno member negated.
out	*nreject*	Number of header keyrecords rejected for syntax errors.
out	*keys*	Pointer to an array of nkeyrec fitskey structs containing all keywords and keyvalues extracted from the header.
		Memory for the array is allocated by **fitshdr**() and this must be freed by the user. See wcsdealloc().

Returns

Status return value:

- 0: Success.
- 1: Null fitskey pointer passed.
- 2: Memory allocation failed.
- 3: Fatal error returned by Flex parser.
- 4: Unrecognised data type.

Notes:

1. Keyword parsing is done in accordance with the syntax defined by NOST 100-2.0, noting the following points in particular:

   a Sect. 5.1.2.1 specifies that keywords be left-justified in columns 1-8, blank-filled with no embedded spaces, composed only of the ASCII characters **ABCDEFGHJKLMNOPQRSTUVWXYZ0123456789−**↩
   _
   fitshdr() accepts any characters in columns 1-8 but flags keywords that do not conform to standard syntax.

   b Sect. 5.1.2.2 defines the "value indicator" as the characters "**=** " occurring in columns 9 and 10. If these are absent then the keyword has no value and columns 9-80 may contain any ASCII text (but see note 2 for **CONTINUE** keyrecords). This is copied to the comment member of the fitskey struct.

   c Sect. 5.1.2.3 states that a keyword may have a null (undefined) value if the value/comment field, columns 11-80, consists entirely of spaces, possibly followed by a comment.

   d Sect. 5.1.1 states that trailing blanks in a string keyvalue are not significant and the parser always removes them. A string containing nothing but blanks will be replaced with a single blank.
   Sect. 5.2.1 also states that a quote character (**'**) in a string value is to be represented by two successive quote characters and the parser removes the repeated quote.

   e The parser recognizes free-format character (NOST 100-2.0, Sect. 5.2.1), integer (Sect. 5.2.3), and floating-point values (Sect. 5.2.4) for all keywords.

f Sect. 5.2.3 offers no comment on the size of an integer keyvalue except indirectly in limiting it to 70 digits. The parser will translate an integer keyvalue to a 32-bit signed integer if it lies in the range -2147483648 to +2147483647, otherwise it interprets it as a 64-bit signed integer if possible, or else a "very long" integer (see fitskey::type).

g **END** not followed by 77 blanks is not considered to be a legitimate end keyrecord.

2. The parser supports a generalization of the OGIP Long String Keyvalue Convention (v1.0) whereby strings may be continued onto successive header keyrecords. A keyrecord contains a segment of a continued string if and only if

a it contains the pseudo-keyword **CONTINUE**,

b columns 9 and 10 are both blank,

c columns 11 to 80 contain what would be considered a valid string keyvalue, including optional key-comment, if column 9 had contained '**=**',

d the previous keyrecord contained either a valid string keyvalue or a valid **CONTINUE** keyrecord.

If any of these conditions is violated, the keyrecord is considered in isolation.

Syntax errors in keycomments in a continued string are treated more permissively than usual; the '**/**' delimiter may be omitted provided that parsing of the string keyvalue is not compromised. However, the FITSHDR_↩COMMENT status bit will be set for the keyrecord (see fitskey::status).

As for normal strings, trailing blanks in a continued string are not significant.

In the OGIP convention "the '**&**' character is used as the last non-blank character of the string to indicate that the string is (probably) continued on the following keyword". This additional syntax is not required by **fitshdr**(), but if '**&**' does occur as the last non-blank character of a continued string keyvalue then it will be removed, along with any trailing blanks. However, blanks that occur before the '**&**' will be preserved.

19.5.6 Variable Documentation

19.5.6.1 fitshdr_errmsg `const char * fitshdr_errmsg[] [extern]`

Error messages to match the status value returned from each function.

19.6 fitshdr.h

Go to the documentation of this file.
```
1  /*============================================================================
2    WCSLIB 7.11 - an implementation of the FITS WCS standard.
3    Copyright (C) 1995-2022, Mark Calabretta
4
5    This file is part of WCSLIB.
6
7    WCSLIB is free software: you can redistribute it and/or modify it under the
8    terms of the GNU Lesser General Public License as published by the Free
9    Software Foundation, either version 3 of the License, or (at your option)
10   any later version.
11
12   WCSLIB is distributed in the hope that it will be useful, but WITHOUT ANY
13   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
14   FOR A PARTICULAR PURPOSE.  See the GNU Lesser General Public License for
15   more details.
16
17   You should have received a copy of the GNU Lesser General Public License
18   along with WCSLIB.  If not, see http://www.gnu.org/licenses.
19
20   Author: Mark Calabretta, Australia Telescope National Facility, CSIRO.
21   http://www.atnf.csiro.au/people/Mark.Calabretta
22   $Id: fitshdr.h,v 7.11 2022/04/26 06:13:52 mcalabre Exp $
23  *============================================================================
```

```
 24 *
 25 * WCSLIB 7.11 - C routines that implement the FITS World Coordinate System
 26 * (WCS) standard.  Refer to the README file provided with WCSLIB for an
 27 * overview of the library.
 28 *
 29 *
 30 * Summary of the fitshdr routines
 31 * -------------------------------
 32 * The Flexible Image Transport System (FITS), is a data format widely used in
 33 * astronomy for data interchange and archive.  It is described in
 34 *
 35 =   "Definition of the Flexible Image Transport System (FITS), version 3.0",
 36 =   Pence, W.D., Chiappetti, L., Page, C.G., Shaw, R.A., & Stobie, E. 2010,
 37 =   A&A, 524, A42 - http://dx.doi.org/10.1051/0004-6361/201015362
 38 *
 39 * See also http://fits.gsfc.nasa.gov
 40 *
 41 * fitshdr() is a generic FITS header parser provided to handle keyrecords that
 42 * are ignored by the WCS header parsers, wcspih() and wcsbth().  Typically the
 43 * latter may be set to remove WCS keyrecords from a header leaving fitshdr()
 44 * to handle the remainder.
 45 *
 46 *
 47 * fitshdr() - FITS header parser routine
 48 * --------------------------------------
 49 * fitshdr() parses a character array containing a FITS header, extracting
 50 * all keywords and their values into an array of fitskey structs.
 51 *
 52 * Given:
 53 *   header    const char []
 54 *                       Character array containing the (entire) FITS header,
 55 *                       for example, as might be obtained conveniently via the
 56 *                       CFITSIO routine fits_hdr2str().
 57 *
 58 *                       Each header "keyrecord" (formerly "card image")
 59 *                       consists of exactly 80 7-bit ASCII printing characters
 60 *                       in the range 0x20 to 0x7e (which excludes NUL, BS,
 61 *                       TAB, LF, FF and CR) especially noting that the
 62 *                       keyrecords are NOT null-terminated.
 63 *
 64 *   nkeyrec   int       Number of keyrecords in header[].
 65 *
 66 *   nkeyids   int       Number of entries in keyids[].
 67 *
 68 * Given and returned:
 69 *   keyids    struct fitskeyid []
 70 *                       While all keywords are extracted from the header,
 71 *                       keyids[] provides a convienient way of indexing them.
 72 *                       The fitskeyid struct contains three members;
 73 *                       fitskeyid::name must be set by the user while
 74 *                       fitskeyid::count and fitskeyid::idx are returned by
 75 *                       fitshdr().  All matched keywords will have their
 76 *                       fitskey::keyno member negated.
 77 *
 78 * Returned:
 79 *   nreject   int*      Number of header keyrecords rejected for syntax
 80 *                       errors.
 81 *
 82 *   keys      struct fitskey**
 83 *                       Pointer to an array of nkeyrec fitskey structs
 84 *                       containing all keywords and keyvalues extracted from
 85 *                       the header.
 86 *
 87 *                       Memory for the array is allocated by fitshdr() and
 88 *                       this must be freed by the user.  See wcsdealloc().
 89 *
 90 * Function return value:
 91 *             int       Status return value:
 92 *                         0: Success.
 93 *                         1: Null fitskey pointer passed.
 94 *                         2: Memory allocation failed.
 95 *                         3: Fatal error returned by Flex parser.
 96 *                         4: Unrecognised data type.
 97 *
 98 * Notes:
 99 *   1: Keyword parsing is done in accordance with the syntax defined by
100 *      NOST 100-2.0, noting the following points in particular:
101 *
102 *      a: Sect. 5.1.2.1 specifies that keywords be left-justified in columns
103 *         1-8, blank-filled with no embedded spaces, composed only of the
104 *         ASCII characters ABCDEFGHJKLMNOPQRSTUVWXYZ0123456789-_
105 *
106 *         fitshdr() accepts any characters in columns 1-8 but flags keywords
107 *         that do not conform to standard syntax.
108 *
109 *      b: Sect. 5.1.2.2 defines the "value indicator" as the characters "= "
110 *         occurring in columns 9 and 10.  If these are absent then the
```

```
111 *        keyword has no value and columns 9-80 may contain any ASCII text
112 *        (but see note 2 for CONTINUE keyrecords).  This is copied to the
113 *        comment member of the fitskey struct.
114 *
115 *      c: Sect. 5.1.2.3 states that a keyword may have a null (undefined)
116 *        value if the value/comment field, columns 11-80, consists entirely
117 *        of spaces, possibly followed by a comment.
118 *
119 *      d: Sect. 5.1.1 states that trailing blanks in a string keyvalue are
120 *        not significant and the parser always removes them.  A string
121 *        containing nothing but blanks will be replaced with a single
122 *        blank.
123 *
124 *        Sect. 5.2.1 also states that a quote character (') in a string
125 *        value is to be represented by two successive quote characters and
126 *        the parser removes the repeated quote.
127 *
128 *      e: The parser recognizes free-format character (NOST 100-2.0,
129 *        Sect. 5.2.1), integer (Sect. 5.2.3), and floating-point values
130 *        (Sect. 5.2.4) for all keywords.
131 *
132 *      f: Sect. 5.2.3 offers no comment on the size of an integer keyvalue
133 *        except indirectly in limiting it to 70 digits.  The parser will
134 *        translate an integer keyvalue to a 32-bit signed integer if it
135 *        lies in the range -2147483648 to +2147483647, otherwise it
136 *        interprets it as a 64-bit signed integer if possible, or else a
137 *        "very long" integer (see fitskey::type).
138 *
139 *      g: END not followed by 77 blanks is not considered to be a legitimate
140 *        end keyrecord.
141 *
142 *   2: The parser supports a generalization of the OGIP Long String Keyvalue
143 *      Convention (v1.0) whereby strings may be continued onto successive
144 *      header keyrecords.  A keyrecord contains a segment of a continued
145 *      string if and only if
146 *
147 *      a: it contains the pseudo-keyword CONTINUE,
148 *
149 *      b: columns 9 and 10 are both blank,
150 *
151 *      c: columns 11 to 80 contain what would be considered a valid string
152 *        keyvalue, including optional keycomment, if column 9 had contained
153 *        '=',
154 *
155 *      d: the previous keyrecord contained either a valid string keyvalue or
156 *        a valid CONTINUE keyrecord.
157 *
158 *      If any of these conditions is violated, the keyrecord is considered in
159 *      isolation.
160 *
161 *      Syntax errors in keycomments in a continued string are treated more
162 *      permissively than usual; the '/' delimiter may be omitted provided that
163 *      parsing of the string keyvalue is not compromised.  However, the
164 *      FITSHDR_COMMENT status bit will be set for the keyrecord (see
165 *      fitskey::status).
166 *
167 *      As for normal strings, trailing blanks in a continued string are not
168 *      significant.
169 *
170 *      In the OGIP convention "the '&' character is used as the last non-blank
171 *      character of the string to indicate that the string is (probably)
172 *      continued on the following keyword".  This additional syntax is not
173 *      required by fitshdr(), but if '&' does occur as the last non-blank
174 *      character of a continued string keyvalue then it will be removed, along
175 *      with any trailing blanks.  However, blanks that occur before the '&'
176 *      will be preserved.
177 *
178 *
179 * fitskeyid struct - Keyword indexing
180 * -----------------------------------
181 * fitshdr() uses the fitskeyid struct to return indexing information for
182 * specified keywords.  The struct contains three members, the first of which,
183 * fitskeyid::name, must be set by the user with the remainder returned by
184 * fitshdr().
185 *
186 *   char name[12]:
187 *     (Given) Name of the required keyword.  This is to be set by the user;
188 *     the '.' character may be used for wildcarding.  Trailing blanks will be
189 *     replaced with nulls.
190 *
191 *   int count:
192 *     (Returned) The number of matches found for the keyword.
193 *
194 *   int idx[2]:
195 *     (Returned) Indices into keys[], the array of fitskey structs returned by
196 *     fitshdr().  Note that these are 0-relative array indices, not keyrecord
197 *     numbers.
```

```
198 *
199 *      If the keyword is found in the header the first index will be set to the
200 *      array index of its first occurrence, otherwise it will be set to -1.
201 *
202 *      If multiples of the keyword are found, the second index will be set to
203 *      the array index of its last occurrence, otherwise it will be set to -1.
204 *
205 *
206 * fitskey struct - Keyword/value information
207 * ------------------------------------------
208 * fitshdr() returns an array of fitskey structs, each of which contains the
209 * result of parsing one FITS header keyrecord.  All members of the fitskey
210 * struct are returned by fitshdr(), none are given by the user.
211 *
212 *   int keyno
213 *      (Returned) Keyrecord number (1-relative) in the array passed as input to
214 *      fitshdr().  This will be negated if the keyword matched any specified in
215 *      the keyids[] index.
216 *
217 *   int keyid
218 *      (Returned) Index into the first entry in keyids[] with which the
219 *      keyrecord matches, else -1.
220 *
221 *   int status
222 *      (Returned) Status flag bit-vector for the header keyrecord employing the
223 *      following bit masks defined as preprocessor macros:
224 *
225 *        - FITSHDR_KEYWORD:   Illegal keyword syntax.
226 *        - FITSHDR_KEYVALUE:  Illegal keyvalue syntax.
227 *        - FITSHDR_COMMENT:   Illegal keycomment syntax.
228 *        - FITSHDR_KEYREC:    Illegal keyrecord, e.g. an END keyrecord with
229 *                             trailing text.
230 *        - FITSHDR_TRAILER:   Keyrecord following a valid END keyrecord.
231 *
232 *      The header keyrecord is syntactically correct if no bits are set.
233 *
234 *   char keyword[12]
235 *      (Returned) Keyword name, null-filled for keywords of less than eight
236 *      characters (trailing blanks replaced by nulls).
237 *
238 *      Use
239 *
240 =        sprintf(dst, "%.8s", keyword)
241 *
242 *      to copy it to a character array with null-termination, or
243 *
244 =        sprintf(dst, "%8.8s", keyword)
245 *
246 *      to blank-fill to eight characters followed by null-termination.
247 *
248 *   int type
249 *      (Returned) Keyvalue data type:
250 *        - 0: No keyvalue (both the value and type are undefined).
251 *        - 1: Logical, represented as int.
252 *        - 2: 32-bit signed integer.
253 *        - 3: 64-bit signed integer (see below).
254 *        - 4: Very long integer (see below).
255 *        - 5: Floating point (stored as double).
256 *        - 6: Integer complex (stored as double[2]).
257 *        - 7: Floating point complex (stored as double[2]).
258 *        - 8: String.
259 *        - 8+10*n: Continued string (described below and in fitshdr() note 2).
260 *
261 *      A negative type indicates that a syntax error was encountered when
262 *      attempting to parse a keyvalue of the particular type.
263 *
264 *      Comments on particular data types:
265 *        - 64-bit signed integers lie in the range
266 *
267 =            (-9223372036854775808 <= int64 <  -2147483648) ||
268 =                    (+2147483647 <  int64 <= +9223372036854775807)
269 *
270 *          A native 64-bit data type may be defined via preprocessor macro
271 *          WCSLIB_INT64 defined in wcsconfig.h, e.g. as 'long long int'; this
272 *          will be typedef'd to 'int64' here.  If WCSLIB_INT64 is not set, then
273 *          int64 is typedef'd to int[3] instead and fitskey::keyvalue is to be
274 *          computed as
275 *
276 =            ((keyvalue.k[2]) * 1000000000 +
277 =              keyvalue.k[1]) * 1000000000 +
278 =              keyvalue.k[0]
279 *
280 *          and may reported via
281 *
282 =              if (keyvalue.k[2]) {
283 =                printf("%d%09d%09d", keyvalue.k[2], abs(keyvalue.k[1]),
284 =                                     abs(keyvalue.k[0]));
```

```
285 =                } else {
286 =                  printf("%d%09d", keyvalue.k[1], abs(keyvalue.k[0]));
287 =                }
288 *
289 *          where keyvalue.k[0] and keyvalue.k[1] range from -999999999 to
290 *          +999999999.
291 *
292 *        - Very long integers, up to 70 decimal digits in length, are encoded
293 *          in keyvalue.l as an array of int[8], each of which stores 9 decimal
294 *          digits.  fitskey::keyvalue is to be computed as
295 *
296 =              ((((((((keyvalue.l[7]) * 1000000000 +
297 =                      keyvalue.l[6]) * 1000000000 +
298 =                      keyvalue.l[5]) * 1000000000 +
299 =                      keyvalue.l[4]) * 1000000000 +
300 =                      keyvalue.l[3]) * 1000000000 +
301 =                      keyvalue.l[2]) * 1000000000 +
302 =                      keyvalue.l[1]) * 1000000000 +
303 =                      keyvalue.l[0]
304 *
305 *        - Continued strings are not reconstructed, they remain split over
306 *          successive fitskey structs in the keys[] array returned by
307 *          fitshdr().  fitskey::keyvalue data type, 8 + 10n, indicates the
308 *          segment number, n, in the continuation.
309 *
310 *    int padding
311 *      (An unused variable inserted for alignment purposes only.)
312 *
313 *    union keyvalue
314 *      (Returned) A union comprised of
315 *
316 *        - fitskey::i,
317 *        - fitskey::k,
318 *        - fitskey::l,
319 *        - fitskey::f,
320 *        - fitskey::c,
321 *        - fitskey::s,
322 *
323 *      used by the fitskey struct to contain the value associated with a
324 *      keyword.
325 *
326 *    int i
327 *      (Returned) Logical (fitskey::type == 1) and 32-bit signed integer
328 *      (fitskey::type == 2) data types in the fitskey::keyvalue union.
329 *
330 *    int64 k
331 *      (Returned) 64-bit signed integer (fitskey::type == 3) data type in the
332 *      fitskey::keyvalue union.
333 *
334 *    int l[8]
335 *      (Returned) Very long integer (fitskey::type == 4) data type in the
336 *      fitskey::keyvalue union.
337 *
338 *    double f
339 *      (Returned) Floating point (fitskey::type == 5) data type in the
340 *      fitskey::keyvalue union.
341 *
342 *    double c[2]
343 *      (Returned) Integer and floating point complex (fitskey::type == 6 || 7)
344 *      data types in the fitskey::keyvalue union.
345 *
346 *    char s[72]
347 *      (Returned) Null-terminated string (fitskey::type == 8) data type in the
348 *      fitskey::keyvalue union.
349 *
350 *    int ulen
351 *      (Returned) Where a keycomment contains a units string in the standard
352 *      form, e.g. [m/s], the ulen member indicates its length, inclusive of
353 *      square brackets.  Otherwise ulen is zero.
354 *
355 *    char comment[84]
356 *      (Returned) Keycomment, i.e. comment associated with the keyword or, for
357 *      keyrecords rejected because of syntax errors, the compete keyrecord
358 *      itself with null-termination.
359 *
360 *      Comments are null-terminated with trailing spaces removed.  Leading
361 *      spaces are also removed from keycomments (i.e. those immediately
362 *      following the '/' character), but not from COMMENT or HISTORY keyrecords
363 *      or keyrecords without a value indicator ("= " in columns 9-80).
364 *
365 *
366 * Global variable: const char *fitshdr_errmsg[] - Status return messages
367 * -------------------------------------------------------------------
368 * Error messages to match the status value returned from each function.
369 *
370 *=========================================================================*/
371
```

```
372 #ifndef WCSLIB_FITSHDR
373 #define WCSLIB_FITSHDR
374
375 #include "wcsconfig.h"
376
377 #ifdef __cplusplus
378 extern "C" {
379 #endif
380
381 #define FITSHDR_KEYWORD  0x01
382 #define FITSHDR_KEYVALUE 0x02
383 #define FITSHDR_COMMENT  0x04
384 #define FITSHDR_KEYREC   0x08
385 #define FITSHDR_CARD     0x08   // Alias for backwards compatibility.
386 #define FITSHDR_TRAILER  0x10
387
388
389 extern const char *fitshdr_errmsg[];
390
391 enum fitshdr_errmsg_enum {
392   FITSHDRERR_SUCCESS      = 0,        // Success.
393   FITSHDRERR_NULL_POINTER = 1,   // Null fitskey pointer passed.
394   FITSHDRERR_MEMORY       = 2, // Memory allocation failed.
395   FITSHDRERR_FLEX_PARSER  = 3,    // Fatal error returned by Flex parser.
396   FITSHDRERR_DATA_TYPE    = 4      // Unrecognised data type.
397 };
398
399 #ifdef WCSLIB_INT64
400   typedef WCSLIB_INT64 int64;
401 #else
402   typedef int int64[3];
403 #endif
404
405
406 // Struct used for indexing the keywords.
407 struct fitskeyid {
408   char name[12];            // Keyword name, null-terminated.
409   int  count;                    // Number of occurrences of keyword.
410   int  idx[2];                     // Indices into fitskey array.
411 };
412
413 // Size of the fitskeyid struct in int units, used by the Fortran wrappers.
414 #define KEYIDLEN (sizeof(struct fitskeyid)/sizeof(int))
415
416
417 // Struct used for storing FITS keywords.
418 struct fitskey {
419   int  keyno;                      // Header keyrecord sequence number (1-rel).
420   int  keyid;                      // Index into fitskeyid[].
421   int  status;                   // Header keyrecord status bit flags.
422   char keyword[12];           // Keyword name, null-filled.
423   int  type;                       // Keyvalue type (see above).
424   int  padding;                  // (Dummy inserted for alignment purposes.)
425   union {
426     int    i;                  // 32-bit integer and logical values.
427     int64  k;                    // 64-bit integer values.
428     int    l[8];               // Very long signed integer values.
429     double f;                  // Floating point values.
430     double c[2];               // Complex values.
431     char   s[72];              // String values, null-terminated.
432   } keyvalue;                    // Keyvalue.
433   int  ulen;                       // Length of units string.
434   char comment[84];              // Comment (or keyrecord), null-terminated.
435 };
436
437 // Size of the fitskey struct in int units, used by the Fortran wrappers.
438 #define KEYLEN (sizeof(struct fitskey)/sizeof(int))
439
440
441 int fitshdr(const char header[], int nkeyrec, int nkeyids,
442            struct fitskeyid keyids[], int *nreject, struct fitskey **keys);
443
444
445 #ifdef __cplusplus
446 }
447 #endif
448
449 #endif // WCSLIB_FITSHDR
```

19.7 getwcstab.h File Reference

```
#include <fitsio.h>
```

Data Structures

- struct wtbarr

    Extraction of coordinate lookup tables from BINTABLE.

Functions

- int fits_read_wcstab (fitsfile ∗fptr, int nwtb, wtbarr ∗wtb, int ∗status)

    *FITS '**TAB**' table reading routine.*

19.7.1 Detailed Description

fits_read_wcstab(), an implementation of a FITS table reading routine for '**TAB**' coordinates, is provided for CFITSIO programmers. It has been incorporated into CFITSIO as of v3.006 with the definitions in this file, getwcstab.h, moved into fitsio.h.

fits_read_wcstab() is not included in the WCSLIB object library but the source code is presented here as it may be useful for programmers using an older version of CFITSIO than 3.006, or as a programming template for non-↩CFITSIO programmers.

19.7.2 Function Documentation

19.7.2.1 fits_read_wcstab() ```int fits_read_wcstab (```
```
        fitsfile * fptr,
        int nwtb,
        wtbarr * wtb,
        int * status )
```

fits_read_wcstab() extracts arrays from a binary table required in constructing '**TAB**' coordinates.

Parameters

in	*fptr*	Pointer to the file handle returned, for example, by the fits_open_file() routine in CFITSIO.
in	*nwtb*	Number of arrays to be read from the binary table(s).
in, out	*wtb*	Address of the first element of an array of wtbarr typedefs. This wtbarr typedef is defined to match the wtbarr struct defined in WCSLIB. An array of such structs returned by the WCSLIB function wcstab() as discussed in the notes below.
out	*status*	CFITSIO status value.

Returns

CFITSIO status value.

Notes:

1. In order to maintain WCSLIB and CFITSIO as independent libraries it is not permissible for any CFITSIO library code to include WCSLIB header files, or vice versa. However, the CFITSIO function **fits_read_↩ wcstab**() accepts an array of wtbarr structs defined in wcs.h within WCSLIB.

The problem therefore is to define the wtbarr struct within fitsio.h without including wcs.h, especially noting that wcs.h will often (but not always) be included together with fitsio.h in an applications program that uses **fits_read_wcstab**().

The solution adopted is for WCSLIB to define "struct wtbarr" while fitsio.h defines "typedef wtbarr" as an untagged struct with identical members. This allows both wcs.h and fitsio.h to define a wtbarr data type without conflict by virtue of the fact that structure tags and typedef names share different name spaces in C; Appendix A, Sect. A11.1 (p227) of the K&R ANSI edition states that:

```
Identifiers fall into several name spaces that do not interfere with
one another; the same identifier may be used for different purposes,
even in the same scope, if the uses are in different name spaces.
These classes are: objects, functions, typedef names, and enum
constants; labels; tags of structures, unions, and enumerations; and
members of each structure or union individually.
```

Therefore, declarations within WCSLIB look like
```
struct wtbarr *w;
```

while within CFITSIO they are simply
```
wtbarr *w;
```

As suggested by the commonality of the names, these are really the same aggregate data type. However, in passing a (struct wtbarr ∗) to **fits_read_wcstab**() a cast to (wtbarr ∗) is formally required.

When using WCSLIB and CFITSIO together in C++ the situation is complicated by the fact that typedefs and structs share the same namespace; C++ Annotated Reference Manual, Sect. 7.1.3 (p105). In that case the wtbarr struct in wcs.h is renamed by preprocessor macro substitution to wtbarr_s to distinguish it from the typedef defined in fitsio.h. However, the scope of this macro substitution is limited to wcs.h itself and CFITSIO programmer code, whether in C++ or C, should always use the wtbarr typedef.

19.8 getwcstab.h

Go to the documentation of this file.
```
1  /*=============================================================================
2    WCSLIB 7.11 - an implementation of the FITS WCS standard.
3    Copyright (C) 1995-2022, Mark Calabretta
4
5    This file is part of WCSLIB.
6
7    WCSLIB is free software: you can redistribute it and/or modify it under the
8    terms of the GNU Lesser General Public License as published by the Free
9    Software Foundation, either version 3 of the License, or (at your option)
10   any later version.
11
12   WCSLIB is distributed in the hope that it will be useful, but WITHOUT ANY
13   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
14   FOR A PARTICULAR PURPOSE.  See the GNU Lesser General Public License for
15   more details.
16
17   You should have received a copy of the GNU Lesser General Public License
18   along with WCSLIB.  If not, see http://www.gnu.org/licenses.
19
20   Author: Mark Calabretta, Australia Telescope National Facility, CSIRO.
21   http://www.atnf.csiro.au/people/Mark.Calabretta
22   $Id: getwcstab.h,v 7.11 2022/04/26 06:13:52 mcalabre Exp $
23  *=============================================================================
24  *
25  * WCSLIB 7.11 - C routines that implement the FITS World Coordinate System
26  * (WCS) standard.  Refer to the README file provided with WCSLIB for an
27  * overview of the library.
28  *
29  * Summary of the getwcstab routines
30  * ---------------------------------
31  * fits_read_wcstab(), an implementation of a FITS table reading routine for
32  * 'TAB' coordinates, is provided for CFITSIO programmers.  It has been
33  * incorporated into CFITSIO as of v3.006 with the definitions in this file,
34  * getwcstab.h, moved into fitsio.h.
35  *
36  * fits_read_wcstab() is not included in the WCSLIB object library but the
37  * source code is presented here as it may be useful for programmers using an
38  * older version of CFITSIO than 3.006, or as a programming template for
39  * non-CFITSIO programmers.
```

```
40 *
41 *
42 * fits_read_wcstab() - FITS 'TAB' table reading routine
43 * -----------------------------------------------------
44 * fits_read_wcstab() extracts arrays from a binary table required in
45 * constructing 'TAB' coordinates.
46 *
47 * Given:
48 *   fptr      fitsfile *
49 *                       Pointer to the file handle returned, for example, by
50 *                       the fits_open_file() routine in CFITSIO.
51 *
52 *   nwtb      int       Number of arrays to be read from the binary table(s).
53 *
54 * Given and returned:
55 *   wtb       wtbarr *  Address of the first element of an array of wtbarr
56 *                       typedefs.  This wtbarr typedef is defined to match the
57 *                       wtbarr struct defined in WCSLIB.  An array of such
58 *                       structs returned by the WCSLIB function wcstab() as
59 *                       discussed in the notes below.
60 *
61 * Returned:
62 *   status    int *     CFITSIO status value.
63 *
64 * Function return value:
65 *             int       CFITSIO status value.
66 *
67 * Notes:
68 *   1: In order to maintain WCSLIB and CFITSIO as independent libraries it is
69 *      not permissible for any CFITSIO library code to include WCSLIB header
70 *      files, or vice versa.  However, the CFITSIO function fits_read_wcstab()
71 *      accepts an array of wtbarr structs defined in wcs.h within WCSLIB.
72 *
73 *      The problem therefore is to define the wtbarr struct within fitsio.h
74 *      without including wcs.h, especially noting that wcs.h will often (but
75 *      not always) be included together with fitsio.h in an applications
76 *      program that uses fits_read_wcstab().
77 *
78 *      The solution adopted is for WCSLIB to define "struct wtbarr" while
79 *      fitsio.h defines "typedef wtbarr" as an untagged struct with identical
80 *      members.  This allows both wcs.h and fitsio.h to define a wtbarr data
81 *      type without conflict by virtue of the fact that structure tags and
82 *      typedef names share different name spaces in C; Appendix A, Sect. A11.1
83 *      (p227) of the K&R ANSI edition states that:
84 *
85 =        Identifiers fall into several name spaces that do not interfere with
86 =        one another; the same identifier may be used for different purposes,
87 =        even in the same scope, if the uses are in different name spaces.
88 =        These classes are: objects, functions, typedef names, and enum
89 =        constants; labels; tags of structures, unions, and enumerations; and
90 =        members of each structure or union individually.
91 *
92 *      Therefore, declarations within WCSLIB look like
93 *
94 =        struct wtbarr *w;
95 *
96 *      while within CFITSIO they are simply
97 *
98 =        wtbarr *w;
99 *
100 *      As suggested by the commonality of the names, these are really the same
101 *      aggregate data type.  However, in passing a (struct wtbarr *) to
102 *      fits_read_wcstab() a cast to (wtbarr *) is formally required.
103 *
104 *      When using WCSLIB and CFITSIO together in C++ the situation is
105 *      complicated by the fact that typedefs and structs share the same
106 *      namespace; C++ Annotated Reference Manual, Sect. 7.1.3 (p105).  In that
107 *      case the wtbarr struct in wcs.h is renamed by preprocessor macro
108 *      substitution to wtbarr_s to distinguish it from the typedef defined in
109 *      fitsio.h.  However, the scope of this macro substitution is limited to
110 *      wcs.h itself and CFITSIO programmer code, whether in C++ or C, should
111 *      always use the wtbarr typedef.
112 *
113 *
114 * wtbarr typedef
115 * --------------
116 * The wtbarr typedef is defined as a struct containing the following members:
117 *
118 *   int i
119 *      Image axis number.
120 *
121 *   int m
122 *      Array axis number for index vectors.
123 *
124 *   int kind
125 *      Character identifying the array type:
126 *         - c: coordinate array,
```

```
127  *         - i: index vector.
128  *
129  *    char extnam[72]
130  *      EXTNAME identifying the binary table extension.
131  *
132  *    int extver
133  *      EXTVER identifying the binary table extension.
134  *
135  *    int extlev
136  *      EXTLEV identifying the binary table extension.
137  *
138  *    char ttype[72]
139  *      TTYPEn identifying the column of the binary table that contains the
140  *      array.
141  *
142  *    long row
143  *      Table row number.
144  *
145  *    int ndim
146  *      Expected dimensionality of the array.
147  *
148  *    int *dimlen
149  *      Address of the first element of an array of int of length ndim into
150  *      which the array axis lengths are to be written.
151  *
152  *    double **arrayp
153  *      Pointer to an array of double which is to be allocated by the user
154  *      and into which the array is to be written.
155  *
156  *===========================================================================*/
157
158  #ifndef WCSLIB_GETWCSTAB
159  #define WCSLIB_GETWCSTAB
160
161  #ifdef __cplusplus
162  extern "C" {
163  #endif
164
165  #include <fitsio.h>
166
167  typedef struct {
168    int  i;                      // Image axis number.
169    int  m;                      // Array axis number for index vectors.
170    int  kind;                      // Array type, 'c' (coord) or 'i' (index).
171    char extnam[72];              // EXTNAME of binary table extension.
172    int  extver;                    // EXTVER  of binary table extension.
173    int  extlev;                    // EXTLEV  of binary table extension.
174    char ttype[72];               // TTYPEn of column containing the array.
175    long row;                  // Table row number.
176    int  ndim;                      // Expected array dimensionality.
177    int  *dimlen;                   // Where to write the array axis lengths.
178    double **arrayp;                // Where to write the address of the array
179                                    // allocated to store the array.
180  } wtbarr;
181
182
183  int fits_read_wcstab(fitsfile *fptr, int nwtb, wtbarr *wtb, int *status);
184
185
186  #ifdef __cplusplus
187  }
188  #endif
189
190  #endif // WCSLIB_GETWCSTAB
```

19.9 lin.h File Reference

Data Structures

- struct linprm

  Linear transformation parameters.

Macros

- #define LINLEN (sizeof(struct linprm)/sizeof(int))

  Size of the linprm struct in int units.

- #define linini_errmsg lin_errmsg

  Deprecated.
- #define lincpy_errmsg lin_errmsg

  Deprecated.
- #define linfree_errmsg lin_errmsg

  Deprecated.
- #define linprt_errmsg lin_errmsg

  Deprecated.
- #define linset_errmsg lin_errmsg

  Deprecated.
- #define linp2x_errmsg lin_errmsg

  Deprecated.
- #define linx2p_errmsg lin_errmsg

  Deprecated.

Enumerations

- enum lin_errmsg_enum {
  LINERR_SUCCESS = 0 , LINERR_NULL_POINTER = 1 , LINERR_MEMORY = 2 , LINERR_SINGULAR_MTX
  = 3 ,
  LINERR_DISTORT_INIT = 4 , LINERR_DISTORT = 5 , LINERR_DEDISTORT = 6 }

Functions

- int linini (int alloc, int naxis, struct linprm ∗lin)

  Default constructor for the linprm struct.
- int lininit (int alloc, int naxis, struct linprm ∗lin, int ndpmax)

  Default constructor for the linprm struct.
- int lindis (int sequence, struct linprm ∗lin, struct disprm ∗dis)

  Assign a distortion to a linprm struct.
- int lindist (int sequence, struct linprm ∗lin, struct disprm ∗dis, int ndpmax)

  Assign a distortion to a linprm struct.
- int lincpy (int alloc, const struct linprm ∗linsrc, struct linprm ∗lindst)

  Copy routine for the linprm struct.
- int linfree (struct linprm ∗lin)

  Destructor for the linprm struct.
- int linsize (const struct linprm ∗lin, int sizes[2])

  Compute the size of a linprm struct.
- int linprt (const struct linprm ∗lin)

  Print routine for the linprm struct.
- int linperr (const struct linprm ∗lin, const char ∗prefix)

  Print error messages from a linprm struct.
- int linset (struct linprm ∗lin)

  Setup routine for the linprm struct.
- int linp2x (struct linprm ∗lin, int ncoord, int nelem, const double pixcrd[ ], double imgcrd[ ])

  Pixel-to-world linear transformation.
- int linx2p (struct linprm ∗lin, int ncoord, int nelem, const double imgcrd[ ], double pixcrd[ ])

  World-to-pixel linear transformation.
- int linwarp (struct linprm ∗lin, const double pixblc[ ], const double pixtrc[ ], const double pixsamp[ ], int ∗nsamp,
  double maxdis[ ], double ∗maxtot, double avgdis[ ], double ∗avgtot, double rmsdis[ ], double ∗rmstot)

  Compute measures of distortion.
- int matinv (int n, const double mat[ ], double inv[ ])

  Matrix inversion.

Variables

- const char ∗ lin_errmsg [ ]

    Status return messages.

19.9.1 Detailed Description

Routines in this suite apply the linear transformation defined by the FITS World Coordinate System (WCS) standard, as described in
```
"Representations of world coordinates in FITS",
Greisen, E.W., & Calabretta, M.R. 2002, A&A, 395, 1061 (WCS Paper I)
```

These routines are based on the linprm struct which contains all information needed for the computations. The struct contains some members that must be set by the user, and others that are maintained by these routines, somewhat like a C++ class but with no encapsulation.

Six routines, linini(), lininit(), lindis(), lindist() lincpy(), and linfree() are provided to manage the linprm struct, linsize() computes its total size including allocated memory, and linprt() prints its contents.

linperr() prints the error message(s) (if any) stored in a linprm struct, and the disprm structs that it may contain.

A setup routine, linset(), computes intermediate values in the linprm struct from parameters in it that were supplied by the user. The struct always needs to be set up by linset() but need not be called explicitly - refer to the explanation of linprm::flag.

linp2x() and linx2p() implement the WCS linear transformations.

An auxiliary routine, linwarp(), computes various measures of the distortion over a specified range of pixel coordinates.

An auxiliary matrix inversion routine, matinv(), is included. It uses LU-triangular factorization with scaled partial pivoting.

19.9.2 Macro Definition Documentation

19.9.2.1 LINLEN `#define LINLEN (sizeof(struct linprm)/sizeof(int))`

Size of the linprm struct in *int* units, used by the Fortran wrappers.

19.9.2.2 linini_errmsg `#define linini_errmsg lin_errmsg`

Deprecated Added for backwards compatibility, use lin_errmsg directly now instead.

19.9.2.3 lincpy_errmsg `#define lincpy_errmsg` `lin_errmsg`

Deprecated Added for backwards compatibility, use lin_errmsg directly now instead.

19.9.2.4 linfree_errmsg `#define linfree_errmsg` `lin_errmsg`

Deprecated Added for backwards compatibility, use lin_errmsg directly now instead.

19.9.2.5 linprt_errmsg `#define linprt_errmsg` `lin_errmsg`

Deprecated Added for backwards compatibility, use lin_errmsg directly now instead.

19.9.2.6 linset_errmsg `#define linset_errmsg` `lin_errmsg`

Deprecated Added for backwards compatibility, use lin_errmsg directly now instead.

19.9.2.7 linp2x_errmsg `#define linp2x_errmsg` `lin_errmsg`

Deprecated Added for backwards compatibility, use lin_errmsg directly now instead.

19.9.2.8 linx2p_errmsg `#define linx2p_errmsg` `lin_errmsg`

Deprecated Added for backwards compatibility, use lin_errmsg directly now instead.

19.9.3 Enumeration Type Documentation

19.9.3.1 lin_errmsg_enum `enum` `lin_errmsg_enum`

Enumerator

LINERR_SUCCESS	
LINERR_NULL_POINTER	
LINERR_MEMORY	
LINERR_SINGULAR_MTX	
LINERR_DISTORT_INIT	
LINERR_DISTORT	
LINERR_DEDISTORT	

19.9.4   Function Documentation

19.9.4.1   linini()   `int linini (`
```
        int alloc,
        int naxis,
        struct linprm * lin )
```

linini() is a thin wrapper on **lininit**(). It invokes it with ndpmax set to -1 which causes it to use the value of the global variable NDPMAX. It is thereby potentially thread-unsafe if NDPMAX is altered dynamically via disndp(). Use **lininit**() for a thread-safe alternative in this case.

19.9.4.2   lininit()   `int lininit (`
```
        int alloc,
        int naxis,
        struct linprm * lin,
        int ndpmax )
```

lininit() allocates memory for arrays in a linprm struct and sets all members of the struct to default values.

PLEASE NOTE: every linprm struct must be initialized by **lininit**(), possibly repeatedly. On the first invokation, and only the first invokation, linprm::flag must be set to -1 to initialize memory management, regardless of whether **lininit**() will actually be used to allocate memory.

Parameters

`in`	*alloc*	If true, allocate memory unconditionally for arrays in the linprm struct. If false, it is assumed that pointers to these arrays have been set by the user except if they are null pointers in which case memory will be allocated for them regardless. (In other words, setting alloc true saves having to initalize these pointers to zero.)
`in`	*naxis*	The number of world coordinate axes, used to determine array sizes.
`in,out`	*lin*	Linear transformation parameters. Note that, in order to initialize memory management linprm::flag should be set to -1 when lin is initialized for the first time (memory leaks may result if it had already been initialized).
`in`	*ndpmax*	The number of DPja or DQia keywords to allocate space for. If set to -1, the value of the global variable NDPMAX will be used. This is potentially thread-unsafe if disndp() is being used dynamically to alter its value.

Returns

Status return value:

- 0: Success.
- 1: Null linprm pointer passed.
- 2: Memory allocation failed.

For returns > 1, a detailed error message is set in linprm::err if enabled, see wcserr_enable().

19.9.4.3 lindis() `int lindis (`
`int sequence,`
`struct linprm * lin,`
`struct disprm * dis )`

lindis() is a thin wrapper on **lindist**(). It invokes it with ndpmax set to -1 which causes the value of the global variable NDPMAX to be used (by disinit()). It is thereby potentially thread-unsafe if NDPMAX is altered dynamically via disndp(). Use **lindist**() for a thread-safe alternative in this case.

19.9.4.4 lindist() `int lindist (`
`int sequence,`
`struct linprm * lin,`
`struct disprm * dis,`
`int ndpmax )`

lindist() may be used to assign the address of a disprm struct to linprm::dispre or linprm::disseq. The linprm struct must already have been initialized by lininit().

The disprm struct must have been allocated from the heap (e.g. using malloc(), calloc(), etc.). **lindist**() will immediately initialize it via a call to disini() using the value of linprm::naxis. Subsequently, it will be reinitialized by calls to lininit(), and freed by linfree(), neither of which would happen if the disprm struct was assigned directly.

If the disprm struct had previously been assigned via **lindist**(), it will be freed before reassignment. It is also permissable for a null disprm pointer to be assigned to disable the distortion correction.

Parameters

in	sequence	Is it a prior or sequent distortion? • 1: Prior, the assignment is to linprm::dispre. • 2: Sequent, the assignment is to linprm::disseq. Anything else is an error.
in,out	lin	Linear transformation parameters.
in,out	dis	Distortion function parameters.
in	ndpmax	The number of DPja or DQia keywords to allocate space for. If set to -1, the value of the global variable NDPMAX will be used. This is potentially thread-unsafe if disndp() is being used dynamically to alter its value.

Returns

Status return value:

- 0: Success.

- 1: Null linprm pointer passed.

- 4: Invalid sequence.

19.9.4.5   lincpy()   `int lincpy (`
        `int alloc,`
        `const struct linprm * linsrc,`
        `struct linprm * lindst )`

lincpy() does a deep copy of one linprm struct to another, using lininit() to allocate memory for its arrays if required. Only the "information to be provided" part of the struct is copied; a call to linset() is required to initialize the remainder.

Parameters

in	*alloc*	If true, allocate memory for the crpix, pc, and cdelt arrays in the destination. Otherwise, it is assumed that pointers to these arrays have been set by the user except if they are null pointers in which case memory will be allocated for them regardless.
in	*linsrc*	Struct to copy from.
in,out	*lindst*	Struct to copy to. linprm::flag should be set to -1 if lindst was not previously initialized (memory leaks may result if it was previously initialized).

Returns

Status return value:

- 0: Success.

- 1: Null linprm pointer passed.

- 2: Memory allocation failed.

For returns > 1, a detailed error message is set in linprm::err if enabled, see wcserr_enable().

19.9.4.6   linfree()   `int linfree (`
        `struct linprm * lin )`

linfree() frees memory allocated for the linprm arrays by lininit() and/or linset(). lininit() keeps a record of the memory it allocates and **linfree**() will only attempt to free this.

PLEASE NOTE: linfree() must not be invoked on a linprm struct that was not initialized by lininit().

Parameters

in	*lin*	Linear transformation parameters.

Returns

Status return value:

- 0: Success.

- 1: Null linprm pointer passed.

19.9.4.7 linsize() `int linsize (`
            `const struct linprm * lin,`
            `int sizes[2] )`

linsize() computes the full size of a linprm struct, including allocated memory.

Parameters

in	*lin*	Linear transformation parameters.
		If NULL, the base size of the struct and the allocated size are both set to zero.
out	*sizes*	The first element is the base size of the struct as returned by sizeof(struct linprm).
		The second element is the total size of memory allocated in the struct, in bytes, assuming that the allocation was done by linini(). This figure includes memory allocated for members of constituent structs, such as linprm::dispre.
		It is not an error for the struct not to have been set up via linset(), which normally results in additional memory allocation.

Returns

Status return value:

- 0: Success.

19.9.4.8 linprt() `int linprt (`
            `const struct linprm * lin )`

linprt() prints the contents of a linprm struct using wcsprintf(). Mainly intended for diagnostic purposes.

Parameters

in	*lin*	Linear transformation parameters.

Returns

Status return value:

- 0: Success.

- 1: Null linprm pointer passed.

19.9.4.9 linperr() `int linperr (`
            `const struct `linprm` * `*lin,*
            `const char * `*prefix* `)`

linperr() prints the error message(s) (if any) stored in a linprm struct, and the disprm structs that it may contain. If there are no errors then nothing is printed. It uses wcserr_prt(), q.v.

Parameters

in	*lin*	Coordinate transformation parameters.
in	*prefix*	If non-NULL, each output line will be prefixed with this string.

Returns

> Status return value:
>
> - 0: Success.
>
> - 1: Null linprm pointer passed.

19.9.4.10 linset() `int linset (`
            `struct `linprm` * `*lin* `)`

linset(), if necessary, allocates memory for the linprm::piximg and linprm::imgpix arrays and sets up the linprm struct according to information supplied within it - refer to the explanation of linprm::flag.

Note that this routine need not be called directly; it will be invoked by linp2x() and linx2p() if the linprm::flag is anything other than a predefined magic value.

Parameters

in,out	*lin*	Linear transformation parameters.

Returns

> Status return value:
>
> - 0: Success.
> - 1: Null linprm pointer passed.
> - 2: Memory allocation failed.
> - 3: **PC**i_ja matrix is singular.
>
> For returns > 1, a detailed error message is set in linprm::err if enabled, see wcserr_enable().

19.9.4.11 linp2x() `int linp2x (`
            `struct `linprm` * `*lin,*
            `int `*ncoord,*
            `int `*nelem,*

```
            const double pixcrd[],
            double imgcrd[] )
```

linp2x() transforms pixel coordinates to intermediate world coordinates.

Parameters

in,out	*lin*	Linear transformation parameters.
in	*ncoord,nelem*	The number of coordinates, each of vector length nelem but containing lin.naxis coordinate elements.
in	*pixcrd*	Array of pixel coordinates.
out	*imgcrd*	Array of intermediate world coordinates.

Returns

Status return value:

- 0: Success.
- 1: Null linprm pointer passed.
- 2: Memory allocation failed.
- 3: **PC**i_ja matrix is singular.

For returns > 1, a detailed error message is set in linprm::err if enabled, see wcserr_enable().

19.9.4.12 linx2p() `int linx2p (`
`          struct linprm * lin,`
`          int ncoord,`
`          int nelem,`
`          const double imgcrd[],`
`          double pixcrd[] )`

linx2p() transforms intermediate world coordinates to pixel coordinates.

Parameters

in,out	*lin*	Linear transformation parameters.
in	*ncoord,nelem*	The number of coordinates, each of vector length nelem but containing lin.naxis coordinate elements.
in	*imgcrd*	Array of intermediate world coordinates.
out	*pixcrd*	Array of pixel coordinates. Status return value: • 0: Success. • 1: Null linprm pointer passed. • 2: Memory allocation failed. • 3: **PC**i_ja matrix is singular. For returns > 1, a detailed error message is set in linprm::err if enabled, see wcserr_enable().

19.9.4.13 linwarp() `int linwarp (`
`          struct linprm * lin,`

```
              const double pixblc[],
              const double pixtrc[],
              const double pixsamp[],
              int * nsamp,
              double maxdis[],
              double * maxtot,
              double avgdis[],
              double * avgtot,
              double rmsdis[],
              double * rmstot )
```

linwarp() computes various measures of the distortion over a specified range of pixel coordinates.

All distortion measures are specified as an offset in pixel coordinates, as given directly by prior distortions. The offset in intermediate pixel coordinates given by sequent distortions is translated back to pixel coordinates by applying the inverse of the linear transformation matrix (**PC**i_ja or **CD**i_ja). The difference may be significant if the matrix introduced a scaling.

If all distortions are prior, then **linwarp**() uses diswarp(), q.v.

Parameters

in,out	*lin*	Linear transformation parameters plus distortions.
in	*pixblc*	Start of the range of pixel coordinates (i.e. "bottom left-hand corner" in the conventional FITS image display orientation). May be specified as a NULL pointer which is interpreted as (1,1,...).
in	*pixtrc*	End of the range of pixel coordinates (i.e. "top right-hand corner" in the conventional FITS image display orientation).
in	*pixsamp*	If positive or zero, the increment on the particular axis, starting at pixblc[]. Zero is interpreted as a unit increment. pixsamp may also be specified as a NULL pointer which is interpreted as all zeroes, i.e. unit increments on all axes. If negative, the grid size on the particular axis (the absolute value being rounded to the nearest integer). For example, if pixsamp is (-128.0,-128.0,...) then each axis will be sampled at 128 points between pixblc[] and pixtrc[] inclusive. Use caution when using this option on non-square images.
out	*nsamp*	The number of pixel coordinates sampled. Can be specified as a NULL pointer if not required.
out	*maxdis*	For each individual distortion function, the maximum absolute value of the distortion. Can be specified as a NULL pointer if not required.
out	*maxtot*	For the combination of all distortion functions, the maximum absolute value of the distortion. Can be specified as a NULL pointer if not required.
out	*avgdis*	For each individual distortion function, the mean value of the distortion. Can be specified as a NULL pointer if not required.
out	*avgtot*	For the combination of all distortion functions, the mean value of the distortion. Can be specified as a NULL pointer if not required.
out	*rmsdis*	For each individual distortion function, the root mean square deviation of the distortion. Can be specified as a NULL pointer if not required.
out	*rmstot*	For the combination of all distortion functions, the root mean square deviation of the distortion. Can be specified as a NULL pointer if not required.

Returns

Status return value:

- 0: Success.
- 1: Null linprm pointer passed.
- 2: Memory allocation failed.
- 3: Invalid parameter.
- 4: Distort error.

19.9.4.14 matinv() matinv (
        int *n,*
        const double *mat[ ],*
        double *inv[ ]*)

matinv() performs matrix inversion using LU-triangular factorization with scaled partial pivoting.

Parameters

in	*n*	Order of the matrix ($n \times n$).
in	*mat*	Matrix to be inverted, stored as mat[$in + j$] where i and j are the row and column indices respectively.
out	*inv*	Inverse of mat with the same storage convention.

Returns

Status return value:

- 0: Success.
- 2: Memory allocation failed.
- 3: Singular matrix.

19.9.5 Variable Documentation

19.9.5.1 lin_errmsg const char ∗ lin_errmsg[ ] [extern]

Error messages to match the status value returned from each function.

19.10 lin.h

Go to the documentation of this file.
```
1 /*============================================================================
2   WCSLIB 7.11 - an implementation of the FITS WCS standard.
3   Copyright (C) 1995-2022, Mark Calabretta
4
5   This file is part of WCSLIB.
6
```

```
 7   WCSLIB is free software: you can redistribute it and/or modify it under the
 8   terms of the GNU Lesser General Public License as published by the Free
 9   Software Foundation, either version 3 of the License, or (at your option)
10   any later version.
11
12   WCSLIB is distributed in the hope that it will be useful, but WITHOUT ANY
13   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
14   FOR A PARTICULAR PURPOSE.  See the GNU Lesser General Public License for
15   more details.
16
17   You should have received a copy of the GNU Lesser General Public License
18   along with WCSLIB.  If not, see http://www.gnu.org/licenses.
19
20   Author: Mark Calabretta, Australia Telescope National Facility, CSIRO.
21   http://www.atnf.csiro.au/people/Mark.Calabretta
22   $Id: lin.h,v 7.11 2022/04/26 06:13:52 mcalabre Exp $
23 *=============================================================================
24 *
25 * WCSLIB 7.11 - C routines that implement the FITS World Coordinate System
26 * (WCS) standard.  Refer to the README file provided with WCSLIB for an
27 * overview of the library.
28 *
29 *
30 * Summary of the lin routines
31 * ---------------------------
32 * Routines in this suite apply the linear transformation defined by the FITS
33 * World Coordinate System (WCS) standard, as described in
34 *
35 =    "Representations of world coordinates in FITS",
36 =    Greisen, E.W., & Calabretta, M.R. 2002, A&A, 395, 1061 (WCS Paper I)
37 *
38 * These routines are based on the linprm struct which contains all information
39 * needed for the computations.  The struct contains some members that must be
40 * set by the user, and others that are maintained by these routines, somewhat
41 * like a C++ class but with no encapsulation.
42 *
43 * Six routines, linini(), lininit(), lindis(), lindist() lincpy(), and
44 * linfree() are provided to manage the linprm struct, linsize() computes its
45 * total size including allocated memory, and linprt() prints its contents.
46 *
47 * linperr() prints the error message(s) (if any) stored in a linprm struct,
48 * and the disprm structs that it may contain.
49 *
50 * A setup routine, linset(), computes intermediate values in the linprm struct
51 * from parameters in it that were supplied by the user.  The struct always
52 * needs to be set up by linset() but need not be called explicitly - refer to
53 * the explanation of linprm::flag.
54 *
55 * linp2x() and linx2p() implement the WCS linear transformations.
56 *
57 * An auxiliary routine, linwarp(), computes various measures of the distortion
58 * over a specified range of pixel coordinates.
59 *
60 * An auxiliary matrix inversion routine, matinv(), is included.  It uses
61 * LU-triangular factorization with scaled partial pivoting.
62 *
63 *
64 * linini() - Default constructor for the linprm struct
65 * ----------------------------------------------------
66 * linini() is a thin wrapper on lininit().  It invokes it with ndpmax set
67 * to -1 which causes it to use the value of the global variable NDPMAX.  It
68 * is thereby potentially thread-unsafe if NDPMAX is altered dynamically via
69 * disndp().  Use lininit() for a thread-safe alternative in this case.
70 *
71 *
72 * lininit() - Default constructor for the linprm struct
73 * -----------------------------------------------------
74 * lininit() allocates memory for arrays in a linprm struct and sets all
75 * members of the struct to default values.
76 *
77 * PLEASE NOTE: every linprm struct must be initialized by lininit(), possibly
78 * repeatedly.  On the first invokation, and only the first invokation,
79 * linprm::flag must be set to -1 to initialize memory management, regardless
80 * of whether lininit() will actually be used to allocate memory.
81 *
82 * Given:
83 *    alloc     int       If true, allocate memory unconditionally for arrays in
84 *                        the linprm struct.
85 *
86 *                        If false, it is assumed that pointers to these arrays
87 *                        have been set by the user except if they are null
88 *                        pointers in which case memory will be allocated for
89 *                        them regardless.  (In other words, setting alloc true
90 *                        saves having to initalize these pointers to zero.)
91 *
92 *    naxis     int       The number of world coordinate axes, used to determine
93 *                        array sizes.
```

```
 94 *
 95 * Given and returned:
 96 *   lin       struct linprm*
 97 *                     Linear transformation parameters.  Note that, in order
 98 *                     to initialize memory management linprm::flag should be
 99 *                     set to -1 when lin is initialized for the first time
100 *                     (memory leaks may result if it had already been
101 *                     initialized).
102 *
103 * Given:
104 *   ndpmax    int       The number of DPja or DQia keywords to allocate space
105 *                     for.  If set to -1, the value of the global variable
106 *                     NDPMAX will be used.  This is potentially
107 *                     thread-unsafe if disndp() is being used dynamically to
108 *                     alter its value.
109 *
110 * Function return value:
111 *             int       Status return value:
112 *                         0: Success.
113 *                         1: Null linprm pointer passed.
114 *                         2: Memory allocation failed.
115 *
116 *                     For returns > 1, a detailed error message is set in
117 *                     linprm::err if enabled, see wcserr_enable().
118 *
119 *
120 * lindis() - Assign a distortion to a linprm struct
121 * -------------------------------------------------
122 * lindis() is a thin wrapper on lindist().  It invokes it with ndpmax set
123 * to -1 which causes the value of the global variable NDPMAX to be used (by
124 * disinit()).  It is thereby potentially thread-unsafe if NDPMAX is altered
125 * dynamically via disndp().  Use lindist() for a thread-safe alternative in
126 * this case.
127 *
128 *
129 * lindist() - Assign a distortion to a linprm struct
130 * -------------------------------------------------
131 * lindist() may be used to assign the address of a disprm struct to
132 * linprm::dispre or linprm::disseq.  The linprm struct must already have been
133 * initialized by lininit().
134 *
135 * The disprm struct must have been allocated from the heap (e.g. using
136 * malloc(), calloc(), etc.).  lindist() will immediately initialize it via a
137 * call to disini() using the value of linprm::naxis.  Subsequently, it will be
138 * reinitialized by calls to lininit(), and freed by linfree(), neither of
139 * which would happen if the disprm struct was assigned directly.
140 *
141 * If the disprm struct had previously been assigned via lindist(), it will be
142 * freed before reassignment.  It is also permissable for a null disprm pointer
143 * to be assigned to disable the distortion correction.
144 *
145 * Given:
146 *   sequence  int       Is it a prior or sequent distortion?
147 *                         1: Prior,   the assignment is to linprm::dispre.
148 *                         2: Sequent, the assignment is to linprm::disseq.
149 *
150 *                     Anything else is an error.
151 *
152 * Given and returned:
153 *   lin       struct linprm*
154 *                     Linear transformation parameters.
155 *
156 *   dis       struct disprm*
157 *                     Distortion function parameters.
158 *
159 * Given:
160 *   ndpmax    int       The number of DPja or DQia keywords to allocate space
161 *                     for.  If set to -1, the value of the global variable
162 *                     NDPMAX will be used.  This is potentially
163 *                     thread-unsafe if disndp() is being used dynamically to
164 *                     alter its value.
165 *
166 * Function return value:
167 *             int       Status return value:
168 *                         0: Success.
169 *                         1: Null linprm pointer passed.
170 *                         4: Invalid sequence.
171 *
172 *
173 * lincpy() - Copy routine for the linprm struct
174 * ---------------------------------------------
175 * lincpy() does a deep copy of one linprm struct to another, using lininit()
176 * to allocate memory for its arrays if required.  Only the "information to be
177 * provided" part of the struct is copied; a call to linset() is required to
178 * initialize the remainder.
179 *
180 * Given:
```

```
181 *   alloc     int       If true, allocate memory for the crpix, pc, and cdelt
182 *                       arrays in the destination.  Otherwise, it is assumed
183 *                       that pointers to these arrays have been set by the
184 *                       user except if they are null pointers in which case
185 *                       memory will be allocated for them regardless.
186 *
187 *   linsrc    const struct linprm*
188 *                       Struct to copy from.
189 *
190 * Given and returned:
191 *   lindst    struct linprm*
192 *                       Struct to copy to.  linprm::flag should be set to -1
193 *                       if lindst was not previously initialized (memory leaks
194 *                       may result if it was previously initialized).
195 *
196 * Function return value:
197 *             int       Status return value:
198 *                         0: Success.
199 *                         1: Null linprm pointer passed.
200 *                         2: Memory allocation failed.
201 *
202 *                       For returns > 1, a detailed error message is set in
203 *                       linprm::err if enabled, see wcserr_enable().
204 *
205 *
206 * linfree() - Destructor for the linprm struct
207 * --------------------------------------------
208 * linfree() frees memory allocated for the linprm arrays by lininit() and/or
209 * linset().  lininit() keeps a record of the memory it allocates and linfree()
210 * will only attempt to free this.
211 *
212 * PLEASE NOTE: linfree() must not be invoked on a linprm struct that was not
213 * initialized by lininit().
214 *
215 * Given:
216 *   lin       struct linprm*
217 *                       Linear transformation parameters.
218 *
219 * Function return value:
220 *             int       Status return value:
221 *                         0: Success.
222 *                         1: Null linprm pointer passed.
223 *
224 *
225 * linsize() - Compute the size of a linprm struct
226 * -----------------------------------------------
227 * linsize() computes the full size of a linprm struct, including allocated
228 * memory.
229 *
230 * Given:
231 *   lin       const struct linprm*
232 *                       Linear transformation parameters.
233 *
234 *                       If NULL, the base size of the struct and the allocated
235 *                       size are both set to zero.
236 *
237 * Returned:
238 *   sizes     int[2]    The first element is the base size of the struct as
239 *                       returned by sizeof(struct linprm).
240 *
241 *                       The second element is the total size of memory
242 *                       allocated in the struct, in bytes, assuming that the
243 *                       allocation was done by linini().  This figure includes
244 *                       memory allocated for members of constituent structs,
245 *                       such as linprm::dispre.
246 *
247 *                       It is not an error for the struct not to have been set
248 *                       up via linset(), which normally results in additional
249 *                       memory allocation.
250 *
251 * Function return value:
252 *             int       Status return value:
253 *                         0: Success.
254 *
255 *
256 * linprt() - Print routine for the linprm struct
257 * ----------------------------------------------
258 * linprt() prints the contents of a linprm struct using wcsprintf().  Mainly
259 * intended for diagnostic purposes.
260 *
261 * Given:
262 *   lin       const struct linprm*
263 *                       Linear transformation parameters.
264 *
265 * Function return value:
266 *             int       Status return value:
267 *                         0: Success.
```

```
268 *                          1: Null linprm pointer passed.
269 *
270 *
271 * linperr() - Print error messages from a linprm struct
272 * -----------------------------------------------------
273 * linperr() prints the error message(s) (if any) stored in a linprm struct,
274 * and the disprm structs that it may contain.  If there are no errors then
275 * nothing is printed.  It uses wcserr_prt(), q.v.
276 *
277 * Given:
278 *   lin       const struct linprm*
279 *                      Coordinate transformation parameters.
280 *
281 *   prefix    const char *
282 *                      If non-NULL, each output line will be prefixed with
283 *                      this string.
284 *
285 * Function return value:
286 *             int       Status return value:
287 *                          0: Success.
288 *                          1: Null linprm pointer passed.
289 *
290 *
291 * linset() - Setup routine for the linprm struct
292 * ----------------------------------------------
293 * linset(), if necessary, allocates memory for the linprm::piximg and
294 * linprm::imgpix arrays and sets up the linprm struct according to information
295 * supplied within it - refer to the explanation of linprm::flag.
296 *
297 * Note that this routine need not be called directly; it will be invoked by
298 * linp2x() and linx2p() if the linprm::flag is anything other than a
299 * predefined magic value.
300 *
301 * Given and returned:
302 *   lin       struct linprm*
303 *                      Linear transformation parameters.
304 *
305 * Function return value:
306 *             int       Status return value:
307 *                          0: Success.
308 *                          1: Null linprm pointer passed.
309 *                          2: Memory allocation failed.
310 *                          3: PCi_ja matrix is singular.
311 *
312 *                      For returns > 1, a detailed error message is set in
313 *                      linprm::err if enabled, see wcserr_enable().
314 *
315 *
316 * linp2x() - Pixel-to-world linear transformation
317 * -----------------------------------------------
318 * linp2x() transforms pixel coordinates to intermediate world coordinates.
319 *
320 * Given and returned:
321 *   lin       struct linprm*
322 *                      Linear transformation parameters.
323 *
324 * Given:
325 *   ncoord,
326 *   nelem     int       The number of coordinates, each of vector length nelem
327 *                      but containing lin.naxis coordinate elements.
328 *
329 *   pixcrd    const double[ncoord][nelem]
330 *                      Array of pixel coordinates.
331 *
332 * Returned:
333 *   imgcrd    double[ncoord][nelem]
334 *                      Array of intermediate world coordinates.
335 *
336 * Function return value:
337 *             int       Status return value:
338 *                          0: Success.
339 *                          1: Null linprm pointer passed.
340 *                          2: Memory allocation failed.
341 *                          3: PCi_ja matrix is singular.
342 *
343 *                      For returns > 1, a detailed error message is set in
344 *                      linprm::err if enabled, see wcserr_enable().
345 *
346 *
347 * linx2p() - World-to-pixel linear transformation
348 * -----------------------------------------------
349 * linx2p() transforms intermediate world coordinates to pixel coordinates.
350 *
351 * Given and returned:
352 *   lin       struct linprm*
353 *                      Linear transformation parameters.
354 *
```

```
355 * Given:
356 *   ncoord,
357 *   nelem     int      The number of coordinates, each of vector length nelem
358 *                      but containing lin.naxis coordinate elements.
359 *
360 *   imgcrd    const double[ncoord][nelem]
361 *                      Array of intermediate world coordinates.
362 *
363 * Returned:
364 *   pixcrd    double[ncoord][nelem]
365 *                      Array of pixel coordinates.
366 *
367 *             int      Status return value:
368 *                        0: Success.
369 *                        1: Null linprm pointer passed.
370 *                        2: Memory allocation failed.
371 *                        3: PCi_ja matrix is singular.
372 *
373 *                      For returns > 1, a detailed error message is set in
374 *                      linprm::err if enabled, see wcserr_enable().
375 *
376 *
377 * linwarp() - Compute measures of distortion
378 * -------------------------------------------
379 * linwarp() computes various measures of the distortion over a specified range
380 * of pixel coordinates.
381 *
382 * All distortion measures are specified as an offset in pixel coordinates,
383 * as given directly by prior distortions.  The offset in intermediate pixel
384 * coordinates given by sequent distortions is translated back to pixel
385 * coordinates by applying the inverse of the linear transformation matrix
386 * (PCi_ja or CDi_ja).  The difference may be significant if the matrix
387 * introduced a scaling.
388 *
389 * If all distortions are prior, then linwarp() uses diswarp(), q.v.
390 *
391 * Given and returned:
392 *   lin       struct linprm*
393 *                      Linear transformation parameters plus distortions.
394 *
395 * Given:
396 *   pixblc    const double[naxis]
397 *                      Start of the range of pixel coordinates (i.e. "bottom
398 *                      left-hand corner" in the conventional FITS image
399 *                      display orientation).  May be specified as a NULL
400 *                      pointer which is interpreted as (1,1,...).
401 *
402 *   pixtrc    const double[naxis]
403 *                      End of the range of pixel coordinates (i.e. "top
404 *                      right-hand corner" in the conventional FITS image
405 *                      display orientation).
406 *
407 *   pixsamp   const double[naxis]
408 *                      If positive or zero, the increment on the particular
409 *                      axis, starting at pixblc[].  Zero is interpreted as a
410 *                      unit increment.  pixsamp may also be specified as a
411 *                      NULL pointer which is interpreted as all zeroes, i.e.
412 *                      unit increments on all axes.
413 *
414 *                      If negative, the grid size on the particular axis (the
415 *                      absolute value being rounded to the nearest integer).
416 *                      For example, if pixsamp is (-128.0,-128.0,...) then
417 *                      each axis will be sampled at 128 points between
418 *                      pixblc[] and pixtrc[] inclusive.  Use caution when
419 *                      using this option on non-square images.
420 *
421 * Returned:
422 *   nsamp     int*     The number of pixel coordinates sampled.
423 *
424 *                      Can be specified as a NULL pointer if not required.
425 *
426 *   maxdis    double[naxis]
427 *                      For each individual distortion function, the
428 *                      maximum absolute value of the distortion.
429 *
430 *                      Can be specified as a NULL pointer if not required.
431 *
432 *   maxtot    double*  For the combination of all distortion functions, the
433 *                      maximum absolute value of the distortion.
434 *
435 *                      Can be specified as a NULL pointer if not required.
436 *
437 *   avgdis    double[naxis]
438 *                      For each individual distortion function, the
439 *                      mean value of the distortion.
440 *
441 *                      Can be specified as a NULL pointer if not required.
```

```
442 *
443 *    avgtot    double*    For the combination of all distortion functions, the
444 *                         mean value of the distortion.
445 *
446 *                         Can be specified as a NULL pointer if not required.
447 *
448 *    rmsdis    double[naxis]
449 *                         For each individual distortion function, the
450 *                         root mean square deviation of the distortion.
451 *
452 *                         Can be specified as a NULL pointer if not required.
453 *
454 *    rmstot    double*    For the combination of all distortion functions, the
455 *                         root mean square deviation of the distortion.
456 *
457 *                         Can be specified as a NULL pointer if not required.
458 *
459 * Function return value:
460 *              int        Status return value:
461 *                           0: Success.
462 *                           1: Null linprm pointer passed.
463 *                           2: Memory allocation failed.
464 *                           3: Invalid parameter.
465 *                           4: Distort error.
466 *
467 *
468 * linprm struct - Linear transformation parameters
469 * -------------------------------------------------
470 * The linprm struct contains all of the information required to perform a
471 * linear transformation.  It consists of certain members that must be set by
472 * the user ("given") and others that are set by the WCSLIB routines
473 * ("returned").
474 *
475 *    int flag
476 *       (Given and returned) This flag must be set to zero whenever any of the
477 *       following members of the linprm struct are set or modified:
478 *
479 *         - linprm::naxis (q.v., not normally set by the user),
480 *         - linprm::pc,
481 *         - linprm::cdelt,
482 *         - linprm::dispre.
483 *         - linprm::disseq.
484 *
485 *       This signals the initialization routine, linset(), to recompute the
486 *       returned members of the linprm struct.  linset() will reset flag to
487 *       indicate that this has been done.
488 *
489 *       PLEASE NOTE: flag should be set to -1 when lininit() is called for the
490 *       first time for a particular linprm struct in order to initialize memory
491 *       management.  It must ONLY be used on the first initialization otherwise
492 *       memory leaks may result.
493 *
494 *    int naxis
495 *       (Given or returned) Number of pixel and world coordinate elements.
496 *
497 *       If lininit() is used to initialize the linprm struct (as would normally
498 *       be the case) then it will set naxis from the value passed to it as a
499 *       function argument.  The user should not subsequently modify it.
500 *
501 *    double *crpix
502 *       (Given) Pointer to the first element of an array of double containing
503 *       the coordinate reference pixel, CRPIXja.
504 *
505 *       It is not necessary to reset the linprm struct (via linset()) when
506 *       linprm::crpix is changed.
507 *
508 *    double *pc
509 *       (Given) Pointer to the first element of the PCi_ja (pixel coordinate)
510 *       transformation matrix.  The expected order is
511 *
512 =          struct linprm lin;
513 =          lin.pc = {PC1_1, PC1_2, PC2_1, PC2_2};
514 *
515 *       This may be constructed conveniently from a 2-D array via
516 *
517 =          double m[2][2] = {{PC1_1, PC1_2},
518 =                            {PC2_1, PC2_2}};
519 *
520 *       which is equivalent to
521 *
522 =          double m[2][2];
523 =          m[0][0] = PC1_1;
524 =          m[0][1] = PC1_2;
525 =          m[1][0] = PC2_1;
526 =          m[1][1] = PC2_2;
527 *
528 *       The storage order for this 2-D array is the same as for the 1-D array,
```

```
529 *      whence
530 *
531 =        lin.pc = *m;
532 *
533 *      would be legitimate.
534 *
535 *    double *cdelt
536 *      (Given) Pointer to the first element of an array of double containing
537 *      the coordinate increments, CDELTia.
538 *
539 *    struct disprm *dispre
540 *      (Given) Pointer to a disprm struct holding parameters for prior
541 *      distortion functions, or a null (0x0) pointer if there are none.
542 *
543 *      Function lindist() may be used to assign a disprm pointer to a linprm
544 *      struct, allowing it to take control of any memory allocated for it, as
545 *      in the following example:
546 *
547 =        void add_distortion(struct linprm *lin)
548 =        {
549 =          struct disprm *dispre;
550 =
551 =          dispre = malloc(sizeof(struct disprm));
552 =          dispre->flag = -1;
553 =          lindist(1, lin, dispre, ndpmax);
554 =            :
555 =           (Set up dispre.)
556 =            :
557 =
558 =          return;
559 =        }
560 *
561 *      Here, after the distortion function parameters etc. are copied into
562 *      dispre, dispre is assigned using lindist() which takes control of the
563 *      allocated memory.  It will be freed later when linfree() is invoked on
564 *      the linprm struct.
565 *
566 *      Consider also the following erroneous code:
567 *
568 =        void bad_code(struct linprm *lin)
569 =        {
570 =          struct disprm dispre;
571 =
572 =          dispre.flag = -1;
573 =          lindist(1, lin, &dispre, ndpmax);   // WRONG.
574 =            :
575 =
576 =          return;
577 =        }
578 *
579 *      Here, dispre is declared as a struct, rather than a pointer.  When the
580 *      function returns, dispre will go out of scope and its memory will most
581 *      likely be reused, thereby trashing its contents.  Later, a segfault will
582 *      occur when linfree() tries to free dispre's stale address.
583 *
584 *    struct disprm *disseq
585 *      (Given) Pointer to a disprm struct holding parameters for sequent
586 *      distortion functions, or a null (0x0) pointer if there are none.
587 *
588 *      Refer to the comments and examples given for disprm::dispre.
589 *
590 *    double *piximg
591 *      (Returned) Pointer to the first element of the matrix containing the
592 *      product of the CDELTia diagonal matrix and the PCi_ja matrix.
593 *
594 *    double *imgpix
595 *      (Returned) Pointer to the first element of the inverse of the
596 *      linprm::piximg matrix.
597 *
598 *    int i_naxis
599 *      (Returned) The dimension of linprm::piximg and linprm::imgpix (normally
600 *      equal to naxis).
601 *
602 *    int unity
603 *      (Returned) True if the linear transformation matrix is unity.
604 *
605 *    int affine
606 *      (Returned) True if there are no distortions.
607 *
608 *    int simple
609 *      (Returned) True if unity and no distortions.
610 *
611 *    struct wcserr *err
612 *      (Returned) If enabled, when an error status is returned, this struct
613 *      contains detailed information about the error, see wcserr_enable().
614 *
615 *    double *tmpcrd
```

```
616 *       (For internal use only.)
617 *     int m_flag
618 *       (For internal use only.)
619 *     int m_naxis
620 *       (For internal use only.)
621 *     double *m_crpix
622 *       (For internal use only.)
623 *     double *m_pc
624 *       (For internal use only.)
625 *     double *m_cdelt
626 *       (For internal use only.)
627 *     struct disprm *m_dispre
628 *       (For internal use only.)
629 *     struct disprm *m_disseq
630 *       (For internal use only.)
631 *
632 *
633 * Global variable: const char *lin_errmsg[] - Status return messages
634 * -------------------------------------------------------------------
635 * Error messages to match the status value returned from each function.
636 *
637 *===========================================================================*/
638
639 #ifndef WCSLIB_LIN
640 #define WCSLIB_LIN
641
642 #ifdef __cplusplus
643 extern "C" {
644 #endif
645
646
647 extern const char *lin_errmsg[];
648
649 enum lin_errmsg_enum {
650   LINERR_SUCCESS      = 0,      // Success.
651   LINERR_NULL_POINTER = 1,   // Null linprm pointer passed.
652   LINERR_MEMORY       = 2,  // Memory allocation failed.
653   LINERR_SINGULAR_MTX = 3,   // PCi_ja matrix is singular.
654   LINERR_DISTORT_INIT = 4,   // Failed to initialise distortions.
655   LINERR_DISTORT      = 5,      // Distort error.
656   LINERR_DEDISTORT    = 6      // De-distort error.
657 };
658
659 struct linprm {
660   // Initialization flag (see the prologue above).
661   //------------------------------------------------------------------------
662   int flag;                   // Set to zero to force initialization.
663
664   // Parameters to be provided (see the prologue above).
665   //------------------------------------------------------------------------
666   int naxis;                          // The number of axes, given by NAXIS.
667   double *crpix;            // CRPIXja keywords for each pixel axis.
668   double *pc;                 // PCi_ja  linear transformation matrix.
669   double *cdelt;            // CDELTia keywords for each coord axis.
670   struct disprm *dispre;    // Prior   distortion parameters, if any.
671   struct disprm *disseq;    // Sequent distortion parameters, if any.
672
673   // Information derived from the parameters supplied.
674   //------------------------------------------------------------------------
675   double *piximg;          // Product of CDELTia and PCi_ja matrices.
676   double *imgpix;          // Inverse of the piximg matrix.
677   int    i_naxis;               // Dimension of piximg and imgpix.
678   int    unity;                    // True if the PCi_ja matrix is unity.
679   int    affine;            // True if there are no distortions.
680   int    simple;            // True if unity and no distortions.
681
682   // Error handling, if enabled.
683   //------------------------------------------------------------------------
684   struct wcserr *err;
685
686   // Private - the remainder are for internal use.
687   //------------------------------------------------------------------------
688   double *tmpcrd;
689
690   int    m_flag, m_naxis;
691   double *m_crpix, *m_pc, *m_cdelt;
692   struct disprm *m_dispre, *m_disseq;
693 };
694
695 // Size of the linprm struct in int units, used by the Fortran wrappers.
696 #define LINLEN (sizeof(struct linprm)/sizeof(int))
697
698
699 int linini(int alloc, int naxis, struct linprm *lin);
700
701 int lininit(int alloc, int naxis, struct linprm *lin, int ndpmax);
702
```

```
703 int lindis(int sequence, struct linprm *lin, struct disprm *dis);
704
705 int lindist(int sequence, struct linprm *lin, struct disprm *dis, int ndpmax);
706
707 int lincpy(int alloc, const struct linprm *linsrc, struct linprm *lindst);
708
709 int linfree(struct linprm *lin);
710
711 int linsize(const struct linprm *lin, int sizes[2]);
712
713 int linprt(const struct linprm *lin);
714
715 int linperr(const struct linprm *lin, const char *prefix);
716
717 int linset(struct linprm *lin);
718
719 int linp2x(struct linprm *lin, int ncoord, int nelem, const double pixcrd[],
720         double imgcrd[]);
721
722 int linx2p(struct linprm *lin, int ncoord, int nelem, const double imgcrd[],
723         double pixcrd[]);
724
725 int linwarp(struct linprm *lin, const double pixblc[], const double pixtrc[],
726           const double pixsamp[], int *nsamp,
727           double maxdis[], double *maxtot,
728           double avgdis[], double *avgtot,
729           double rmsdis[], double *rmstot);
730
731 int matinv(int n, const double mat[], double inv[]);
732
733
734 // Deprecated.
735 #define linini_errmsg lin_errmsg
736 #define lincpy_errmsg lin_errmsg
737 #define linfree_errmsg lin_errmsg
738 #define linprt_errmsg lin_errmsg
739 #define linset_errmsg lin_errmsg
740 #define linp2x_errmsg lin_errmsg
741 #define linx2p_errmsg lin_errmsg
742
743 #ifdef __cplusplus
744 }
745 #endif
746
747 #endif // WCSLIB_LIN
```

19.11   log.h File Reference

Enumerations

- enum log_errmsg_enum {
  LOGERR_SUCCESS = 0 , LOGERR_NULL_POINTER = 1 , LOGERR_BAD_LOG_REF_VAL = 2 ,
  LOGERR_BAD_X = 3 ,
  LOGERR_BAD_WORLD = 4 }

Functions

- int logx2s (double crval, int nx, int sx, int slogc, const double x[ ], double logc[ ], int stat[ ])

  Transform to logarithmic coordinates.
- int logs2x (double crval, int nlogc, int slogc, int sx, const double logc[ ], double x[ ], int stat[ ])

  Transform logarithmic coordinates.

Variables

- const char ∗ log_errmsg [ ]

  Status return messages.

19.11.1   Detailed Description

Routines in this suite implement the part of the FITS World Coordinate System (WCS) standard that deals with logarithmic coordinates, as described in

"Representations of world coordinates in FITS", Greisen, E.W., & Calabretta, M.R. 2002, A&A, 395, 1061 (WCS Paper I)

"Representations of spectral coordinates in FITS", Greisen, E.W., Calabretta, M.R., Valdes, F.G., & Allen, S.L. 2006, A&A, 446, 747 (WCS Paper III)

These routines define methods to be used for computing logarithmic world coordinates from intermediate world coordinates (a linear transformation of image pixel coordinates), and vice versa.

logx2s() and logs2x() implement the WCS logarithmic coordinate transformations.

Argument checking:
The input log-coordinate values are only checked for values that would result in floating point exceptions and the same is true for the log-coordinate reference value.

Accuracy:
No warranty is given for the accuracy of these routines (refer to the copyright notice); intending users must satisfy for themselves their adequacy for the intended purpose. However, closure effectively to within double precision rounding error was demonstrated by test routine tlog.c which accompanies this software.

19.11.2   Enumeration Type Documentation

19.11.2.1   log_errmsg_enum   enum log_errmsg_enum

Enumerator

LOGERR_SUCCESS	
LOGERR_NULL_POINTER	
LOGERR_BAD_LOG_REF_VAL	
LOGERR_BAD_X	
LOGERR_BAD_WORLD	

19.11.3   Function Documentation

19.11.3.1   logx2s()   int logx2s (
            double *crval,*
            int *nx,*
            int *sx,*
            int *slogc,*
            const double *x[ ],*

```
        double logc[],
        int stat[] )
```

logx2s() transforms intermediate world coordinates to logarithmic coordinates.

Parameters

in,out	*crval*	Log-coordinate reference value (**CRVAL**ia).
in	*nx*	Vector length.
in	*sx*	Vector stride.
in	*slogc*	Vector stride.
in	*x*	Intermediate world coordinates, in SI units.
out	*logc*	Logarithmic coordinates, in SI units.
out	*stat*	Status return value status for each vector element: • 0: Success.

Returns

Status return value:

- 0: Success.
- 2: Invalid log-coordinate reference value.

19.11.3.2 logs2x() `int logs2x (`
```
        double crval,
        int nlogc,
        int slogc,
        int sx,
        const double logc[],
        double x[],
        int stat[] )
```

logs2x() transforms logarithmic world coordinates to intermediate world coordinates.

Parameters

in,out	*crval*	Log-coordinate reference value (**CRVAL**ia).
in	*nlogc*	Vector length.
in	*slogc*	Vector stride.
in	*sx*	Vector stride.
in	*logc*	Logarithmic coordinates, in SI units.
out	*x*	Intermediate world coordinates, in SI units.
out	*stat*	Status return value status for each vector element: • 0: Success. • 1: Invalid value of logc.

Returns

> Status return value:
>
> - 0: Success.
> - 2: Invalid log-coordinate reference value.
> - 4: One or more of the world-coordinate values are incorrect, as indicated by the stat vector.

19.11.4 Variable Documentation

19.11.4.1 **log_errmsg** `const char * log_errmsg[]` `[extern]`

Error messages to match the status value returned from each function.

19.12 log.h

[Go to the documentation of this file.](#)
```
1  /*============================================================================
2    WCSLIB 7.11 - an implementation of the FITS WCS standard.
3    Copyright (C) 1995-2022, Mark Calabretta
4
5    This file is part of WCSLIB.
6
7    WCSLIB is free software: you can redistribute it and/or modify it under the
8    terms of the GNU Lesser General Public License as published by the Free
9    Software Foundation, either version 3 of the License, or (at your option)
10   any later version.
11
12   WCSLIB is distributed in the hope that it will be useful, but WITHOUT ANY
13   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
14   FOR A PARTICULAR PURPOSE.  See the GNU Lesser General Public License for
15   more details.
16
17   You should have received a copy of the GNU Lesser General Public License
18   along with WCSLIB.  If not, see http://www.gnu.org/licenses.
19
20   Author: Mark Calabretta, Australia Telescope National Facility, CSIRO.
21   http://www.atnf.csiro.au/people/Mark.Calabretta
22   $Id: log.h,v 7.11 2022/04/26 06:13:52 mcalabre Exp $
23  *=============================================================================
24  *
25  * WCSLIB 7.11 - C routines that implement the FITS World Coordinate System
26  * (WCS) standard.  Refer to the README file provided with WCSLIB for an
27  * overview of the library.
28  *
29  *
30  * Summary of the log routines
31  * ---------------------------
32  * Routines in this suite implement the part of the FITS World Coordinate
33  * System (WCS) standard that deals with logarithmic coordinates, as described
34  * in
35  *
36  *   "Representations of world coordinates in FITS",
37  *   Greisen, E.W., & Calabretta, M.R. 2002, A&A, 395, 1061 (WCS Paper I)
38  *
39  *   "Representations of spectral coordinates in FITS",
40  *   Greisen, E.W., Calabretta, M.R., Valdes, F.G., & Allen, S.L.
41  *   2006, A&A, 446, 747 (WCS Paper III)
42  *
43  * These routines define methods to be used for computing logarithmic world
44  * coordinates from intermediate world coordinates (a linear transformation of
45  * image pixel coordinates), and vice versa.
46  *
47  * logx2s() and logs2x() implement the WCS logarithmic coordinate
48  * transformations.
49  *
50  * Argument checking:
51  * ------------------
52  * The input log-coordinate values are only checked for values that would
53  * result in floating point exceptions and the same is true for the
```

```
54  * log-coordinate reference value.
55  *
56  * Accuracy:
57  * ---------
58  * No warranty is given for the accuracy of these routines (refer to the
59  * copyright notice); intending users must satisfy for themselves their
60  * adequacy for the intended purpose.  However, closure effectively to within
61  * double precision rounding error was demonstrated by test routine tlog.c
62  * which accompanies this software.
63  *
64  *
65  * logx2s() - Transform to logarithmic coordinates
66  * -----------------------------------------------
67  * logx2s() transforms intermediate world coordinates to logarithmic
68  * coordinates.
69  *
70  * Given and returned:
71  *   crval     double    Log-coordinate reference value (CRVALia).
72  *
73  * Given:
74  *   nx        int       Vector length.
75  *
76  *   sx        int       Vector stride.
77  *
78  *   slogc     int       Vector stride.
79  *
80  *   x         const double[]
81  *                       Intermediate world coordinates, in SI units.
82  *
83  * Returned:
84  *   logc      double[]  Logarithmic coordinates, in SI units.
85  *
86  *   stat      int[]     Status return value status for each vector element:
87  *                         0: Success.
88  *
89  * Function return value:
90  *             int       Status return value:
91  *                         0: Success.
92  *                         2: Invalid log-coordinate reference value.
93  *
94  *
95  * logs2x() - Transform logarithmic coordinates
96  * --------------------------------------------
97  * logs2x() transforms logarithmic world coordinates to intermediate world
98  * coordinates.
99  *
100 * Given and returned:
101 *   crval     double    Log-coordinate reference value (CRVALia).
102 *
103 * Given:
104 *   nlogc     int       Vector length.
105 *
106 *   slogc     int       Vector stride.
107 *
108 *   sx        int       Vector stride.
109 *
110 *   logc      const double[]
111 *                       Logarithmic coordinates, in SI units.
112 *
113 * Returned:
114 *   x         double[]  Intermediate world coordinates, in SI units.
115 *
116 *   stat      int[]     Status return value status for each vector element:
117 *                         0: Success.
118 *                         1: Invalid value of logc.
119 *
120 * Function return value:
121 *             int       Status return value:
122 *                         0: Success.
123 *                         2: Invalid log-coordinate reference value.
124 *                         4: One or more of the world-coordinate values
125 *                            are incorrect, as indicated by the stat vector.
126 *
127 *
128 * Global variable: const char *log_errmsg[] - Status return messages
129 * -----------------------------------------------------------------
130 * Error messages to match the status value returned from each function.
131 *
132 *===========================================================================*/
133
134 #ifndef WCSLIB_LOG
135 #define WCSLIB_LOG
136
137 #ifdef __cplusplus
138 extern "C" {
139 #endif
140
```

```
141 extern const char *log_errmsg[];
142
143 enum log_errmsg_enum {
144   LOGERR_SUCCESS       = 0,     // Success.
145   LOGERR_NULL_POINTER   = 1,       // Null pointer passed.
146   LOGERR_BAD_LOG_REF_VAL = 2,     // Invalid log-coordinate reference value.
147   LOGERR_BAD_X         = 3,       // One or more of the x coordinates were
148                                   // invalid.
149   LOGERR_BAD_WORLD     = 4     // One or more of the world coordinates were
150                                   // invalid.
151 };
152
153 int logx2s(double crval, int nx, int sx, int slogc, const double x[],
154            double logc[], int stat[]);
155
156 int logs2x(double crval, int nlogc, int slogc, int sx, const double logc[],
157            double x[], int stat[]);
158
159
160 #ifdef __cplusplus
161 }
162 #endif
163
164 #endif // WCSLIB_LOG
```

19.13   prj.h File Reference

Data Structures

- struct prjprm

    Projection parameters.

Macros

- #define PVN 30

    Total number of projection parameters.
- #define PRJX2S_ARGS

    For use in declaring deprojection function prototypes.
- #define PRJS2X_ARGS

    For use in declaring projection function prototypes.
- #define PRJLEN (sizeof(struct prjprm)/sizeof(int))

    Size of the prjprm struct in int units.
- #define prjini_errmsg prj_errmsg

    Deprecated.
- #define prjprt_errmsg prj_errmsg

    Deprecated.
- #define prjset_errmsg prj_errmsg

    Deprecated.
- #define prjx2s_errmsg prj_errmsg

    Deprecated.
- #define prjs2x_errmsg prj_errmsg

    Deprecated.

Enumerations

- enum prj_errmsg_enum {
    PRJERR_SUCCESS = 0 , PRJERR_NULL_POINTER = 1 , PRJERR_BAD_PARAM = 2 , PRJERR_BAD_PIX
    = 3 ,
    PRJERR_BAD_WORLD = 4 }

Functions

- int prjini (struct prjprm ∗prj)

    Default constructor for the prjprm struct.

- int prjfree (struct prjprm ∗prj)

    Destructor for the prjprm struct.

- int prjsize (const struct prjprm ∗prj, int sizes[2])

    Compute the size of a prjprm struct.

- int prjprt (const struct prjprm ∗prj)

    Print routine for the prjprm struct.

- int prjperr (const struct prjprm ∗prj, const char ∗prefix)

    Print error messages from a prjprm struct.

- int prjbchk (double tol, int nphi, int ntheta, int spt, double phi[ ], double theta[ ], int stat[ ])

    Bounds checking on native coordinates.

- int prjset (struct prjprm ∗prj)

    Generic setup routine for the prjprm struct.

- int prjx2s (PRJX2S_ARGS)

    Generic Cartesian-to-spherical deprojection.

- int prjs2x (PRJS2X_ARGS)

    Generic spherical-to-Cartesian projection.

- int azpset (struct prjprm ∗prj)

    *Set up a prjprm struct for the **zenithal/azimuthal perspective (`AZP`)** projection.*

- int azpx2s (PRJX2S_ARGS)

    *Cartesian-to-spherical transformation for the **zenithal/azimuthal perspective (`AZP`)** projection.*

- int azps2x (PRJS2X_ARGS)

    *Spherical-to-Cartesian transformation for the **zenithal/azimuthal perspective (`AZP`)** projection.*

- int szpset (struct prjprm ∗prj)

    *Set up a prjprm struct for the **slant zenithal perspective (`SZP`)** projection.*

- int szpx2s (PRJX2S_ARGS)

    *Cartesian-to-spherical transformation for the **slant zenithal perspective (`SZP`)** projection.*

- int szps2x (PRJS2X_ARGS)

    *Spherical-to-Cartesian transformation for the **slant zenithal perspective (`SZP`)** projection.*

- int tanset (struct prjprm ∗prj)

    *Set up a prjprm struct for the **gnomonic (`TAN`)** projection.*

- int tanx2s (PRJX2S_ARGS)

    *Cartesian-to-spherical transformation for the **gnomonic (`TAN`)** projection.*

- int tans2x (PRJS2X_ARGS)

    *Spherical-to-Cartesian transformation for the **gnomonic (`TAN`)** projection.*

- int stgset (struct prjprm ∗prj)

    *Set up a prjprm struct for the **stereographic (`STG`)** projection.*

- int stgx2s (PRJX2S_ARGS)

    *Cartesian-to-spherical transformation for the **stereographic (`STG`)** projection.*

- int stgs2x (PRJS2X_ARGS)

    *Spherical-to-Cartesian transformation for the **stereographic (`STG`)** projection.*

- int sinset (struct prjprm ∗prj)

    *Set up a prjprm struct for the **orthographic/synthesis (`SIN`)** projection.*

- int sinx2s (PRJX2S_ARGS)

    *Cartesian-to-spherical transformation for the **orthographic/synthesis (`SIN`)** projection.*

- int sins2x (PRJS2X_ARGS)

    *Spherical-to-Cartesian transformation for the **orthographic/synthesis (`SIN`)** projection.*

- int arcset (struct prjprm ∗prj)

*Set up a [prjprm](#) struct for the **zenithal/azimuthal equidistant (ARC)** projection.*

- int [arcx2s](#) ([PRJX2S_ARGS](#))

  *Cartesian-to-spherical transformation for the **zenithal/azimuthal equidistant (ARC)** projection.*
- int [arcs2x](#) ([PRJS2X_ARGS](#))

  *Spherical-to-Cartesian transformation for the **zenithal/azimuthal equidistant (ARC)** projection.*
- int [zpnset](#) (struct [prjprm](#) ∗prj)

  *Set up a [prjprm](#) struct for the **zenithal/azimuthal polynomial (ZPN)** projection.*
- int [zpnx2s](#) ([PRJX2S_ARGS](#))

  *Cartesian-to-spherical transformation for the **zenithal/azimuthal polynomial (ZPN)** projection.*
- int [zpns2x](#) ([PRJS2X_ARGS](#))

  *Spherical-to-Cartesian transformation for the **zenithal/azimuthal polynomial (ZPN)** projection.*
- int [zeaset](#) (struct [prjprm](#) ∗prj)

  *Set up a [prjprm](#) struct for the **zenithal/azimuthal equal area (ZEA)** projection.*
- int [zeax2s](#) ([PRJX2S_ARGS](#))

  *Cartesian-to-spherical transformation for the **zenithal/azimuthal equal area (ZEA)** projection.*
- int [zeas2x](#) ([PRJS2X_ARGS](#))

  *Spherical-to-Cartesian transformation for the **zenithal/azimuthal equal area (ZEA)** projection.*
- int [airset](#) (struct [prjprm](#) ∗prj)

  *Set up a [prjprm](#) struct for **Airy's (AIR)** projection.*
- int [airx2s](#) ([PRJX2S_ARGS](#))

  *Cartesian-to-spherical transformation for **Airy's (AIR)** projection.*
- int [airs2x](#) ([PRJS2X_ARGS](#))

  *Spherical-to-Cartesian transformation for **Airy's (AIR)** projection.*
- int [cypset](#) (struct [prjprm](#) ∗prj)

  *Set up a [prjprm](#) struct for the **cylindrical perspective (CYP)** projection.*
- int [cypx2s](#) ([PRJX2S_ARGS](#))

  *Cartesian-to-spherical transformation for the **cylindrical perspective (CYP)** projection.*
- int [cyps2x](#) ([PRJS2X_ARGS](#))

  *Spherical-to-Cartesian transformation for the **cylindrical perspective (CYP)** projection.*
- int [ceaset](#) (struct [prjprm](#) ∗prj)

  *Set up a [prjprm](#) struct for the **cylindrical equal area (CEA)** projection.*
- int [ceax2s](#) ([PRJX2S_ARGS](#))

  *Cartesian-to-spherical transformation for the **cylindrical equal area (CEA)** projection.*
- int [ceas2x](#) ([PRJS2X_ARGS](#))

  *Spherical-to-Cartesian transformation for the **cylindrical equal area (CEA)** projection.*
- int [carset](#) (struct [prjprm](#) ∗prj)

  *Set up a [prjprm](#) struct for the **plate carrée (CAR)** projection.*
- int [carx2s](#) ([PRJX2S_ARGS](#))

  *Cartesian-to-spherical transformation for the **plate carrée (CAR)** projection.*
- int [cars2x](#) ([PRJS2X_ARGS](#))

  *Spherical-to-Cartesian transformation for the **plate carrée (CAR)** projection.*
- int [merset](#) (struct [prjprm](#) ∗prj)

  *Set up a [prjprm](#) struct for **Mercator's (MER)** projection.*
- int [merx2s](#) ([PRJX2S_ARGS](#))

  *Cartesian-to-spherical transformation for **Mercator's (MER)** projection.*
- int [mers2x](#) ([PRJS2X_ARGS](#))

  *Spherical-to-Cartesian transformation for **Mercator's (MER)** projection.*
- int [sflset](#) (struct [prjprm](#) ∗prj)

  *Set up a [prjprm](#) struct for the **Sanson-Flamsteed (SFL)** projection.*
- int [sflx2s](#) ([PRJX2S_ARGS](#))

  *Cartesian-to-spherical transformation for the **Sanson-Flamsteed (SFL)** projection.*

- int sfls2x (PRJS2X_ARGS)

  *Spherical-to-Cartesian transformation for the **Sanson-Flamsteed (`SFL`)** projection.*
- int parset (struct prjprm ∗prj)

  *Set up a prjprm struct for the **parabolic (`PAR`)** projection.*
- int parx2s (PRJX2S_ARGS)

  *Cartesian-to-spherical transformation for the **parabolic (`PAR`)** projection.*
- int pars2x (PRJS2X_ARGS)

  *Spherical-to-Cartesian transformation for the **parabolic (`PAR`)** projection.*
- int molset (struct prjprm ∗prj)

  *Set up a prjprm struct for **Mollweide's (`MOL`)** projection.*
- int molx2s (PRJX2S_ARGS)

  *Cartesian-to-spherical transformation for **Mollweide's (`MOL`)** projection.*
- int mols2x (PRJS2X_ARGS)

  *Spherical-to-Cartesian transformation for **Mollweide's (`MOL`)** projection.*
- int aitset (struct prjprm ∗prj)

  *Set up a prjprm struct for the **Hammer-Aitoff (`AIT`)** projection.*
- int aitx2s (PRJX2S_ARGS)

  *Cartesian-to-spherical transformation for the **Hammer-Aitoff (`AIT`)** projection.*
- int aits2x (PRJS2X_ARGS)

  *Spherical-to-Cartesian transformation for the **Hammer-Aitoff (`AIT`)** projection.*
- int copset (struct prjprm ∗prj)

  *Set up a prjprm struct for the **conic perspective (`COP`)** projection.*
- int copx2s (PRJX2S_ARGS)

  *Cartesian-to-spherical transformation for the **conic perspective (`COP`)** projection.*
- int cops2x (PRJS2X_ARGS)

  *Spherical-to-Cartesian transformation for the **conic perspective (`COP`)** projection.*
- int coeset (struct prjprm ∗prj)

  *Set up a prjprm struct for the **conic equal area (`COE`)** projection.*
- int coex2s (PRJX2S_ARGS)

  *Cartesian-to-spherical transformation for the **conic equal area (`COE`)** projection.*
- int coes2x (PRJS2X_ARGS)

  *Spherical-to-Cartesian transformation for the **conic equal area (`COE`)** projection.*
- int codset (struct prjprm ∗prj)

  *Set up a prjprm struct for the **conic equidistant (`COD`)** projection.*
- int codx2s (PRJX2S_ARGS)

  *Cartesian-to-spherical transformation for the **conic equidistant (`COD`)** projection.*
- int cods2x (PRJS2X_ARGS)

  *Spherical-to-Cartesian transformation for the **conic equidistant (`COD`)** projection.*
- int cooset (struct prjprm ∗prj)

  *Set up a prjprm struct for the **conic orthomorphic (`COO`)** projection.*
- int coox2s (PRJX2S_ARGS)

  *Cartesian-to-spherical transformation for the **conic orthomorphic (`COO`)** projection.*
- int coos2x (PRJS2X_ARGS)

  *Spherical-to-Cartesian transformation for the **conic orthomorphic (`COO`)** projection.*
- int bonset (struct prjprm ∗prj)

  *Set up a prjprm struct for **Bonne's (`BON`)** projection.*
- int bonx2s (PRJX2S_ARGS)

  *Cartesian-to-spherical transformation for **Bonne's (`BON`)** projection.*
- int bons2x (PRJS2X_ARGS)

  *Spherical-to-Cartesian transformation for **Bonne's (`BON`)** projection.*
- int pcoset (struct prjprm ∗prj)

*Set up a [prjprm](#) struct for the **polyconic (PCO)** projection.*

- int [pcox2s](#) ([PRJX2S_ARGS](#))

    *Cartesian-to-spherical transformation for the **polyconic (PCO)** projection.*

- int [pcos2x](#) ([PRJS2X_ARGS](#))

    *Spherical-to-Cartesian transformation for the **polyconic (PCO)** projection.*

- int [tscset](#) (struct [prjprm](#) ∗prj)

    *Set up a [prjprm](#) struct for the **tangential spherical cube (TSC)** projection.*

- int [tscx2s](#) ([PRJX2S_ARGS](#))

    *Cartesian-to-spherical transformation for the **tangential spherical cube (TSC)** projection.*

- int [tscs2x](#) ([PRJS2X_ARGS](#))

    *Spherical-to-Cartesian transformation for the **tangential spherical cube (TSC)** projection.*

- int [cscset](#) (struct [prjprm](#) ∗prj)

    *Set up a [prjprm](#) struct for the **COBE spherical cube (CSC)** projection.*

- int [cscx2s](#) ([PRJX2S_ARGS](#))

    *Cartesian-to-spherical transformation for the **COBE spherical cube (CSC)** projection.*

- int [cscs2x](#) ([PRJS2X_ARGS](#))

    *Spherical-to-Cartesian transformation for the **COBE spherical cube (CSC)** projection.*

- int [qscset](#) (struct [prjprm](#) ∗prj)

    *Set up a [prjprm](#) struct for the **quadrilateralized spherical cube (QSC)** projection.*

- int [qscx2s](#) ([PRJX2S_ARGS](#))

    *Cartesian-to-spherical transformation for the **quadrilateralized spherical cube (QSC)** projection.*

- int [qscs2x](#) ([PRJS2X_ARGS](#))

    *Spherical-to-Cartesian transformation for the **quadrilateralized spherical cube (QSC)** projection.*

- int [hpxset](#) (struct [prjprm](#) ∗prj)

    *Set up a [prjprm](#) struct for the **HEALPix (HPX)** projection.*

- int [hpxx2s](#) ([PRJX2S_ARGS](#))

    *Cartesian-to-spherical transformation for the **HEALPix (HPX)** projection.*

- int [hpxs2x](#) ([PRJS2X_ARGS](#))

    *Spherical-to-Cartesian transformation for the **HEALPix (HPX)** projection.*

- int [xphset](#) (struct [prjprm](#) ∗prj)
- int [xphx2s](#) ([PRJX2S_ARGS](#))
- int [xphs2x](#) ([PRJS2X_ARGS](#))

Variables

- const char ∗ [prj_errmsg](#) [ ]

    Status return messages.

- const int [CONIC](#)

    Identifier for conic projections.

- const int [CONVENTIONAL](#)

    Identifier for conventional projections.

- const int [CYLINDRICAL](#)

    Identifier for cylindrical projections.

- const int [POLYCONIC](#)

    Identifier for polyconic projections.

- const int [PSEUDOCYLINDRICAL](#)

    Identifier for pseudocylindrical projections.

- const int [QUADCUBE](#)

    Identifier for quadcube projections.

- const int [ZENITHAL](#)

Identifier for zenithal/azimuthal projections.

- const int HEALPIX

  Identifier for the HEALPix projection.

- const char prj_categories [9][32]

  Projection categories.

- const int prj_ncode

  The number of recognized three-letter projection codes.

- const char prj_codes [28][4]

  Recognized three-letter projection codes.

19.13.1   Detailed Description

Routines in this suite implement the spherical map projections defined by the FITS World Coordinate System (WCS) standard, as described in

```
"Representations of world coordinates in FITS",
Greisen, E.W., & Calabretta, M.R. 2002, A&A, 395, 1061 (WCS Paper I)
"Representations of celestial coordinates in FITS",
Calabretta, M.R., & Greisen, E.W. 2002, A&A, 395, 1077 (WCS Paper II)
"Mapping on the HEALPix grid",
Calabretta, M.R., & Roukema, B.F. 2007, MNRAS, 381, 865 (WCS Paper V)
"Representing the 'Butterfly' Projection in FITS -- Projection Code XPH",
Calabretta, M.R., & Lowe, S.R. 2013, PASA, 30, e050 (WCS Paper VI)
```

These routines are based on the prjprm struct which contains all information needed for the computations. The struct contains some members that must be set by the user, and others that are maintained by these routines, somewhat like a C++ class but with no encapsulation.

Routine prjini() is provided to initialize the prjprm struct with default values, prjfree() reclaims any memory that may have been allocated to store an error message, prjsize() computes its total size including allocated memory, and prjprt() prints its contents.

prjperr() prints the error message(s) (if any) stored in a prjprm struct. prjbchk() performs bounds checking on native spherical coordinates.

Setup routines for each projection with names of the form **???set()**, where "???" is the down-cased three-letter projection code, compute intermediate values in the prjprm struct from parameters in it that were supplied by the user. The struct always needs to be set by the projection's setup routine but that need not be called explicitly - refer to the explanation of prjprm::flag.

Each map projection is implemented via separate functions for the spherical projection, **???s2x()**, and deprojection, **???x2s()**.

A set of driver routines, prjset(), prjx2s(), and prjs2x(), provides a generic interface to the specific projection routines which they invoke via pointers-to-functions stored in the prjprm struct.

In summary, the routines are:

- prjini() Initialization routine for the prjprm struct.

- prjfree() Reclaim memory allocated for error messages.

- prjsize() Compute total size of a prjprm struct.

- prjprt() Print a prjprm struct.

- prjperr() Print error message (if any).

- prjbchk() Bounds checking on native coordinates.

- prjset(), prjx2s(), prjs2x(): Generic driver routines

- azpset(), azpx2s(), azps2x(): **AZP** (zenithal/azimuthal perspective)

- szpset(), szpx2s(), szps2x(): **SZP** (slant zenithal perspective)

- tanset(), tanx2s(), tans2x(): **TAN** (gnomonic)

- stgset(), stgx2s(), stgs2x(): **STG** (stereographic)

- sinset(), sinx2s(), sins2x(): **SIN** (orthographic/synthesis)

- arcset(), arcx2s(), arcs2x(): **ARC** (zenithal/azimuthal equidistant)

- zpnset(), zpnx2s(), zpns2x(): **ZPN** (zenithal/azimuthal polynomial)

- zeaset(), zeax2s(), zeas2x(): **ZEA** (zenithal/azimuthal equal area)

- airset(), airx2s(), airs2x(): **AIR** (Airy)

- cypset(), cypx2s(), cyps2x(): **CYP** (cylindrical perspective)

- ceaset(), ceax2s(), ceas2x(): **CEA** (cylindrical equal area)

- carset(), carx2s(), cars2x(): **CAR** (Plate carée)

- merset(), merx2s(), mers2x(): **MER** (Mercator)

- sflset(), sflx2s(), sfls2x(): **SFL** (Sanson-Flamsteed)

- parset(), parx2s(), pars2x(): **PAR** (parabolic)

- molset(), molx2s(), mols2x(): **MOL** (Mollweide)

- aitset(), aitx2s(), aits2x(): **AIT** (Hammer-Aitoff)

- copset(), copx2s(), cops2x(): **COP** (conic perspective)

- coeset(), coex2s(), coes2x(): **COE** (conic equal area)

- codset(), codx2s(), cods2x(): **COD** (conic equidistant)

- cooset(), coox2s(), coos2x(): **COO** (conic orthomorphic)

- bonset(), bonx2s(), bons2x(): **BON** (Bonne)

- pcoset(), pcox2s(), pcos2x(): **PCO** (polyconic)

- tscset(), tscx2s(), tscs2x(): **TSC** (tangential spherical cube)

- cscset(), cscx2s(), cscs2x(): **CSC** (COBE spherical cube)

- qscset(), qscx2s(), qscs2x(): **QSC** (quadrilateralized spherical cube)

- hpxset(), hpxx2s(), hpxs2x(): **HPX** (HEALPix)

- xphset(), xphx2s(), xphs2x(): XPH (HEALPix polar, aka "butterfly")

Argument checking (projection routines):

The values of ϕ and θ (the native longitude and latitude) normally lie in the range $[-180°, 180°]$ for ϕ, and $[-90°, 90°]$ for θ. However, all projection routines will accept any value of ϕ and will not normalize it.

The projection routines do not explicitly check that θ lies within the range $[-90°, 90°]$. They do check for any value of θ that produces an invalid argument to the projection equations (e.g. leading to division by zero). The projection routines for **AZP**, **SZP**, **TAN**, **SIN**, **ZPN**, and **COP** also return error 2 if (ϕ, θ) corresponds to the overlapped (far) side of the projection but also return the corresponding value of (x, y). This strict bounds checking may be relaxed at any time by setting prjprm::bounds%2 to 0 (rather than 1); the projections need not be reinitialized.

Argument checking (deprojection routines):
Error checking on the projected coordinates (x, y) is limited to that required to ascertain whether a solution exists. Where a solution does exist, an optional check is made that the value of ϕ and θ obtained lie within the ranges $[-180°, 180°]$ for ϕ, and $[-90°, 90°]$ for θ. This check, performed by prjbchk(), is enabled by default. It may be disabled by setting prjprm::bounds%4 to 0 (rather than 1); the projections need not be reinitialized.

Accuracy:
No warranty is given for the accuracy of these routines (refer to the copyright notice); intending users must satisfy for themselves their adequacy for the intended purpose. However, closure to a precision of at least $0°.0000000001$ of longitude and latitude has been verified for typical projection parameters on the $1°$ degree graticule of native longitude and latitude (to within $5°$ of any latitude where the projection may diverge). Refer to the tprj1.c and tprj2.c test routines that accompany this software.

19.13.2 Macro Definition Documentation

19.13.2.1 PVN `#define PVN 30`

The total number of projection parameters numbered 0 to **PVN**-1.

19.13.2.2 PRJX2S_ARGS `#define PRJX2S_ARGS`

Value:
```
struct prjprm *prj, int nx, int ny, int sxy, int spt, \
const double x[], const double y[], double phi[], double theta[], int stat[]
```

Preprocessor macro used for declaring deprojection function prototypes.

19.13.2.3 PRJS2X_ARGS `#define PRJS2X_ARGS`

Value:
```
struct prjprm *prj, int nx, int ny, int sxy, int spt, \
const double phi[], const double theta[], double x[], double y[], int stat[]
```

Preprocessor macro used for declaring projection function prototypes.

19.13.2.4 PRJLEN `#define PRJLEN (sizeof(struct prjprm)/sizeof(int))`

Size of the prjprm struct in *int* units, used by the Fortran wrappers.

19.13.2.5 prjini_errmsg `#define prjini_errmsg prj_errmsg`

Deprecated Added for backwards compatibility, use prj_errmsg directly now instead.

19.13.2.6   prjprt_errmsg   #define prjprt_errmsg `prj_errmsg`

Deprecated Added for backwards compatibility, use prj_errmsg directly now instead.

19.13.2.7   prjset_errmsg   #define prjset_errmsg `prj_errmsg`

Deprecated Added for backwards compatibility, use prj_errmsg directly now instead.

19.13.2.8   prjx2s_errmsg   #define prjx2s_errmsg `prj_errmsg`

Deprecated Added for backwards compatibility, use prj_errmsg directly now instead.

19.13.2.9   prjs2x_errmsg   #define prjs2x_errmsg `prj_errmsg`

Deprecated Added for backwards compatibility, use prj_errmsg directly now instead.

19.13.3   Enumeration Type Documentation

19.13.3.1   prj_errmsg_enum   enum `prj_errmsg_enum`

Enumerator

PRJERR_SUCCESS	
PRJERR_NULL_POINTER	
PRJERR_BAD_PARAM	
PRJERR_BAD_PIX	
PRJERR_BAD_WORLD	

19.13.4   Function Documentation

19.13.4.1 prjini() `int prjini (`
        `struct prjprm * prj )`

prjini() sets all members of a prjprm struct to default values. It should be used to initialize every prjprm struct.

PLEASE NOTE: If the prjprm struct has already been initialized, then before reinitializing, it prjfree() should be used to free any memory that may have been allocated to store an error message. A memory leak may otherwise result.

Parameters

out	*prj*	Projection parameters.

Returns

>   Status return value:

>   • 0: Success.

>   • 1: Null prjprm pointer passed.

19.13.4.2 prjfree() `int prjfree (`
        `struct prjprm * prj )`

prjfree() frees any memory that may have been allocated to store an error message in the prjprm struct.

Parameters

in	*prj*	Projection parameters.

Returns

>   Status return value:

>   • 0: Success.

>   • 1: Null prjprm pointer passed.

19.13.4.3 prjsize() `int prjsize (`
        `const struct prjprm * prj,`
        `int sizes[2] )`

prjsize() computes the full size of a prjprm struct, including allocated memory.

Parameters

in	*prj*	Projection parameters.
		If NULL, the base size of the struct and the allocated size are both set to zero.
out	*sizes*	The first element is the base size of the struct as returned by sizeof(struct prjprm). The second element is the total allocated size, in bytes. This figure includes memory allocated for the constituent struct, prjprm::err.
		It is not an error for the struct not to have been set up via prjset().

Returns

Status return value:

- 0: Success.

19.13.4.4   prjprt() `int prjprt (`
            `const struct prjprm * prj )`

prjprt() prints the contents of a prjprm struct using wcsprintf(). Mainly intended for diagnostic purposes.

Parameters

in	*prj*	Projection parameters.

Returns

Status return value:

- 0: Success.
- 1: Null prjprm pointer passed.

19.13.4.5   prjperr() `int prjperr (`
            `const struct prjprm * prj,`
            `const char * prefix )`

prjperr() prints the error message(s) (if any) stored in a prjprm struct. If there are no errors then nothing is printed. It uses wcserr_prt(), q.v.

Parameters

in	*prj*	Projection parameters.
in	*prefix*	If non-NULL, each output line will be prefixed with this string.

Returns

Status return value:

- 0: Success.
- 1: Null prjprm pointer passed.

19.13.4.6   prjbchk() `int prjbchk (`
            `double tol,`
            `int nphi,`

```
            int ntheta,
            int spt,
            double phi[],
            double theta[],
            int stat[] )
```

prjbchk() performs bounds checking on native spherical coordinates. As returned by the deprojection (x2s) routines, native longitude is expected to lie in the closed interval $[-180°, 180°]$, with latitude in $[-90°, 90°]$.

A tolerance may be specified to provide a small allowance for numerical imprecision. Values that lie outside the allowed range by not more than the specified tolerance will be adjusted back into range.

If prjprm::bounds&4 is set, as it is by prjini(), then **prjbchk**() will be invoked automatically by the Cartesian-to-spherical deprojection (x2s) routines with an appropriate tolerance set for each projection.

Parameters

in	*tol*	Tolerance for the bounds check [deg].
in	*nphi,ntheta*	Vector lengths.
in	*spt*	Vector stride.
in, out	*phi,theta*	Native longitude and latitude (ϕ, θ) [deg].
out	*stat*	Status value for each vector element: • 0: Valid value of (ϕ, θ). • 1: Invalid value.

Returns

Status return value:

- 0: Success.
- 1: One or more of the (ϕ, θ) coordinates were, invalid, as indicated by the stat vector.

19.13.4.7 prjset() `int prjset (`
            `struct prjprm * prj )`

prjset() sets up a prjprm struct according to information supplied within it.

Note that this routine need not be called directly; it will be invoked by prjx2s() and prjs2x() if prj.flag is anything other than a predefined magic value.

The one important distinction between **prjset**() and the setup routines for the specific projections is that the projection code must be defined in the prjprm struct in order for **prjset**() to identify the required projection. Once **prjset**() has initialized the prjprm struct, prjx2s() and prjs2x() use the pointers to the specific projection and deprojection routines contained therein.

Parameters

in, out	*prj*	Projection parameters.

Returns

Status return value:

- 0: Success.

- 1: Null prjprm pointer passed.

- 2: Invalid projection parameters.

For returns > 1, a detailed error message is set in prjprm::err if enabled, see wcserr_enable().

19.13.4.8 prjx2s() `int prjx2s (`
                    `PRJX2S_ARGS )`

Deproject Cartesian (x, y) coordinates in the plane of projection to native spherical coordinates (ϕ, θ).

The projection is that specified by prjprm::code.

Parameters

`in,out`	*prj*	Projection parameters.
`in`	*nx,ny*	Vector lengths.
`in`	*sxy,spt*	Vector strides.
`in`	*x,y*	Projected coordinates.
`out`	*phi,theta*	Longitude and latitude (ϕ, θ) of the projected point in native spherical coordinates [deg].
`out`	*stat*	Status value for each vector element: • 0: Success. • 1: Invalid value of (x, y).

Returns

Status return value:

- 0: Success.
- 1: Null prjprm pointer passed.
- 2: Invalid projection parameters.
- 3: One or more of the (x, y) coordinates were invalid, as indicated by the stat vector.

For returns > 1, a detailed error message is set in prjprm::err if enabled, see wcserr_enable().

19.13.4.9 prjs2x() `int prjs2x (`
                    `PRJS2X_ARGS )`

Project native spherical coordinates (ϕ, θ) to Cartesian (x, y) coordinates in the plane of projection.

The projection is that specified by prjprm::code.

Parameters

in,out	*prj*	Projection parameters.
in	*nphi,ntheta*	Vector lengths.
in	*spt,sxy*	Vector strides.
in	*phi,theta*	Longitude and latitude (ϕ, θ) of the projected point in native spherical coordinates [deg].
out	*x,y*	Projected coordinates.
out	*stat*	Status value for each vector element: • 0: Success. • 1: Invalid value of (ϕ, θ).

Returns

Status return value:

- 0: Success.
- 1: Null prjprm pointer passed.
- 2: Invalid projection parameters.
- 4: One or more of the (ϕ, θ) coordinates were, invalid, as indicated by the stat vector.

For returns > 1, a detailed error message is set in prjprm::err if enabled, see wcserr_enable().

19.13.4.10 azpset() int azpset (
         struct prjprm * *prj*)

azpset() sets up a prjprm struct for a **zenithal/azimuthal perspective (AZP)** projection.

See prjset() for a description of the API.

19.13.4.11 azpx2s() int azpx2s (
         PRJX2S_ARGS)

azpx2s() deprojects Cartesian (x, y) coordinates in the plane of a **zenithal/azimuthal perspective (AZP)** projection to native spherical coordinates (ϕ, θ).

See prjx2s() for a description of the API.

19.13.4.12 azps2x() int azps2x (
         PRJS2X_ARGS)

azps2x() projects native spherical coordinates (ϕ, θ) to Cartesian (x, y) coordinates in the plane of a **zenithal/azimuthal perspective (AZP)** projection.

See prjs2x() for a description of the API.

19.13.4.13 szpset() `int szpset (`
   `struct prjprm * prj )`

szpset() sets up a [prjprm](#) struct for a **slant zenithal perspective (SZP)** projection.

See [prjset()](#) for a description of the API.

19.13.4.14 szpx2s() `int szpx2s (`
   `PRJX2S_ARGS )`

szpx2s() deprojects Cartesian (x, y) coordinates in the plane of a **slant zenithal perspective (SZP)** projection to native spherical coordinates (ϕ, θ).

See [prjx2s()](#) for a description of the API.

19.13.4.15 szps2x() `int szps2x (`
   `PRJS2X_ARGS )`

szps2x() projects native spherical coordinates (ϕ, θ) to Cartesian (x, y) coordinates in the plane of a **slant zenithal perspective (SZP)** projection.

See [prjs2x()](#) for a description of the API.

19.13.4.16 tanset() `int tanset (`
   `struct prjprm * prj )`

tanset() sets up a [prjprm](#) struct for a **gnomonic (TAN)** projection.

See [prjset()](#) for a description of the API.

19.13.4.17 tanx2s() `int tanx2s (`
   `PRJX2S_ARGS )`

tanx2s() deprojects Cartesian (x, y) coordinates in the plane of a **gnomonic (TAN)** projection to native spherical coordinates (ϕ, θ).

See [prjx2s()](#) for a description of the API.

19.13.4.18 tans2x() `int tans2x (`
   `PRJS2X_ARGS )`

tans2x() projects native spherical coordinates (ϕ, θ) to Cartesian (x, y) coordinates in the plane of a **gnomonic (TAN)** projection.

See [prjs2x()](#) for a description of the API.

19.13.4.19 stgset() `int stgset (`
   `struct prjprm * prj )`

stgset() sets up a [prjprm](#) struct for a **stereographic (STG)** projection.

See [prjset()](#) for a description of the API.

19.13.4.20 stgx2s() `int stgx2s (`
`PRJX2S_ARGS )`

stgx2s() deprojects Cartesian (x, y) coordinates in the plane of a **stereographic (STG)** projection to native spherical coordinates (ϕ, θ).

See prjx2s() for a description of the API.

19.13.4.21 stgs2x() `int stgs2x (`
`PRJS2X_ARGS )`

stgs2x() projects native spherical coordinates (ϕ, θ) to Cartesian (x, y) coordinates in the plane of a **stereographic (STG)** projection.

See prjs2x() for a description of the API.

19.13.4.22 sinset() `int sinset (`
`struct prjprm * prj )`

stgset() sets up a prjprm struct for an **orthographic/synthesis (SIN)** projection.

See prjset() for a description of the API.

19.13.4.23 sinx2s() `int sinx2s (`
`PRJX2S_ARGS )`

sinx2s() deprojects Cartesian (x, y) coordinates in the plane of an **orthographic/synthesis (SIN)** projection to native spherical coordinates (ϕ, θ).

See prjx2s() for a description of the API.

19.13.4.24 sins2x() `int sins2x (`
`PRJS2X_ARGS )`

sins2x() projects native spherical coordinates (ϕ, θ) to Cartesian (x, y) coordinates in the plane of an **orthographic/synthesis (SIN)** projection.

See prjs2x() for a description of the API.

19.13.4.25 arcset() `int arcset (`
`struct prjprm * prj )`

arcset() sets up a prjprm struct for a **zenithal/azimuthal equidistant (ARC)** projection.

See prjset() for a description of the API.

19.13.4.26 arcx2s() `int arcx2s (`
`PRJX2S_ARGS )`

arcx2s() deprojects Cartesian (x, y) coordinates in the plane of a **zenithal/azimuthal equidistant (ARC)** projection to native spherical coordinates (ϕ, θ).

See prjx2s() for a description of the API.

19.13.4.27 arcs2x() `int arcs2x (`
           `PRJS2X_ARGS )`

arcs2x() projects native spherical coordinates (ϕ, θ) to Cartesian (x, y) coordinates in the plane of a **zenithal/azimuthal equidistant (ARC)** projection.

See prjs2x() for a description of the API.

19.13.4.28 zpnset() `int zpnset (`
           `struct prjprm * prj )`

zpnset() sets up a prjprm struct for a **zenithal/azimuthal polynomial (ZPN)** projection.

See prjset() for a description of the API.

19.13.4.29 zpnx2s() `int zpnx2s (`
           `PRJX2S_ARGS )`

zpnx2s() deprojects Cartesian (x, y) coordinates in the plane of a **zenithal/azimuthal polynomial (ZPN)** projection to native spherical coordinates (ϕ, θ).

See prjx2s() for a description of the API.

19.13.4.30 zpns2x() `int zpns2x (`
           `PRJS2X_ARGS )`

zpns2x() projects native spherical coordinates (ϕ, θ) to Cartesian (x, y) coordinates in the plane of a **zenithal/azimuthal polynomial (ZPN)** projection.

See prjs2x() for a description of the API.

19.13.4.31 zeaset() `int zeaset (`
           `struct prjprm * prj )`

zeaset() sets up a prjprm struct for a **zenithal/azimuthal equal area (ZEA)** projection.

See prjset() for a description of the API.

19.13.4.32 zeax2s() `int zeax2s (`
           `PRJX2S_ARGS )`

zeax2s() deprojects Cartesian (x, y) coordinates in the plane of a **zenithal/azimuthal equal area (ZEA)** projection to native spherical coordinates (ϕ, θ).

See prjx2s() for a description of the API.

19.13.4.33 zeas2x() `int zeas2x (`
           `PRJS2X_ARGS )`

zeas2x() projects native spherical coordinates (ϕ, θ) to Cartesian (x, y) coordinates in the plane of a **zenithal/azimuthal equal area (ZEA)** projection.

See prjs2x() for a description of the API.

19.13.4.34 airset() `int airset (`
        `struct prjprm * prj )`

airset() sets up a prjprm struct for an **Airy (AIR)** projection.

See prjset() for a description of the API.

19.13.4.35 airx2s() `int airx2s (`
        `PRJX2S_ARGS )`

airx2s() deprojects Cartesian (x, y) coordinates in the plane of an **Airy (AIR)** projection to native spherical coordinates (ϕ, θ).

See prjx2s() for a description of the API.

19.13.4.36 airs2x() `int airs2x (`
        `PRJS2X_ARGS )`

airs2x() projects native spherical coordinates (ϕ, θ) to Cartesian (x, y) coordinates in the plane of an **Airy (AIR)** projection.

See prjs2x() for a description of the API.

19.13.4.37 cypset() `int cypset (`
        `struct prjprm * prj )`

cypset() sets up a prjprm struct for a **cylindrical perspective (CYP)** projection.

See prjset() for a description of the API.

19.13.4.38 cypx2s() `int cypx2s (`
        `PRJX2S_ARGS )`

cypx2s() deprojects Cartesian (x, y) coordinates in the plane of a **cylindrical perspective (CYP)** projection to native spherical coordinates (ϕ, θ).

See prjx2s() for a description of the API.

19.13.4.39 cyps2x() `int cyps2x (`
        `PRJS2X_ARGS )`

cyps2x() projects native spherical coordinates (ϕ, θ) to Cartesian (x, y) coordinates in the plane of a **cylindrical perspective (CYP)** projection.

See prjs2x() for a description of the API.

19.13.4.40 ceaset() `int ceaset (`
        `struct prjprm * prj )`

ceaset() sets up a prjprm struct for a **cylindrical equal area (CEA)** projection.

See prjset() for a description of the API.

19.13.4.41   ceax2s() `int ceax2s (`
            `PRJX2S_ARGS  )`

ceax2s() deprojects Cartesian (x, y) coordinates in the plane of a **cylindrical equal area (CEA)** projection to native spherical coordinates (ϕ, θ).

See prjx2s() for a description of the API.

19.13.4.42   ceas2x() `int ceas2x (`
            `PRJS2X_ARGS  )`

ceas2x() projects native spherical coordinates (ϕ, θ) to Cartesian (x, y) coordinates in the plane of a **cylindrical equal area (CEA)** projection.

See prjs2x() for a description of the API.

19.13.4.43   carset() `int carset (`
            `struct prjprm * prj )`

carset() sets up a prjprm struct for a **plate carrée (CAR)** projection.

See prjset() for a description of the API.

19.13.4.44   carx2s() `int carx2s (`
            `PRJX2S_ARGS  )`

carx2s() deprojects Cartesian (x, y) coordinates in the plane of a **plate carrée (CAR)** projection to native spherical coordinates (ϕ, θ).

See prjx2s() for a description of the API.

19.13.4.45   cars2x() `int cars2x (`
            `PRJS2X_ARGS  )`

cars2x() projects native spherical coordinates (ϕ, θ) to Cartesian (x, y) coordinates in the plane of a **plate carrée (CAR)** projection.

See prjs2x() for a description of the API.

19.13.4.46   merset() `int merset (`
            `struct prjprm * prj )`

merset() sets up a prjprm struct for a **Mercator (MER)** projection.

See prjset() for a description of the API.

19.13.4.47   merx2s() `int merx2s (`
            `PRJX2S_ARGS  )`

merx2s() deprojects Cartesian (x, y) coordinates in the plane of a **Mercator (MER)** projection to native spherical coordinates (ϕ, θ).

See prjx2s() for a description of the API.

19.13.4.48 mers2x() `int mers2x (`
        `PRJS2X_ARGS  )`

mers2x() projects native spherical coordinates (ϕ, θ) to Cartesian (x, y) coordinates in the plane of a **Mercator (MER)** projection.

See prjs2x() for a description of the API.

19.13.4.49 sflset() `int sflset (`
        `struct prjprm * prj )`

sflset() sets up a prjprm struct for a **Sanson-Flamsteed (SFL)** projection.

See prjset() for a description of the API.

19.13.4.50 sflx2s() `int sflx2s (`
        `PRJX2S_ARGS  )`

sflx2s() deprojects Cartesian (x, y) coordinates in the plane of a **Sanson-Flamsteed (SFL)** projection to native spherical coordinates (ϕ, θ).

See prjx2s() for a description of the API.

19.13.4.51 sfls2x() `int sfls2x (`
        `PRJS2X_ARGS  )`

sfls2x() projects native spherical coordinates (ϕ, θ) to Cartesian (x, y) coordinates in the plane of a **Sanson-↩ Flamsteed (SFL)** projection.

See prjs2x() for a description of the API.

19.13.4.52 parset() `int parset (`
        `struct prjprm * prj )`

parset() sets up a prjprm struct for a **parabolic (PAR)** projection.

See prjset() for a description of the API.

19.13.4.53 parx2s() `int parx2s (`
        `PRJX2S_ARGS  )`

parx2s() deprojects Cartesian (x, y) coordinates in the plane of a **parabolic (PAR)** projection to native spherical coordinates (ϕ, θ).

See prjx2s() for a description of the API.

19.13.4.54 pars2x() `int pars2x (`
        `PRJS2X_ARGS  )`

pars2x() projects native spherical coordinates (ϕ, θ) to Cartesian (x, y) coordinates in the plane of a **parabolic (PAR)** projection.

See prjs2x() for a description of the API.

19.13.4.55 molset() `int molset (`
`            struct prjprm * prj )`

molset() sets up a prjprm struct for a **Mollweide (MOL)** projection.

See prjset() for a description of the API.

19.13.4.56 molx2s() `int molx2s (`
`            PRJX2S_ARGS )`

molx2s() deprojects Cartesian (x, y) coordinates in the plane of a **Mollweide (MOL)** projection to native spherical coordinates (ϕ, θ).

See prjx2s() for a description of the API.

19.13.4.57 mols2x() `int mols2x (`
`            PRJS2X_ARGS )`

mols2x() projects native spherical coordinates (ϕ, θ) to Cartesian (x, y) coordinates in the plane of a **Mollweide (MOL)** projection.

See prjs2x() for a description of the API.

19.13.4.58 aitset() `int aitset (`
`            struct prjprm * prj )`

aitset() sets up a prjprm struct for a **Hammer-Aitoff (AIT)** projection.

See prjset() for a description of the API.

19.13.4.59 aitx2s() `int aitx2s (`
`            PRJX2S_ARGS )`

aitx2s() deprojects Cartesian (x, y) coordinates in the plane of a **Hammer-Aitoff (AIT)** projection to native spherical coordinates (ϕ, θ).

See prjx2s() for a description of the API.

19.13.4.60 aits2x() `int aits2x (`
`            PRJS2X_ARGS )`

aits2x() projects native spherical coordinates (ϕ, θ) to Cartesian (x, y) coordinates in the plane of a **Hammer-Aitoff (AIT)** projection.

See prjs2x() for a description of the API.

19.13.4.61 copset() `int copset (`
`            struct prjprm * prj )`

copset() sets up a prjprm struct for a **conic perspective (COP)** projection.

See prjset() for a description of the API.

19.13.4.62 copx2s() `int copx2s (`
　　　　`PRJX2S_ARGS )`

copx2s() deprojects Cartesian (x, y) coordinates in the plane of a **conic perspective (COP)** projection to native spherical coordinates (ϕ, θ).

See prjx2s() for a description of the API.

19.13.4.63 cops2x() `int cops2x (`
　　　　`PRJS2X_ARGS )`

cops2x() projects native spherical coordinates (ϕ, θ) to Cartesian (x, y) coordinates in the plane of a **conic perspective (COP)** projection.

See prjs2x() for a description of the API.

19.13.4.64 coeset() `int coeset (`
　　　　`struct prjprm * prj )`

coeset() sets up a prjprm struct for a **conic equal area (COE)** projection.

See prjset() for a description of the API.

19.13.4.65 coex2s() `int coex2s (`
　　　　`PRJX2S_ARGS )`

coex2s() deprojects Cartesian (x, y) coordinates in the plane of a **conic equal area (COE)** projection to native spherical coordinates (ϕ, θ).

See prjx2s() for a description of the API.

19.13.4.66 coes2x() `int coes2x (`
　　　　`PRJS2X_ARGS )`

coes2x() projects native spherical coordinates (ϕ, θ) to Cartesian (x, y) coordinates in the plane of a **conic equal area (COE)** projection.

See prjs2x() for a description of the API.

19.13.4.67 codset() `int codset (`
　　　　`struct prjprm * prj )`

codset() sets up a prjprm struct for a **conic equidistant (COD)** projection.

See prjset() for a description of the API.

19.13.4.68 codx2s() `int codx2s (`
　　　　`PRJX2S_ARGS )`

codx2s() deprojects Cartesian (x, y) coordinates in the plane of a **conic equidistant (COD)** projection to native spherical coordinates (ϕ, θ).

See prjx2s() for a description of the API.

19.13.4.69   cods2x()   `int cods2x (`
              `PRJS2X_ARGS   )`

cods2x() projects native spherical coordinates (ϕ, θ) to Cartesian (x, y) coordinates in the plane of a **conic equidistant (COD)** projection.

See prjs2x() for a description of the API.

19.13.4.70   cooset()   `int cooset (`
              `struct prjprm * prj )`

cooset() sets up a prjprm struct for a **conic orthomorphic (COO)** projection.

See prjset() for a description of the API.

19.13.4.71   coox2s()   `int coox2s (`
              `PRJX2S_ARGS   )`

coox2s() deprojects Cartesian (x, y) coordinates in the plane of a **conic orthomorphic (COO)** projection to native spherical coordinates (ϕ, θ).

See prjx2s() for a description of the API.

19.13.4.72   coos2x()   `int coos2x (`
              `PRJS2X_ARGS   )`

coos2x() projects native spherical coordinates (ϕ, θ) to Cartesian (x, y) coordinates in the plane of a **conic orthomorphic (COO)** projection.

See prjs2x() for a description of the API.

19.13.4.73   bonset()   `int bonset (`
              `struct prjprm * prj )`

bonset() sets up a prjprm struct for a **Bonne (BON)** projection.

See prjset() for a description of the API.

19.13.4.74   bonx2s()   `int bonx2s (`
              `PRJX2S_ARGS   )`

bonx2s() deprojects Cartesian (x, y) coordinates in the plane of a **Bonne (BON)** projection to native spherical coordinates (ϕ, θ).

See prjx2s() for a description of the API.

19.13.4.75   bons2x()   `int bons2x (`
              `PRJS2X_ARGS   )`

bons2x() projects native spherical coordinates (ϕ, θ) to Cartesian (x, y) coordinates in the plane of a **Bonne (BON)** projection.

See prjs2x() for a description of the API.

19.13.4.76 pcoset() `int pcoset (`
           `struct prjprm * prj )`

pcoset() sets up a prjprm struct for a **polyconic (PCO)** projection.

See prjset() for a description of the API.

19.13.4.77 pcox2s() `int pcox2s (`
           `PRJX2S_ARGS )`

pcox2s() deprojects Cartesian (x, y) coordinates in the plane of a **polyconic (PCO)** projection to native spherical coordinates (ϕ, θ).

See prjx2s() for a description of the API.

19.13.4.78 pcos2x() `int pcos2x (`
           `PRJS2X_ARGS )`

pcos2x() projects native spherical coordinates (ϕ, θ) to Cartesian (x, y) coordinates in the plane of a **polyconic (PCO)** projection.

See prjs2x() for a description of the API.

19.13.4.79 tscset() `int tscset (`
           `struct prjprm * prj )`

tscset() sets up a prjprm struct for a **tangential spherical cube (TSC)** projection.

See prjset() for a description of the API.

19.13.4.80 tscx2s() `int tscx2s (`
           `PRJX2S_ARGS )`

tscx2s() deprojects Cartesian (x, y) coordinates in the plane of a **tangential spherical cube (TSC)** projection to native spherical coordinates (ϕ, θ).

See prjx2s() for a description of the API.

19.13.4.81 tscs2x() `int tscs2x (`
           `PRJS2X_ARGS )`

tscs2x() projects native spherical coordinates (ϕ, θ) to Cartesian (x, y) coordinates in the plane of a **tangential spherical cube (TSC)** projection.

See prjs2x() for a description of the API.

19.13.4.82 cscset() `int cscset (`
           `struct prjprm * prj )`

cscset() sets up a prjprm struct for a **COBE spherical cube (CSC)** projection.

See prjset() for a description of the API.

19.13.4.83 cscx2s() `int cscx2s (`
  `PRJX2S_ARGS  )`

cscx2s() deprojects Cartesian (x, y) coordinates in the plane of a **COBE spherical cube (CSC)** projection to native spherical coordinates (ϕ, θ).

See prjx2s() for a description of the API.

19.13.4.84 cscs2x() `int cscs2x (`
  `PRJS2X_ARGS  )`

cscs2x() projects native spherical coordinates (ϕ, θ) to Cartesian (x, y) coordinates in the plane of a **COBE spherical cube (CSC)** projection.

See prjs2x() for a description of the API.

19.13.4.85 qscset() `int qscset (`
  `struct prjprm * prj )`

qscset() sets up a prjprm struct for a **quadrilateralized spherical cube (QSC)** projection.

See prjset() for a description of the API.

19.13.4.86 qscx2s() `int qscx2s (`
  `PRJX2S_ARGS  )`

qscx2s() deprojects Cartesian (x, y) coordinates in the plane of a **quadrilateralized spherical cube (QSC)** projection to native spherical coordinates (ϕ, θ).

See prjx2s() for a description of the API.

19.13.4.87 qscs2x() `int qscs2x (`
  `PRJS2X_ARGS  )`

qscs2x() projects native spherical coordinates (ϕ, θ) to Cartesian (x, y) coordinates in the plane of a **quadrilateralized spherical cube (QSC)** projection.

See prjs2x() for a description of the API.

19.13.4.88 hpxset() `int hpxset (`
  `struct prjprm * prj )`

hpxset() sets up a prjprm struct for a **HEALPix (HPX)** projection.

See prjset() for a description of the API.

19.13.4.89 hpxx2s() `int hpxx2s (`
  `PRJX2S_ARGS  )`

hpxx2s() deprojects Cartesian (x, y) coordinates in the plane of a **HEALPix (HPX)** projection to native spherical coordinates (ϕ, θ).

See prjx2s() for a description of the API.

19.13.4.90 hpxs2x() `int hpxs2x (`
          `PRJS2X_ARGS  )`

hpxs2x() projects native spherical coordinates (ϕ, θ) to Cartesian (x, y) coordinates in the plane of a **HEALPix (HPX)** projection.

See [prjs2x()](#) for a description of the API.

19.13.4.91 xphset() `int xphset (`
          `struct prjprm * prj )`

19.13.4.92 xphx2s() `int xphx2s (`
          `PRJX2S_ARGS  )`

19.13.4.93 xphs2x() `int xphs2x (`
          `PRJS2X_ARGS  )`

19.13.5 Variable Documentation

19.13.5.1 prj_errmsg `const char * prj_errmsg[]  [extern]`

Error messages to match the status value returned from each function.

19.13.5.2 CONIC `const int CONIC  [extern]`

Identifier for conic projections, see [prjprm::category](#).

19.13.5.3 CONVENTIONAL `const int CONVENTIONAL`

Identifier for conventional projections, see [prjprm::category](#).

19.13.5.4 CYLINDRICAL `const int CYLINDRICAL`

Identifier for cylindrical projections, see [prjprm::category](#).

19.13.5.5 POLYCONIC `const int POLYCONIC`

Identifier for polyconic projections, see [prjprm::category](#).

19.13.5.6   PSEUDOCYLINDRICAL `const int PSEUDOCYLINDRICAL`

Identifier for pseudocylindrical projections, see [prjprm::category](#).

19.13.5.7   QUADCUBE `const int QUADCUBE`

Identifier for quadcube projections, see [prjprm::category](#).

19.13.5.8   ZENITHAL `const int ZENITHAL`

Identifier for zenithal/azimuthal projections, see [prjprm::category](#).

19.13.5.9   HEALPIX `const int HEALPIX`

Identifier for the HEALPix projection, see [prjprm::category](#).

19.13.5.10   prj_categories `const char prj_categories[9][32]` `[extern]`

Names of the projection categories, all in lower-case except for "HEALPix".

Provided for information only, not used by the projection routines.

19.13.5.11   prj_ncode `const int prj_ncode` `[extern]`

The number of recognized three-letter projection codes (currently 27), see [prj_codes](#).

19.13.5.12   prj_codes `const char prj_codes[27][4]` `[extern]`

List of all recognized three-letter projection codes (currently 27), e.g. **SIN**, **TAN**, etc.

19.14   prj.h

[Go to the documentation of this file.](#)
```
1  /*============================================================================
2    WCSLIB 7.11 - an implementation of the FITS WCS standard.
3    Copyright (C) 1995-2022, Mark Calabretta
4
5    This file is part of WCSLIB.
6
7    WCSLIB is free software: you can redistribute it and/or modify it under the
8    terms of the GNU Lesser General Public License as published by the Free
9    Software Foundation, either version 3 of the License, or (at your option)
10   any later version.
11
12   WCSLIB is distributed in the hope that it will be useful, but WITHOUT ANY
13   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
14   FOR A PARTICULAR PURPOSE.  See the GNU Lesser General Public License for
15   more details.
16
17   You should have received a copy of the GNU Lesser General Public License
18   along with WCSLIB.  If not, see http://www.gnu.org/licenses.
19
20   Author: Mark Calabretta, Australia Telescope National Facility, CSIRO.
21   http://www.atnf.csiro.au/people/Mark.Calabretta
22   $Id: prj.h,v 7.11 2022/04/26 06:13:52 mcalabre Exp $
23 *============================================================================
24 *
25 * WCSLIB 7.11 - C routines that implement the FITS World Coordinate System
```

```
 26 * (WCS) standard.  Refer to the README file provided with WCSLIB for an
 27 * overview of the library.
 28 *
 29 *
 30 * Summary of the prj routines
 31 * ---------------------------
 32 * Routines in this suite  implement the spherical map projections defined by
 33 * the FITS World Coordinate System (WCS) standard, as described in
 34 *
 35 =   "Representations of world coordinates in FITS",
 36 =   Greisen, E.W., & Calabretta, M.R. 2002, A&A, 395, 1061 (WCS Paper I)
 37 =
 38 =   "Representations of celestial coordinates in FITS",
 39 =   Calabretta, M.R., & Greisen, E.W. 2002, A&A, 395, 1077 (WCS Paper II)
 40 =
 41 =   "Mapping on the HEALPix grid",
 42 =   Calabretta, M.R., & Roukema, B.F. 2007, MNRAS, 381, 865 (WCS Paper V)
 43 =
 44 =   "Representing the 'Butterfly' Projection in FITS -- Projection Code XPH",
 45 =   Calabretta, M.R., & Lowe, S.R. 2013, PASA, 30, e050 (WCS Paper VI)
 46 *
 47 * These routines are based on the prjprm struct which contains all information
 48 * needed for the computations.  The struct contains some members that must be
 49 * set by the user, and others that are maintained by these routines, somewhat
 50 * like a C++ class but with no encapsulation.
 51 *
 52 * Routine prjini() is provided to initialize the prjprm struct with default
 53 * values, prjfree() reclaims any memory that may have been allocated to store
 54 * an error message, prjsize() computes its total size including allocated
 55 * memory, and prjprt() prints its contents.
 56 *
 57 * prjperr() prints the error message(s) (if any) stored in a prjprm struct.
 58 * prjbchk() performs bounds checking on native spherical coordinates.
 59 *
 60 * Setup routines for each projection with names of the form ???set(), where
 61 * "???" is the down-cased three-letter projection code, compute intermediate
 62 * values in the prjprm struct from parameters in it that were supplied by the
 63 * user.  The struct always needs to be set by the projection's setup routine
 64 * but that need not be called explicitly - refer to the explanation of
 65 * prjprm::flag.
 66 *
 67 * Each map projection is implemented via separate functions for the spherical
 68 * projection, ???s2x(), and deprojection, ???x2s().
 69 *
 70 * A set of driver routines, prjset(), prjx2s(), and prjs2x(), provides a
 71 * generic interface to the specific projection routines which they invoke
 72 * via pointers-to-functions stored in the prjprm struct.
 73 *
 74 * In summary, the routines are:
 75 *   - prjini()                Initialization routine for the prjprm struct.
 76 *   - prjfree()               Reclaim memory allocated for error messages.
 77 *   - prjsize()               Compute total size of a prjprm struct.
 78 *   - prjprt()                Print a prjprm struct.
 79 *   - prjperr()               Print error message (if any).
 80 *   - prjbchk()               Bounds checking on native coordinates.
 81 *
 82 *   - prjset(), prjx2s(), prjs2x():  Generic driver routines
 83 *
 84 *   - azpset(), azpx2s(), azps2x():  AZP (zenithal/azimuthal perspective)
 85 *   - szpset(), szpx2s(), szps2x():  SZP (slant zenithal perspective)
 86 *   - tanset(), tanx2s(), tans2x():  TAN (gnomonic)
 87 *   - stgset(), stgx2s(), stgs2x():  STG (stereographic)
 88 *   - sinset(), sinx2s(), sins2x():  SIN (orthographic/synthesis)
 89 *   - arcset(), arcx2s(), arcs2x():  ARC (zenithal/azimuthal equidistant)
 90 *   - zpnset(), zpnx2s(), zpns2x():  ZPN (zenithal/azimuthal polynomial)
 91 *   - zeaset(), zeax2s(), zeas2x():  ZEA (zenithal/azimuthal equal area)
 92 *   - airset(), airx2s(), airs2x():  AIR (Airy)
 93 *   - cypset(), cypx2s(), cyps2x():  CYP (cylindrical perspective)
 94 *   - ceaset(), ceax2s(), ceas2x():  CEA (cylindrical equal area)
 95 *   - carset(), carx2s(), cars2x():  CAR (Plate carree)
 96 *   - merset(), merx2s(), mers2x():  MER (Mercator)
 97 *   - sflset(), sflx2s(), sfls2x():  SFL (Sanson-Flamsteed)
 98 *   - parset(), parx2s(), pars2x():  PAR (parabolic)
 99 *   - molset(), molx2s(), mols2x():  MOL (Mollweide)
100 *   - aitset(), aitx2s(), aits2x():  AIT (Hammer-Aitoff)
101 *   - copset(), copx2s(), cops2x():  COP (conic perspective)
102 *   - coeset(), coex2s(), coes2x():  COE (conic equal area)
103 *   - codset(), codx2s(), cods2x():  COD (conic equidistant)
104 *   - cooset(), coox2s(), coos2x():  COO (conic orthomorphic)
105 *   - bonset(), bonx2s(), bons2x():  BON (Bonne)
106 *   - pcoset(), pcox2s(), pcos2x():  PCO (polyconic)
107 *   - tscset(), tscx2s(), tscs2x():  TSC (tangential spherical cube)
108 *   - cscset(), cscx2s(), cscs2x():  CSC (COBE spherical cube)
109 *   - qscset(), qscx2s(), qscs2x():  QSC (quadrilateralized spherical cube)
110 *   - hpxset(), hpxx2s(), hpxs2x():  HPX (HEALPix)
111 *   - xphset(), xphx2s(), xphs2x():  XPH (HEALPix polar, aka "butterfly")
112 *
```

```
113 * Argument checking (projection routines):
114 * -------------------------------------
115 * The values of phi and theta (the native longitude and latitude) normally lie
116 * in the range [-180,180] for phi, and [-90,90] for theta.  However, all
117 * projection routines will accept any value of phi and will not normalize it.
118 *
119 * The projection routines do not explicitly check that theta lies within the
120 * range [-90,90].  They do check for any value of theta that produces an
121 * invalid argument to the projection equations (e.g. leading to division by
122 * zero).  The projection routines for AZP, SZP, TAN, SIN, ZPN, and COP also
123 * return error 2 if (phi,theta) corresponds to the overlapped (far) side of
124 * the projection but also return the corresponding value of (x,y).  This
125 * strict bounds checking may be relaxed at any time by setting
126 * prjprm::bounds%2 to 0 (rather than 1); the projections need not be
127 * reinitialized.
128 *
129 * Argument checking (deprojection routines):
130 * ---------------------------------------
131 * Error checking on the projected coordinates (x,y) is limited to that
132 * required to ascertain whether a solution exists.  Where a solution does
133 * exist, an optional check is made that the value of phi and theta obtained
134 * lie within the ranges [-180,180] for phi, and [-90,90] for theta.  This
135 * check, performed by prjbchk(), is enabled by default.  It may be disabled by
136 * setting prjprm::bounds%4 to 0 (rather than 1); the projections need not be
137 * reinitialized.
138 *
139 * Accuracy:
140 * ---------
141 * No warranty is given for the accuracy of these routines (refer to the
142 * copyright notice); intending users must satisfy for themselves their
143 * adequacy for the intended purpose.  However, closure to a precision of at
144 * least 1E-10 degree of longitude and latitude has been verified for typical
145 * projection parameters on the 1 degree graticule of native longitude and
146 * latitude (to within 5 degrees of any latitude where the projection may
147 * diverge).  Refer to the tprj1.c and tprj2.c test routines that accompany
148 * this software.
149 *
150 *
151 * prjini() - Default constructor for the prjprm struct
152 * ----------------------------------------------------
153 * prjini() sets all members of a prjprm struct to default values.  It should
154 * be used to initialize every prjprm struct.
155 *
156 * PLEASE NOTE: If the prjprm struct has already been initialized, then before
157 * reinitializing, it prjfree() should be used to free any memory that may have
158 * been allocated to store an error message.  A memory leak may otherwise
159 * result.
160 *
161 * Returned:
162 *   prj       struct prjprm*
163 *                       Projection parameters.
164 *
165 * Function return value:
166 *             int       Status return value:
167 *                         0: Success.
168 *                         1: Null prjprm pointer passed.
169 *
170 *
171 * prjfree() - Destructor for the prjprm struct
172 * --------------------------------------------
173 * prjfree() frees any memory that may have been allocated to store an error
174 * message in the prjprm struct.
175 *
176 * Given:
177 *   prj       struct prjprm*
178 *                       Projection parameters.
179 *
180 * Function return value:
181 *             int       Status return value:
182 *                         0: Success.
183 *                         1: Null prjprm pointer passed.
184 *
185 *
186 * prjsize() - Compute the size of a prjprm struct
187 * -----------------------------------------------
188 * prjsize() computes the full size of a prjprm struct, including allocated
189 * memory.
190 *
191 * Given:
192 *   prj       const struct prjprm*
193 *                       Projection parameters.
194 *
195 *                       If NULL, the base size of the struct and the allocated
196 *                       size are both set to zero.
197 *
198 * Returned:
199 *   sizes     int[2]    The first element is the base size of the struct as
```

```
200 *                          returned by sizeof(struct prjprm).  The second element
201 *                          is the total allocated size, in bytes.  This figure
202 *                          includes memory allocated for the constituent struct,
203 *                          prjprm::err.
204 *
205 *                          It is not an error for the struct not to have been set
206 *                          up via prjset().
207 *
208 * Function return value:
209 *             int       Status return value:
210 *                          0: Success.
211 *
212 *
213 * prjprt() - Print routine for the prjprm struct
214 * ----------------------------------------------
215 * prjprt() prints the contents of a prjprm struct using wcsprintf().  Mainly
216 * intended for diagnostic purposes.
217 *
218 * Given:
219 *   prj       const struct prjprm*
220 *                          Projection parameters.
221 *
222 * Function return value:
223 *             int       Status return value:
224 *                          0: Success.
225 *                          1: Null prjprm pointer passed.
226 *
227 *
228 * prjperr() - Print error messages from a prjprm struct
229 * -----------------------------------------------------
230 * prjperr() prints the error message(s) (if any) stored in a prjprm struct.
231 * If there are no errors then nothing is printed.  It uses wcserr_prt(), q.v.
232 *
233 * Given:
234 *   prj       const struct prjprm*
235 *                          Projection parameters.
236 *
237 *   prefix    const char *
238 *                          If non-NULL, each output line will be prefixed with
239 *                          this string.
240 *
241 * Function return value:
242 *             int       Status return value:
243 *                          0: Success.
244 *                          1: Null prjprm pointer passed.
245 *
246 *
247 * prjbchk() - Bounds checking on native coordinates
248 * -------------------------------------------------
249 * prjbchk() performs bounds checking on native spherical coordinates.  As
250 * returned by the deprojection (x2s) routines, native longitude is expected
251 * to lie in the closed interval [-180,180], with latitude in [-90,90].
252 *
253 * A tolerance may be specified to provide a small allowance for numerical
254 * imprecision.  Values that lie outside the allowed range by not more than
255 * the specified tolerance will be adjusted back into range.
256 *
257 * If prjprm::bounds&4 is set, as it is by prjini(), then prjbchk() will be
258 * invoked automatically by the Cartesian-to-spherical deprojection (x2s)
259 * routines with an appropriate tolerance set for each projection.
260 *
261 * Given:
262 *   tol       double    Tolerance for the bounds check [deg].
263 *
264 *   nphi,
265 *   ntheta    int       Vector lengths.
266 *
267 *   spt       int       Vector stride.
268 *
269 * Given and returned:
270 *   phi,theta double[]  Native longitude and latitude (phi,theta) [deg].
271 *
272 * Returned:
273 *   stat      int[]     Status value for each vector element:
274 *                          0: Valid value of (phi,theta).
275 *                          1: Invalid value.
276 *
277 * Function return value:
278 *             int       Status return value:
279 *                          0: Success.
280 *                          1: One or more of the (phi,theta) coordinates
281 *                             were, invalid, as indicated by the stat vector.
282 *
283 *
284 * prjset() - Generic setup routine for the prjprm struct
285 * ------------------------------------------------------
286 * prjset() sets up a prjprm struct according to information supplied within
```

```
287 * it.
288 *
289 * Note that this routine need not be called directly; it will be invoked by
290 * prjx2s() and prjs2x() if prj.flag is anything other than a predefined magic
291 * value.
292 *
293 * The one important distinction between prjset() and the setup routines for
294 * the specific projections is that the projection code must be defined in the
295 * prjprm struct in order for prjset() to identify the required projection.
296 * Once prjset() has initialized the prjprm struct, prjx2s() and prjs2x() use
297 * the pointers to the specific projection and deprojection routines contained
298 * therein.
299 *
300 * Given and returned:
301 *   prj       struct prjprm*
302 *                       Projection parameters.
303 *
304 * Function return value:
305 *             int       Status return value:
306 *                         0: Success.
307 *                         1: Null prjprm pointer passed.
308 *                         2: Invalid projection parameters.
309 *
310 *                       For returns > 1, a detailed error message is set in
311 *                       prjprm::err if enabled, see wcserr_enable().
312 *
313 *
314 * prjx2s() - Generic Cartesian-to-spherical deprojection
315 * ------------------------------------------------------
316 * Deproject Cartesian (x,y) coordinates in the plane of projection to native
317 * spherical coordinates (phi,theta).
318 *
319 * The projection is that specified by prjprm::code.
320 *
321 * Given and returned:
322 *   prj       struct prjprm*
323 *                       Projection parameters.
324 *
325 * Given:
326 *   nx,ny     int       Vector lengths.
327 *
328 *   sxy,spt   int       Vector strides.
329 *
330 *   x,y       const double[]
331 *                       Projected coordinates.
332 *
333 * Returned:
334 *   phi,theta double[]  Longitude and latitude (phi,theta) of the projected
335 *                       point in native spherical coordinates [deg].
336 *
337 *   stat      int[]     Status value for each vector element:
338 *                         0: Success.
339 *                         1: Invalid value of (x,y).
340 *
341 * Function return value:
342 *             int       Status return value:
343 *                         0: Success.
344 *                         1: Null prjprm pointer passed.
345 *                         2: Invalid projection parameters.
346 *                         3: One or more of the (x,y) coordinates were
347 *                            invalid, as indicated by the stat vector.
348 *
349 *                       For returns > 1, a detailed error message is set in
350 *                       prjprm::err if enabled, see wcserr_enable().
351 *
352 *
353 * prjs2x() - Generic spherical-to-Cartesian projection
354 * ----------------------------------------------------
355 * Project native spherical coordinates (phi,theta) to Cartesian (x,y)
356 * coordinates in the plane of projection.
357 *
358 * The projection is that specified by prjprm::code.
359 *
360 * Given and returned:
361 *   prj       struct prjprm*
362 *                       Projection parameters.
363 *
364 * Given:
365 *   nphi,
366 *   ntheta    int       Vector lengths.
367 *
368 *   spt,sxy   int       Vector strides.
369 *
370 *   phi,theta const double[]
371 *                       Longitude and latitude (phi,theta) of the projected
372 *                       point in native spherical coordinates [deg].
373 *
```

```
374 * Returned:
375 *   x,y        double[]  Projected coordinates.
376 *
377 *   stat       int[]     Status value for each vector element:
378 *                          0: Success.
379 *                          1: Invalid value of (phi,theta).
380 *
381 * Function return value:
382 *              int       Status return value:
383 *                          0: Success.
384 *                          1: Null prjprm pointer passed.
385 *                          2: Invalid projection parameters.
386 *                          4: One or more of the (phi,theta) coordinates
387 *                             were, invalid, as indicated by the stat vector.
388 *
389 *                        For returns > 1, a detailed error message is set in
390 *                        prjprm::err if enabled, see wcserr_enable().
391 *
392 *
393 * ???set() - Specific setup routines for the prjprm struct
394 * --------------------------------------------------------
395 * Set up a prjprm struct for a particular projection according to information
396 * supplied within it.
397 *
398 * Given and returned:
399 *   prj        struct prjprm*
400 *                        Projection parameters.
401 *
402 * Function return value:
403 *              int       Status return value:
404 *                          0: Success.
405 *                          1: Null prjprm pointer passed.
406 *                          2: Invalid projection parameters.
407 *
408 *                        For returns > 1, a detailed error message is set in
409 *                        prjprm::err if enabled, see wcserr_enable().
410 *
411 *
412 * ???x2s() - Specific Cartesian-to-spherical deprojection routines
413 * ----------------------------------------------------------------
414 * Transform (x,y) coordinates in the plane of projection to native spherical
415 * coordinates (phi,theta).
416 *
417 * Given and returned:
418 *   prj        struct prjprm*
419 *                        Projection parameters.
420 *
421 * Given:
422 *   nx,ny      int       Vector lengths.
423 *
424 *   sxy,spt    int       Vector strides.
425 *
426 *   x,y        const double[]
427 *                        Projected coordinates.
428 *
429 * Returned:
430 *   phi,theta  double[]  Longitude and latitude of the projected point in
431 *                        native spherical coordinates [deg].
432 *
433 *   stat       int[]     Status value for each vector element:
434 *                          0: Success.
435 *                          1: Invalid value of (x,y).
436 *
437 * Function return value:
438 *              int       Status return value:
439 *                          0: Success.
440 *                          1: Null prjprm pointer passed.
441 *                          2: Invalid projection parameters.
442 *                          3: One or more of the (x,y) coordinates were
443 *                             invalid, as indicated by the stat vector.
444 *
445 *                        For returns > 1, a detailed error message is set in
446 *                        prjprm::err if enabled, see wcserr_enable().
447 *
448 *
449 * ???s2x() - Specific spherical-to-Cartesian projection routines
450 *---------------------------------------------------------------
451 * Transform native spherical coordinates (phi,theta) to (x,y) coordinates in
452 * the plane of projection.
453 *
454 * Given and returned:
455 *   prj        struct prjprm*
456 *                        Projection parameters.
457 *
458 * Given:
459 *   nphi,
460 *   ntheta     int       Vector lengths.
```

```
461 *
462 *   spt,sxy   int       Vector strides.
463 *
464 *   phi,theta const double[]
465 *                       Longitude and latitude of the projected point in
466 *                       native spherical coordinates [deg].
467 *
468 * Returned:
469 *   x,y       double[]  Projected coordinates.
470 *
471 *   stat      int[]     Status value for each vector element:
472 *                         0: Success.
473 *                         1: Invalid value of (phi,theta).
474 *
475 * Function return value:
476 *             int       Status return value:
477 *                         0: Success.
478 *                         1: Null prjprm pointer passed.
479 *                         2: Invalid projection parameters.
480 *                         4: One or more of the (phi,theta) coordinates
481 *                            were, invalid, as indicated by the stat vector.
482 *
483 *                       For returns > 1, a detailed error message is set in
484 *                       prjprm::err if enabled, see wcserr_enable().
485 *
486 *
487 * prjprm struct - Projection parameters
488 * -------------------------------------
489 * The prjprm struct contains all information needed to project or deproject
490 * native spherical coordinates.  It consists of certain members that must be
491 * set by the user ("given") and others that are set by the WCSLIB routines
492 * ("returned").  Some of the latter are supplied for informational purposes
493 * while others are for internal use only.
494 *
495 *   int flag
496 *     (Given and returned) This flag must be set to zero whenever any of the
497 *     following prjprm struct members are set or changed:
498 *
499 *       - prjprm::code,
500 *       - prjprm::r0,
501 *       - prjprm::pv[],
502 *       - prjprm::phi0,
503 *       - prjprm::theta0.
504 *
505 *     This signals the initialization routine (prjset() or ???set()) to
506 *     recompute the returned members of the prjprm struct.  flag will then be
507 *     reset to indicate that this has been done.
508 *
509 *     Note that flag need not be reset when prjprm::bounds is changed.
510 *
511 *   char code[4]
512 *     (Given) Three-letter projection code defined by the FITS standard.
513 *
514 *   double r0
515 *     (Given) The radius of the generating sphere for the projection, a linear
516 *     scaling parameter.  If this is zero, it will be reset to its default
517 *     value of 180/pi (the value for FITS WCS).
518 *
519 *   double pv[30]
520 *     (Given) Projection parameters.  These correspond to the PVi_ma keywords
521 *     in FITS, so pv[0] is PVi_0a, pv[1] is PVi_1a, etc., where i denotes the
522 *     latitude-like axis.  Many projections use pv[1] (PVi_1a), some also use
523 *     pv[2] (PVi_2a) and SZP uses pv[3] (PVi_3a).  ZPN is currently the only
524 *     projection that uses any of the others.
525 *
526 *     Usage of the pv[] array as it applies to each projection is described in
527 *     the prologue to each trio of projection routines in prj.c.
528 *
529 *   double phi0
530 *     (Given) The native longitude, phi_0 [deg], and ...
531 *   double theta0
532 *     (Given) ... the native latitude, theta_0 [deg], of the reference point,
533 *     i.e. the point (x,y) = (0,0).  If undefined (set to a magic value by
534 *     prjini()) the initialization routine will set this to a
535 *     projection-specific default.
536 *
537 *   int bounds
538 *     (Given) Controls bounds checking.  If bounds&1 then enable strict bounds
539 *     checking for the spherical-to-Cartesian (s2x) transformation for the
540 *     AZP, SZP, TAN, SIN, ZPN, and COP projections.  If bounds&2 then enable
541 *     strict bounds checking for the Cartesian-to-spherical transformation
542 *     (x2s) for the HPX and XPH projections.  If bounds&4 then the Cartesian-
543 *     to-spherical transformations (x2s) will invoke prjbchk() to perform
544 *     bounds checking on the computed native coordinates, with a tolerance set
545 *     to suit each projection.  bounds is set to 7 by prjini() by default
546 *     which enables all checks.  Zero it to disable all checking.
547 *
```

```
548 *      It is not necessary to reset the prjprm struct (via prjset() or
549 *      ???set()) when prjprm::bounds is changed.
550 *
551 * The remaining members of the prjprm struct are maintained by the setup
552 * routines and must not be modified elsewhere:
553 *
554 *   char name[40]
555 *      (Returned) Long name of the projection.
556 *
557 *      Provided for information only, not used by the projection routines.
558 *
559 *   int   category
560 *      (Returned) Projection category matching the value of the relevant global
561 *      variable:
562 *
563 *      - ZENITHAL,
564 *      - CYLINDRICAL,
565 *      - PSEUDOCYLINDRICAL,
566 *      - CONVENTIONAL,
567 *      - CONIC,
568 *      - POLYCONIC,
569 *      - QUADCUBE, and
570 *      - HEALPIX.
571 *
572 *      The category name may be identified via the prj_categories character
573 *      array, e.g.
574 *
575 =        struct prjprm prj;
576 =           ...
577 =        printf("%s\n", prj_categories[prj.category]);
578 *
579 *      Provided for information only, not used by the projection routines.
580 *
581 *   int   pvrange
582 *      (Returned) Range of projection parameter indices: 100 times the first
583 *      allowed index plus the number of parameters, e.g. TAN is 0 (no
584 *      parameters), SZP is 103 (1 to 3), and ZPN is 30 (0 to 29).
585 *
586 *      Provided for information only, not used by the projection routines.
587 *
588 *   int   simplezen
589 *      (Returned) True if the projection is a radially-symmetric zenithal
590 *      projection.
591 *
592 *      Provided for information only, not used by the projection routines.
593 *
594 *   int   equiareal
595 *      (Returned) True if the projection is equal area.
596 *
597 *      Provided for information only, not used by the projection routines.
598 *
599 *   int   conformal
600 *      (Returned) True if the projection is conformal.
601 *
602 *      Provided for information only, not used by the projection routines.
603 *
604 *   int   global
605 *      (Returned) True if the projection can represent the whole sphere in a
606 *      finite, non-overlapped mapping.
607 *
608 *      Provided for information only, not used by the projection routines.
609 *
610 *   int   divergent
611 *      (Returned) True if the projection diverges in latitude.
612 *
613 *      Provided for information only, not used by the projection routines.
614 *
615 *   double x0
616 *      (Returned) The offset in x, and ...
617 *   double y0
618 *      (Returned) ... the offset in y used to force (x,y) = (0,0) at
619 *      (phi_0,theta_0).
620 *
621 *   struct wcserr *err
622 *      (Returned) If enabled, when an error status is returned, this struct
623 *      contains detailed information about the error, see wcserr_enable().
624 *
625 *   void *padding
626 *      (An unused variable inserted for alignment purposes only.)
627 *
628 *   double w[10]
629 *      (Returned) Intermediate floating-point values derived from the
630 *      projection parameters, cached here to save recomputation.
631 *
632 *      Usage of the w[] array as it applies to each projection is described in
633 *      the prologue to each trio of projection routines in prj.c.
634 *
```

```
635 *    int n
636 *      (Returned) Intermediate integer value (used only for the ZPN and HPX
637 *      projections).
638 *
639 *    int (*prjx2s)(PRJX2S_ARGS)
640 *      (Returned) Pointer to the spherical projection ...
641 *    int (*prjs2x)(PRJ_ARGS)
642 *      (Returned) ... and deprojection routines.
643 *
644 *
645 * Global variable: const char *prj_errmsg[] - Status return messages
646 * -------------------------------------------------------------------
647 * Error messages to match the status value returned from each function.
648 *
649 *===========================================================================*/
650
651 #ifndef WCSLIB_PROJ
652 #define WCSLIB_PROJ
653
654 #ifdef __cplusplus
655 extern "C" {
656 #endif
657
658
659 // Total number of projection parameters; 0 to PVN-1.
660 #define PVN 30
661
662 extern const char *prj_errmsg[];
663
664 enum prj_errmsg_enum {
665   PRJERR_SUCCESS      = 0,       // Success.
666   PRJERR_NULL_POINTER = 1,    // Null prjprm pointer passed.
667   PRJERR_BAD_PARAM    = 2,      // Invalid projection parameters.
668   PRJERR_BAD_PIX      = 3,        // One or more of the (x, y) coordinates were
669                                  // invalid.
670   PRJERR_BAD_WORLD    = 4        // One or more of the (phi, theta) coordinates
671                                  // were invalid.
672 };
673
674 extern const int CONIC, CONVENTIONAL, CYLINDRICAL, POLYCONIC,
675                  PSEUDOCYLINDRICAL, QUADCUBE, ZENITHAL, HEALPIX;
676 extern const char prj_categories[9][32];
677
678 extern const int  prj_ncode;
679 extern const char prj_codes[28][4];
680
681 #ifdef PRJX2S_ARGS
682 #undef PRJX2S_ARGS
683 #endif
684
685 #ifdef PRJS2X_ARGS
686 #undef PRJS2X_ARGS
687 #endif
688
689 // For use in declaring deprojection function prototypes.
690 #define PRJX2S_ARGS struct prjprm *prj, int nx, int ny, int sxy, int spt, \
691 const double x[], const double y[], double phi[], double theta[], int stat[]
692
693 // For use in declaring projection function prototypes.
694 #define PRJS2X_ARGS struct prjprm *prj, int nx, int ny, int sxy, int spt, \
695 const double phi[], const double theta[], double x[], double y[], int stat[]
696
697
698 struct prjprm {
699   // Initialization flag (see the prologue above).
700   //------------------------------------------------------------------------
701   int    flag;                    // Set to zero to force initialization.
702
703   // Parameters to be provided (see the prologue above).
704   //------------------------------------------------------------------------
705   char   code[4];           // Three-letter projection code.
706   double r0;                   // Radius of the generating sphere.
707   double pv[PVN];           // Projection parameters.
708   double phi0, theta0;             // Fiducial native coordinates.
709   int    bounds;           // Controls bounds checking.
710
711   // Information derived from the parameters supplied.
712   //------------------------------------------------------------------------
713   char   name[40];          // Projection name.
714   int    category;             // Projection category.
715   int    pvrange;              // Range of projection parameter indices.
716   int    simplezen;          // Is it a simple zenithal projection?
717   int    equiareal;          // Is it an equal area projection?
718   int    conformal;          // Is it a conformal projection?
719   int    global;           // Can it map the whole sphere?
720   int    divergent;            // Does the projection diverge in latitude?
721   double x0, y0;             // Fiducial offsets.
```

```
722
723   // Error handling
724   //----------------------------------------------------------------------
725   struct wcserr *err;
726
727   // Private
728   //----------------------------------------------------------------------
729   void   *padding;                  // (Dummy inserted for alignment purposes.)
730   double w[10];                         // Intermediate values.
731   int    m, n;                          // Intermediate values.
732
733   int (*prjx2s)(PRJX2S_ARGS);  // Pointers to the spherical projection and
734   int (*prjs2x)(PRJS2X_ARGS);  // deprojection functions.
735 };
736
737 // Size of the prjprm struct in int units, used by the Fortran wrappers.
738 #define PRJLEN (sizeof(struct prjprm)/sizeof(int))
739
740
741 int prjini(struct prjprm *prj);
742
743 int prjfree(struct prjprm *prj);
744
745 int prjsize(const struct prjprm *prj, int sizes[2]);
746
747 int prjprt(const struct prjprm *prj);
748
749 int prjperr(const struct prjprm *prj, const char *prefix);
750
751 int prjbchk(double tol, int nphi, int ntheta, int spt, double phi[],
752             double theta[], int stat[]);
753
754 // Use the preprocessor to help declare function prototypes (see above).
755 int prjset(struct prjprm *prj);
756 int prjx2s(PRJX2S_ARGS);
757 int prjs2x(PRJS2X_ARGS);
758
759 int azpset(struct prjprm *prj);
760 int azpx2s(PRJX2S_ARGS);
761 int azps2x(PRJS2X_ARGS);
762
763 int szpset(struct prjprm *prj);
764 int szpx2s(PRJX2S_ARGS);
765 int szps2x(PRJS2X_ARGS);
766
767 int tanset(struct prjprm *prj);
768 int tanx2s(PRJX2S_ARGS);
769 int tans2x(PRJS2X_ARGS);
770
771 int stgset(struct prjprm *prj);
772 int stgx2s(PRJX2S_ARGS);
773 int stgs2x(PRJS2X_ARGS);
774
775 int sinset(struct prjprm *prj);
776 int sinx2s(PRJX2S_ARGS);
777 int sins2x(PRJS2X_ARGS);
778
779 int arcset(struct prjprm *prj);
780 int arcx2s(PRJX2S_ARGS);
781 int arcs2x(PRJS2X_ARGS);
782
783 int zpnset(struct prjprm *prj);
784 int zpnx2s(PRJX2S_ARGS);
785 int zpns2x(PRJS2X_ARGS);
786
787 int zeaset(struct prjprm *prj);
788 int zeax2s(PRJX2S_ARGS);
789 int zeas2x(PRJS2X_ARGS);
790
791 int airset(struct prjprm *prj);
792 int airx2s(PRJX2S_ARGS);
793 int airs2x(PRJS2X_ARGS);
794
795 int cypset(struct prjprm *prj);
796 int cypx2s(PRJX2S_ARGS);
797 int cyps2x(PRJS2X_ARGS);
798
799 int ceaset(struct prjprm *prj);
800 int ceax2s(PRJX2S_ARGS);
801 int ceas2x(PRJS2X_ARGS);
802
803 int carset(struct prjprm *prj);
804 int carx2s(PRJX2S_ARGS);
805 int cars2x(PRJS2X_ARGS);
806
807 int merset(struct prjprm *prj);
808 int merx2s(PRJX2S_ARGS);
```

```
809 int mers2x(PRJS2X_ARGS);
810
811 int sflset(struct prjprm *prj);
812 int sflx2s(PRJX2S_ARGS);
813 int sfls2x(PRJS2X_ARGS);
814
815 int parset(struct prjprm *prj);
816 int parx2s(PRJX2S_ARGS);
817 int pars2x(PRJS2X_ARGS);
818
819 int molset(struct prjprm *prj);
820 int molx2s(PRJX2S_ARGS);
821 int mols2x(PRJS2X_ARGS);
822
823 int aitset(struct prjprm *prj);
824 int aitx2s(PRJX2S_ARGS);
825 int aits2x(PRJS2X_ARGS);
826
827 int copset(struct prjprm *prj);
828 int copx2s(PRJX2S_ARGS);
829 int cops2x(PRJS2X_ARGS);
830
831 int coeset(struct prjprm *prj);
832 int coex2s(PRJX2S_ARGS);
833 int coes2x(PRJS2X_ARGS);
834
835 int codset(struct prjprm *prj);
836 int codx2s(PRJX2S_ARGS);
837 int cods2x(PRJS2X_ARGS);
838
839 int cooset(struct prjprm *prj);
840 int coox2s(PRJX2S_ARGS);
841 int coos2x(PRJS2X_ARGS);
842
843 int bonset(struct prjprm *prj);
844 int bonx2s(PRJX2S_ARGS);
845 int bons2x(PRJX2S_ARGS);
846
847 int pcoset(struct prjprm *prj);
848 int pcox2s(PRJX2S_ARGS);
849 int pcos2x(PRJX2S_ARGS);
850
851 int tscset(struct prjprm *prj);
852 int tscx2s(PRJX2S_ARGS);
853 int tscs2x(PRJS2X_ARGS);
854
855 int cscset(struct prjprm *prj);
856 int cscx2s(PRJX2S_ARGS);
857 int cscs2x(PRJS2X_ARGS);
858
859 int qscset(struct prjprm *prj);
860 int qscx2s(PRJX2S_ARGS);
861 int qscs2x(PRJS2X_ARGS);
862
863 int hpxset(struct prjprm *prj);
864 int hpxx2s(PRJX2S_ARGS);
865 int hpxs2x(PRJS2X_ARGS);
866
867 int xphset(struct prjprm *prj);
868 int xphx2s(PRJX2S_ARGS);
869 int xphs2x(PRJS2X_ARGS);
870
871
872 // Deprecated.
873 #define prjini_errmsg prj_errmsg
874 #define prjprt_errmsg prj_errmsg
875 #define prjset_errmsg prj_errmsg
876 #define prjx2s_errmsg prj_errmsg
877 #define prjs2x_errmsg prj_errmsg
878
879 #ifdef __cplusplus
880 }
881 #endif
882
883 #endif // WCSLIB_PROJ
```

19.15 spc.h File Reference

```
#include "spx.h"
```

Data Structures

- struct spcprm

    Spectral transformation parameters.

Macros

- #define SPCLEN (sizeof(struct spcprm)/sizeof(int))

    Size of the spcprm struct in int units.
- #define spcini_errmsg spc_errmsg

    Deprecated.
- #define spcprt_errmsg spc_errmsg

    Deprecated.
- #define spcset_errmsg spc_errmsg

    Deprecated.
- #define spcx2s_errmsg spc_errmsg

    Deprecated.
- #define spcs2x_errmsg spc_errmsg

    Deprecated.

Enumerations

- enum spc_errmsg_enum {
  SPCERR_NO_CHANGE = -1 , SPCERR_SUCCESS = 0 , SPCERR_NULL_POINTER = 1 , SPCERR_BAD_SPEC_PARAMS = 2 ,
  SPCERR_BAD_X = 3 , SPCERR_BAD_SPEC = 4 }

Functions

- int spcini (struct spcprm ∗spc)

    Default constructor for the spcprm struct.
- int spcfree (struct spcprm ∗spc)

    Destructor for the spcprm struct.
- int spcsize (const struct spcprm ∗spc, int sizes[2])

    Compute the size of a spcprm struct.
- int spcprt (const struct spcprm ∗spc)

    Print routine for the spcprm struct.
- int spcperr (const struct spcprm ∗spc, const char ∗prefix)

    Print error messages from a spcprm struct.
- int spcset (struct spcprm ∗spc)

    Setup routine for the spcprm struct.
- int spcx2s (struct spcprm ∗spc, int nx, int sx, int sspec, const double x[ ], double spec[ ], int stat[ ])

    Transform to spectral coordinates.
- int spcs2x (struct spcprm ∗spc, int nspec, int sspec, int sx, const double spec[ ], double x[ ], int stat[ ])

    Transform spectral coordinates.
- int spctype (const char ctype[9], char stype[ ], char scode[ ], char sname[ ], char units[ ], char ∗ptype, char ∗xtype, int ∗restreq, struct wcserr ∗∗err)

    *Spectral **CTYPE**ia keyword analysis.*
- int spcspxe (const char ctypeS[9], double crvalS, double restfrq, double restwav, char ∗ptype, char ∗xtype, int ∗restreq, double ∗crvalX, double ∗dXdS, struct wcserr ∗∗err)

Spectral keyword analysis.

- int spcxpse (const char ctypeS[9], double crvalX, double restfrq, double restwav, char ∗ptype, char ∗xtype, int ∗restreq, double ∗crvalS, double ∗dSdX, struct wcserr ∗∗err)

    Spectral keyword synthesis.

- int spctrne (const char ctypeS1[9], double crvalS1, double cdeltS1, double restfrq, double restwav, char ctypeS2[9], double ∗crvalS2, double ∗cdeltS2, struct wcserr ∗∗err)

    Spectral keyword translation.

- int spcaips (const char ctypeA[9], int velref, char ctype[9], char specsys[9])

    Translate AIPS-convention spectral keywords.

- int spctyp (const char ctype[9], char stype[ ], char scode[ ], char sname[ ], char units[ ], char ∗ptype, char ∗xtype, int ∗restreq)

- int spcspx (const char ctypeS[9], double crvalS, double restfrq, double restwav, char ∗ptype, char ∗xtype, int ∗restreq, double ∗crvalX, double ∗dXdS)

- int spcxps (const char ctypeS[9], double crvalX, double restfrq, double restwav, char ∗ptype, char ∗xtype, int ∗restreq, double ∗crvalS, double ∗dSdX)

- int spctrn (const char ctypeS1[9], double crvalS1, double cdeltS1, double restfrq, double restwav, char ctype↩S2[9], double ∗crvalS2, double ∗cdeltS2)

Variables

- const char ∗ spc_errmsg [ ]

    Status return messages.

19.15.1   Detailed Description

Routines in this suite implement the part of the FITS World Coordinate System (WCS) standard that deals with spectral coordinates, as described in

```
"Representations of world coordinates in FITS",
Greisen, E.W., & Calabretta, M.R. 2002, A&A, 395, 1061 (WCS Paper I)
"Representations of spectral coordinates in FITS",
Greisen, E.W., Calabretta, M.R., Valdes, F.G., & Allen, S.L.
2006, A&A, 446, 747 (WCS Paper III)
```

These routines define methods to be used for computing spectral world coordinates from intermediate world coordinates (a linear transformation of image pixel coordinates), and vice versa. They are based on the spcprm struct which contains all information needed for the computations. The struct contains some members that must be set by the user, and others that are maintained by these routines, somewhat like a C++ class but with no encapsulation.

Routine spcini() is provided to initialize the spcprm struct with default values, spcfree() reclaims any memory that may have been allocated to store an error message, spcsize() computes its total size including allocated memory, and spcprt() prints its contents.

spcperr() prints the error message(s) (if any) stored in a spcprm struct.

A setup routine, spcset(), computes intermediate values in the spcprm struct from parameters in it that were supplied by the user. The struct always needs to be set up by spcset() but it need not be called explicitly - refer to the explanation of spcprm::flag.

spcx2s() and spcs2x() implement the WCS spectral coordinate transformations. In fact, they are high level driver routines for the lower level spectral coordinate transformation routines described in spx.h.

A number of routines are provided to aid in analysing or synthesising sets of FITS spectral axis keywords:

- spctype() checks a spectral **CTYPE**ia keyword for validity and returns information derived from it.

- Spectral keyword analysis routine [spcspxe()](#) computes the values of the X-type spectral variables for the S-type variables supplied.

- Spectral keyword synthesis routine, [spcxpse()](#), computes the S-type variables for the X-types supplied.

- Given a set of spectral keywords, a translation routine, [spctrne()](#), produces the corresponding set for the specified spectral **CTYPE**ia.

- [spcaips()](#) translates AIPS-convention spectral **CTYPE**ia and **VELREF** keyvalues.

Spectral variable types - S, P, and X:
A few words of explanation are necessary regarding spectral variable types in FITS.

Every FITS spectral axis has three associated spectral variables:

S-type: the spectral variable in which coordinates are to be expressed. Each S-type is encoded as four characters and is linearly related to one of four basic types as follows:

F (Frequency):

- '**FREQ**': frequency
- '**AFRQ**': angular frequency
- '**ENER**': photon energy
- '**WAVN**': wave number
- '**VRAD**': radio velocity

W (Wavelength in vacuo):

- '**WAVE**': wavelength
- '**VOPT**': optical velocity
- '**ZOPT**': redshift

A (wavelength in Air):

- '**AWAV**': wavelength in air

V (Velocity):

- '**VELO**': relativistic velocity
- '**BETA**': relativistic beta factor

The S-type forms the first four characters of the **CTYPE**ia keyvalue, and **CRVAL**ia and **CDELT**ia are expressed as S-type quantities so that they provide a first-order approximation to the S-type variable at the reference point.

Note that '**AFRQ**', angular frequency, is additional to the variables defined in WCS Paper III.

P-type: the basic spectral variable (F, W, A, or V) with which the S-type variable is associated (see list above).

For non-grism axes, the P-type is encoded as the eighth character of **CTYPE**ia.

X-type: the basic spectral variable (F, W, A, or V) for which the spectral axis is linear, grisms excluded (see below).

For non-grism axes, the X-type is encoded as the sixth character of **CTYPE**ia.

Grisms: Grism axes have normal S-, and P-types but the axis is linear, not in any spectral variable, but in a special "grism parameter". The X-type spectral variable is either W or A for grisms in vacuo or air respectively, but is encoded as 'w' or 'a' to indicate that an additional transformation is required to convert to or from the grism parameter. The spectral algorithm code for grisms also has a special encoding in **CTYPE**ia, either '**GRI**' (in vacuo) or '**GRA**' (in air).

In the algorithm chain, the non-linear transformation occurs between the X-type and the P-type variables; the transformation between P-type and S-type variables is always linear.

When the P-type and X-type variables are the same, the spectral axis is linear in the S-type variable and the second four characters of **CTYPE**ia are blank. This can never happen for grism axes.

As an example, correlating radio spectrometers always produce spectra that are regularly gridded in frequency; a redshift scale on such a spectrum is non-linear. The required value of **CTYPE**ia would be **'ZOPT-F2W'**, where the desired S-type is **'ZOPT'** (redshift), the P-type is necessarily 'W' (wavelength), and the X-type is 'F' (frequency) by the nature of the instrument.

Air-to-vacuum wavelength conversion:
Please refer to the prologue of spx.h for important comments relating to the air-to-vacuum wavelength conversion.

Argument checking:
The input spectral values are only checked for values that would result in floating point exceptions. In particular, negative frequencies and wavelengths are allowed, as are velocities greater than the speed of light. The same is true for the spectral parameters - rest frequency and wavelength.

Accuracy:
No warranty is given for the accuracy of these routines (refer to the copyright notice); intending users must satisfy for themselves their adequacy for the intended purpose. However, closure effectively to within double precision rounding error was demonstrated by test routine tspc.c which accompanies this software.

19.15.2   Macro Definition Documentation

19.15.2.1   SPCLEN   #define SPCLEN (sizeof(struct spcprm)/sizeof(int))

Size of the spcprm struct in *int* units, used by the Fortran wrappers.

19.15.2.2   spcini_errmsg   #define spcini_errmsg spc_errmsg

Deprecated   Added for backwards compatibility, use spc_errmsg directly now instead.

19.15.2.3 spcprt_errmsg #define spcprt_errmsg spc_errmsg

Deprecated Added for backwards compatibility, use spc_errmsg directly now instead.

19.15.2.4 spcset_errmsg #define spcset_errmsg spc_errmsg

Deprecated Added for backwards compatibility, use spc_errmsg directly now instead.

19.15.2.5 spcx2s_errmsg #define spcx2s_errmsg spc_errmsg

Deprecated Added for backwards compatibility, use spc_errmsg directly now instead.

19.15.2.6 spcs2x_errmsg #define spcs2x_errmsg spc_errmsg

Deprecated Added for backwards compatibility, use spc_errmsg directly now instead.

19.15.3 Enumeration Type Documentation

19.15.3.1 spc_errmsg_enum enum spc_errmsg_enum

Enumerator

SPCERR_NO_CHANGE	
SPCERR_SUCCESS	
SPCERR_NULL_POINTER	
SPCERR_BAD_SPEC_PARAMS	
SPCERR_BAD_X	
SPCERR_BAD_SPEC	

19.15.4 Function Documentation

19.15.4.1   spcini() `int spcini (`
            `struct spcprm * spc )`

spcini() sets all members of a spcprm struct to default values. It should be used to initialize every spcprm struct.

PLEASE NOTE: If the spcprm struct has already been initialized, then before reinitializing, it spcfree() should be used to free any memory that may have been allocated to store an error message. A memory leak may otherwise result.

Parameters

in,out	*spc*	Spectral transformation parameters.

Returns

Status return value:

- 0: Success.
- 1: Null spcprm pointer passed.

19.15.4.2   spcfree() `int spcfree (`
            `struct spcprm * spc )`

spcfree() frees any memory that may have been allocated to store an error message in the spcprm struct.

Parameters

in	*spc*	Spectral transformation parameters.

Returns

Status return value:

- 0: Success.
- 1: Null spcprm pointer passed.

19.15.4.3   spcsize() `int spcsize (`
            `const struct spcprm * spc,`
            `int sizes[2] )`

spcsize() computes the full size of a spcprm struct, including allocated memory.

Parameters

in	*spc*	Spectral transformation parameters.
		If NULL, the base size of the struct and the allocated size are both set to zero.
out	*sizes*	The first element is the base size of the struct as returned by sizeof(struct spcprm). The second element is the total allocated size, in bytes. This figure includes memory allocated for the constituent struct, spcprm::err.
Generated by Doxygen		It is not an error for the struct not to have been set up via spcset().

Returns

Status return value:

- 0: Success.

19.15.4.4 spcprt() `int spcprt (`
`const struct` spcprm `*` *spc* `)`

spcprt() prints the contents of a spcprm struct using wcsprintf(). Mainly intended for diagnostic purposes.

Parameters

in	*spc*	Spectral transformation parameters.

Returns

Status return value:

- 0: Success.
- 1: Null spcprm pointer passed.

19.15.4.5 spcperr() `int spcperr (`
`const struct` spcprm `*` *spc,*
`const char` `*` *prefix* `)`

spcperr() prints the error message(s) (if any) stored in a spcprm struct. If there are no errors then nothing is printed. It uses wcserr_prt(), q.v.

Parameters

in	*spc*	Spectral transformation parameters.
in	*prefix*	If non-NULL, each output line will be prefixed with this string.

Returns

Status return value:

- 0: Success.
- 1: Null spcprm pointer passed.

19.15.4.6 spcset() `int spcset (`
`struct` spcprm `*` *spc* `)`

spcset() sets up a spcprm struct according to information supplied within it.

Note that this routine need not be called directly; it will be invoked by spcx2s() and spcs2x() if spcprm::flag is anything other than a predefined magic value.

Parameters

in,out	*spc*	Spectral transformation parameters.

Returns

Status return value:

- 0: Success.

- 1: Null spcprm pointer passed.

- 2: Invalid spectral parameters.

For returns > 1, a detailed error message is set in spcprm::err if enabled, see wcserr_enable().

19.15.4.7   spcx2s() `int spcx2s (`
`            struct spcprm * spc,`
`            int nx,`
`            int sx,`
`            int sspec,`
`            const double x[],`
`            double spec[],`
`            int stat[] )`

spcx2s() transforms intermediate world coordinates to spectral coordinates.

Parameters

in,out	*spc*	Spectral transformation parameters.
in	*nx*	Vector length.
in	*sx*	Vector stride.
in	*sspec*	Vector stride.
in	*x*	Intermediate world coordinates, in SI units.
out	*spec*	Spectral coordinates, in SI units.
out	*stat*	Status return value status for each vector element: • 0: Success. • 1: Invalid value of x.

Returns

Status return value:

- 0: Success.
- 1: Null spcprm pointer passed.
- 2: Invalid spectral parameters.
- 3: One or more of the x coordinates were invalid, as indicated by the stat vector.

For returns > 1, a detailed error message is set in spcprm::err if enabled, see wcserr_enable().

19.15.4.8 spcs2x() `int spcs2x (`
        `struct` [`spcprm`](#) `* spc,`
        `int nspec,`
        `int sspec,`
        `int sx,`
        `const double spec[],`
        `double x[],`
        `int stat[] )`

spcs2x() transforms spectral world coordinates to intermediate world coordinates.

Parameters

`in,out`	*spc*	Spectral transformation parameters.
`in`	*nspec*	Vector length.
`in`	*sspec*	Vector stride.
`in`	*sx*	Vector stride.
`in`	*spec*	Spectral coordinates, in SI units.
`out`	*x*	Intermediate world coordinates, in SI units.

`out`	*stat*	Status return value status for each vector element:
		
		    • 0: Success.
		
		    • 1: Invalid value of spec.

Returns

Status return value:

- 0: Success.
- 1: Null spcprm pointer passed.
- 2: Invalid spectral parameters.
- 4: One or more of the spec coordinates were invalid, as indicated by the stat vector.

For returns > 1, a detailed error message is set in [spcprm::err](#) if enabled, see [wcserr_enable()](#).

19.15.4.9 spctype() `int spctype (`
        `const char ctype[9],`
        `char stype[],`
        `char scode[],`
        `char sname[],`
        `char units[],`
        `char * ptype,`
        `char * xtype,`
        `int * restreq,`
        `struct` [`wcserr`](#) `** err )`

spctype() checks whether a **CTYPE**i*a* keyvalue is a valid spectral axis type and if so returns information derived from it relating to the associated S-, P-, and X-type spectral variables (see explanation above).

The return arguments are guaranteed not be modified if **CTYPE**i*a* is not a valid spectral type; zero-pointers may be specified for any that are not of interest.

A deprecated form of this function, [spctyp()](#), lacks the wcserr∗∗ parameter.

Parameters

in	*ctype*	The **CTYPE**ia keyvalue, (eight characters with null termination).
out	*stype*	The four-letter name of the S-type spectral variable copied or translated from ctype. If a non-zero pointer is given, the array must accomodate a null- terminated string of length 5.
out	*scode*	The three-letter spectral algorithm code copied or translated from ctype. Logarithmic ('**LOG**') and tabular ('**TAB**') codes are also recognized. If a non-zero pointer is given, the array must accomodate a null-terminated string of length 4.
out	*sname*	Descriptive name of the S-type spectral variable. If a non-zero pointer is given, the array must accomodate a null-terminated string of length 22.
out	*units*	SI units of the S-type spectral variable. If a non-zero pointer is given, the array must accomodate a null-terminated string of length 8.
out	*ptype*	Character code for the P-type spectral variable derived from ctype, one of 'F', 'W', 'A', or 'V'.
out	*xtype*	Character code for the X-type spectral variable derived from ctype, one of 'F', 'W', 'A', or 'V'. Also, 'w' and 'a' are synonymous to 'W' and 'A' for grisms in vacuo and air respectively. Set to 'L' or 'T' for logarithmic ('**LOG**') and tabular ('**TAB**') axes.
out	*restreq*	Multivalued flag that indicates whether rest frequency or wavelength is required to compute spectral variables for this **CTYPE**ia: • 0: Not required. • 1: Required for the conversion between S- and P-types (e.g. **'ZOPT-F2W'**). • 2: Required for the conversion between P- and X-types (e.g. **'BETA-W2V'**). • 3: Required for the conversion between S- and P-types, and between P- and X-types, but not between S- and X-types (this applies only for **'VRAD-V2F'**, **'VOPT-V2W'**, and **'ZOPT-V2W'**). Thus the rest frequency or wavelength is required for spectral coordinate computations (i.e. between S- and X-types) only if `restreq%3 != 0` .
out	*err*	If enabled, for function return values > 1, this struct will contain a detailed error message, see wcserr_enable(). May be NULL if an error message is not desired. Otherwise, the user is responsible for deleting the memory allocated for the wcserr struct.

Returns

Status return value:

• 0: Success.

• 2: Invalid spectral parameters (not a spectral **CTYPE**ia).

19.15.4.10 spcspxe() int spcspxe (
```
        const char ctypeS[9],
        double crvalS,
        double restfrq,
        double restwav,
        char * ptype,
        char * xtype,
        int * restreq,
        double * crvalX,
```

```
        double * dXdS,
        struct wcserr ** err )
```

spcspxe() analyses the **CTYPE**ia and **CRVAL**ia FITS spectral axis keyword values and returns information about the associated X-type spectral variable.

A deprecated form of this function, spcspx(), lacks the wcserr∗∗ parameter.

Parameters

in	*ctypeS*	Spectral axis type, i.e. the **CTYPE**ia keyvalue, (eight characters with null termination). For non-grism axes, the character code for the P-type spectral variable in the algorithm code (i.e. the eighth character of **CTYPE**ia) may be set to '?' (it will not be reset).
in	*crvalS*	Value of the S-type spectral variable at the reference point, i.e. the **CRVAL**ia keyvalue, SI units.
in	*restfrq,restwav*	Rest frequency [Hz] and rest wavelength in vacuo [m], only one of which need be given, the other should be set to zero.
out	*ptype*	Character code for the P-type spectral variable derived from ctypeS, one of 'F', 'W', 'A', or 'V'.
out	*xtype*	Character code for the X-type spectral variable derived from ctypeS, one of 'F', 'W', 'A', or 'V'. Also, 'w' and 'a' are synonymous to 'W' and 'A' for grisms in vacuo and air respectively; crvalX and dXdS (see below) will conform to these.
out	*restreq*	Multivalued flag that indicates whether rest frequency or wavelength is required to compute spectral variables for this **CTYPE**ia, as for spctype().
out	*crvalX*	Value of the X-type spectral variable at the reference point, SI units.
out	*dXdS*	The derivative, dX/dS, evaluated at the reference point, SI units. Multiply the **CDELT**ia keyvalue by this to get the pixel spacing in the X-type spectral coordinate.
out	*err*	If enabled, for function return values > 1, this struct will contain a detailed error message, see wcserr_enable(). May be NULL if an error message is not desired. Otherwise, the user is responsible for deleting the memory allocated for the wcserr struct.

Returns

Status return value:

- 0: Success.
- 2: Invalid spectral parameters.

19.15.4.11 spcxpse() int spcxpse (
```
        const char ctypeS[9],
        double crvalX,
        double restfrq,
        double restwav,
        char * ptype,
        char * xtype,
        int * restreq,
        double * crvalS,
        double * dSdX,
        struct wcserr ** err )
```

spcxpse(), for the spectral axis type specified and the value provided for the X-type spectral variable at the reference point, deduces the value of the FITS spectral axis keyword **CRVAL**ia and also the derivative dS/dX which may be used to compute **CDELT**ia. See above for an explanation of the S-, P-, and X-type spectral variables.

A deprecated form of this function, spcxps(), lacks the wcserr∗∗ parameter.

Parameters

in	ctypeS	The required spectral axis type, i.e. the **CTYPE**ia keyvalue, (eight characters with null termination). For non-grism axes, the character code for the P-type spectral variable in the algorithm code (i.e. the eighth character of **CTYPE**ia) may be set to '?' (it will not be reset).
in	crvalX	Value of the X-type spectral variable at the reference point (N.B. NOT the **CRVAL**ia keyvalue), SI units.
in	restfrq,restwav	Rest frequency [Hz] and rest wavelength in vacuo [m], only one of which need be given, the other should be set to zero.
out	ptype	Character code for the P-type spectral variable derived from ctypeS, one of 'F', 'W', 'A', or 'V'.
out	xtype	Character code for the X-type spectral variable derived from ctypeS, one of 'F', 'W', 'A', or 'V'. Also, 'w' and 'a' are synonymous to 'W' and 'A' for grisms; crvalX and cdeltX must conform to these.
out	restreq	Multivalued flag that indicates whether rest frequency or wavelength is required to compute spectral variables for this **CTYPE**ia, as for spctype().
out	crvalS	Value of the S-type spectral variable at the reference point (i.e. the appropriate **CRVAL**ia keyvalue), SI units.
out	dSdX	The derivative, dS/dX, evaluated at the reference point, SI units. Multiply this by the pixel spacing in the X-type spectral coordinate to get the **CDELT**ia keyvalue.
out	err	If enabled, for function return values > 1, this struct will contain a detailed error message, see wcserr_enable(). May be NULL if an error message is not desired. Otherwise, the user is responsible for deleting the memory allocated for the wcserr struct.

Returns

Status return value:

- 0: Success.
- 2: Invalid spectral parameters.

19.15.4.12   spctrne()   int spctrne (
        const char *ctypeS1[9],*
        double *crvalS1,*
        double *cdeltS1,*
        double *restfrq,*
        double *restwav,*
        char *ctypeS2[9],*
        double ∗ *crvalS2,*
        double ∗ *cdeltS2,*
        struct wcserr ∗∗ *err*)

spctrne() translates a set of FITS spectral axis keywords into the corresponding set for the specified spectral axis type. For example, a '**FREQ**' axis may be translated into **'ZOPT-F2W'** and vice versa.

A deprecated form of this function, spctrn(), lacks the wcserr∗∗ parameter.

Parameters

in	*ctypeS1*	Spectral axis type, i.e. the **CTYPE**ia keyvalue, (eight characters with null termination). For non-grism axes, the character code for the P-type spectral variable in the algorithm code (i.e. the eighth character of **CTYPE**ia) may be set to '?' (it will not be reset).
in	*crvalS1*	Value of the S-type spectral variable at the reference point, i.e. the **CRVAL**ia keyvalue, SI units.
in	*cdeltS1*	Increment of the S-type spectral variable at the reference point, SI units.
in	*restfrq,restwav*	Rest frequency [Hz] and rest wavelength in vacuo [m], only one of which need be given, the other should be set to zero. Neither are required if the translation is between wave-characteristic types, or between velocity-characteristic types. E.g., required for **'FREQ'** -> **'ZOPT-F2W'**, but not required for **'VELO-F2V'** -> **'ZOPT-F2W'**.
in,out	*ctypeS2*	Required spectral axis type (eight characters with null termination). The first four characters are required to be given and are never modified. The remaining four, the algorithm code, are completely determined by, and must be consistent with, ctypeS1 and the first four characters of ctypeS2. A non-zero status value will be returned if they are inconsistent (see below). However, if the final three characters are specified as "???", or if just the eighth character is specified as '?', the correct algorithm code will be substituted (applies for grism axes as well as non-grism).
out	*crvalS2*	Value of the new S-type spectral variable at the reference point, i.e. the new **CRVAL**ia keyvalue, SI units.
out	*cdeltS2*	Increment of the new S-type spectral variable at the reference point, i.e. the new **CDELT**ia keyvalue, SI units.
out	*err*	If enabled, for function return values > 1, this struct will contain a detailed error message, see wcserr_enable(). May be NULL if an error message is not desired. Otherwise, the user is responsible for deleting the memory allocated for the wcserr struct.

Returns

Status return value:

- 0: Success.
- 2: Invalid spectral parameters.

A status value of 2 will be returned if restfrq or restwav are not specified when required, or if ctypeS1 or ctypeS2 are self-inconsistent, or have different spectral X-type variables.

19.15.4.13 spcaips() `int spcaips (`
        `const char ctypeA[9],`
        `int velref,`
        `char ctype[9],`
        `char specsys[9] )`

spcaips() translates AIPS-convention spectral **CTYPE**ia and **VELREF** keyvalues.

Parameters

in	*ctypeA*	**CTYPE**ia keyvalue possibly containing an AIPS-convention spectral code (eight characters, need not be null-terminated).
in	*velref*	AIPS-convention **VELREF** code. It has the following integer values: • 1: LSR kinematic, originally described simply as "LSR" without distinction between the kinematic and dynamic definitions. • 2: Barycentric, originally described as "HEL" meaning heliocentric. • 3: Topocentric, originally described as "OBS" meaning geocentric but widely interpreted as topocentric. AIPS++ extensions to **VELREF** are also recognized: • 4: LSR dynamic. • 5: Geocentric. • 6: Source rest frame. • 7: Galactocentric. For an AIPS '**VELO**' axis, a radio convention velocity (VRAD) is denoted by adding 256 to **VELREF**, otherwise an optical velocity (VOPT) is indicated (this is not applicable to '**FREQ**' or '**FELO**' axes). Setting velref to 0 or 256 chooses between optical and radio velocity without specifying a Doppler frame, provided that a frame is encoded in ctypeA. If not, i.e. for ctypeA = '**VELO**', ctype will be returned as '**VELO**'. **VELREF** takes precedence over **CTYPE**ia in defining the Doppler frame, e.g. `ctypeA = 'VELO-HEL' velref = 1` returns ctype = '**VOPT**' with specsys set to 'LSRK'. If omitted from the header, the default value of **VELREF** is 0.
out	*ctype*	Translated **CTYPE**ia keyvalue, or a copy of ctypeA if no translation was performed (in which case any trailing blanks in ctypeA will be replaced with nulls).
out	*specsys*	Doppler reference frame indicated by **VELREF** or else by **CTYPE**ia with value corresponding to the SPECSYS keyvalue in the FITS WCS standard. May be returned blank if neither specifies a Doppler frame, e.g. ctypeA = '**FELO**' and velref%256 == 0.

Returns

Status return value:

• -1: No translation required (not an error).
• 0: Success.
• 2: Invalid value of **VELREF**.

19.15.4.14   spctyp() int spctyp (
        const char *ctype[9],*
        char *stype[],*
        char *scode[],*
        char *sname[],*
        char *units[],*
        char * *ptype,*
        char * *xtype,*
        int * *restreq*)

19.15.4.15 spcspx() `int spcspx (`
`        const char ctypeS[9],`
`        double crvalS,`
`        double restfrq,`
`        double restwav,`
`        char * ptype,`
`        char * xtype,`
`        int * restreq,`
`        double * crvalX,`
`        double * dXdS )`

19.15.4.16 spcxps() `int spcxps (`
`        const char ctypeS[9],`
`        double crvalX,`
`        double restfrq,`
`        double restwav,`
`        char * ptype,`
`        char * xtype,`
`        int * restreq,`
`        double * crvalS,`
`        double * dSdX )`

19.15.4.17 spctrn() `int spctrn (`
`        const char ctypeS1[9],`
`        double crvalS1,`
`        double cdeltS1,`
`        double restfrq,`
`        double restwav,`
`        char ctypeS2[9],`
`        double * crvalS2,`
`        double * cdeltS2 )`

19.15.5 Variable Documentation

19.15.5.1 spc_errmsg `const char * spc_errmsg[]` `[extern]`

Error messages to match the status value returned from each function.

19.16 spc.h

```
1 /*============================================================================
2   WCSLIB 7.11 - an implementation of the FITS WCS standard.
3   Copyright (C) 1995-2022, Mark Calabretta
4
5   This file is part of WCSLIB.
6
7   WCSLIB is free software: you can redistribute it and/or modify it under the
8   terms of the GNU Lesser General Public License as published by the Free
9   Software Foundation, either version 3 of the License, or (at your option)
10  any later version.
11
12  WCSLIB is distributed in the hope that it will be useful, but WITHOUT ANY
13  WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
14  FOR A PARTICULAR PURPOSE.  See the GNU Lesser General Public License for
15  more details.
16
17  You should have received a copy of the GNU Lesser General Public License
18  along with WCSLIB.  If not, see http://www.gnu.org/licenses.
19
20  Author: Mark Calabretta, Australia Telescope National Facility, CSIRO.
21  http://www.atnf.csiro.au/people/Mark.Calabretta
22  $Id: spc.h,v 7.11 2022/04/26 06:13:52 mcalabre Exp $
23 *=============================================================================
24 *
25 * WCSLIB 7.11 - C routines that implement the FITS World Coordinate System
26 * (WCS) standard.  Refer to the README file provided with WCSLIB for an
27 * overview of the library.
28 *
29 *
30 * Summary of the spc routines
31 * ---------------------------
32 * Routines in this suite implement the part of the FITS World Coordinate
33 * System (WCS) standard that deals with spectral coordinates, as described in
34 *
35 =   "Representations of world coordinates in FITS",
36 =   Greisen, E.W., & Calabretta, M.R. 2002, A&A, 395, 1061 (WCS Paper I)
37 =
38 =   "Representations of spectral coordinates in FITS",
39 =   Greisen, E.W., Calabretta, M.R., Valdes, F.G., & Allen, S.L.
40 =   2006, A&A, 446, 747 (WCS Paper III)
41 *
42 * These routines define methods to be used for computing spectral world
43 * coordinates from intermediate world coordinates (a linear transformation
44 * of image pixel coordinates), and vice versa.  They are based on the spcprm
45 * struct which contains all information needed for the computations.  The
46 * struct contains some members that must be set by the user, and others that
47 * are maintained by these routines, somewhat like a C++ class but with no
48 * encapsulation.
49 *
50 * Routine spcini() is provided to initialize the spcprm struct with default
51 * values, spcfree() reclaims any memory that may have been allocated to store
52 * an error message, spcsize() computes its total size including allocated
53 * memory, and spcprt() prints its contents.
54 *
55 * spcperr() prints the error message(s) (if any) stored in a spcprm struct.
56 *
57 * A setup routine, spcset(), computes intermediate values in the spcprm struct
58 * from parameters in it that were supplied by the user.  The struct always
59 * needs to be set up by spcset() but it need not be called explicitly - refer
60 * to the explanation of spcprm::flag.
61 *
62 * spcx2s() and spcs2x() implement the WCS spectral coordinate transformations.
63 * In fact, they are high level driver routines for the lower level spectral
64 * coordinate transformation routines described in spx.h.
65 *
66 * A number of routines are provided to aid in analysing or synthesising sets
67 * of FITS spectral axis keywords:
68 *
69 *   - spctype() checks a spectral CTYPEia keyword for validity and returns
70 *     information derived from it.
71 *
72 *   - Spectral keyword analysis routine spcspxe() computes the values of the
73 *     X-type spectral variables for the S-type variables supplied.
74 *
75 *   - Spectral keyword synthesis routine, spcxpse(), computes the S-type
76 *     variables for the X-types supplied.
77 *
78 *   - Given a set of spectral keywords, a translation routine, spctrne(),
79 *     produces the corresponding set for the specified spectral CTYPEia.
80 *
81 *   - spcaips() translates AIPS-convention spectral CTYPEia and VELREF
82 *     keyvalues.
83 *
```

```
 84 * Spectral variable types - S, P, and X:
 85 * -------------------------------------
 86 * A few words of explanation are necessary regarding spectral variable types
 87 * in FITS.
 88 *
 89 * Every FITS spectral axis has three associated spectral variables:
 90 *
 91 *   S-type: the spectral variable in which coordinates are to be
 92 *     expressed.  Each S-type is encoded as four characters and is
 93 *     linearly related to one of four basic types as follows:
 94 *
 95 *     F (Frequency):
 96 *       - 'FREQ':  frequency
 97 *       - 'AFRQ':  angular frequency
 98 *       - 'ENER':  photon energy
 99 *       - 'WAVN':  wave number
100 *       - 'VRAD':  radio velocity
101 *
102 *     W (Wavelength in vacuo):
103 *       - 'WAVE':  wavelength
104 *       - 'VOPT':  optical velocity
105 *       - 'ZOPT':  redshift
106 *
107 *     A (wavelength in Air):
108 *       - 'AWAV':  wavelength in air
109 *
110 *     V (Velocity):
111 *       - 'VELO':  relativistic velocity
112 *       - 'BETA':  relativistic beta factor
113 *
114 *     The S-type forms the first four characters of the CTYPEia keyvalue,
115 *     and CRVALia and CDELTia are expressed as S-type quantities so that
116 *     they provide a first-order approximation to the S-type variable at
117 *     the reference point.
118 *
119 *     Note that 'AFRQ', angular frequency, is additional to the variables
120 *     defined in WCS Paper III.
121 *
122 *   P-type: the basic spectral variable (F, W, A, or V) with which the
123 *     S-type variable is associated (see list above).
124 *
125 *     For non-grism axes, the P-type is encoded as the eighth character of
126 *     CTYPEia.
127 *
128 *   X-type: the basic spectral variable (F, W, A, or V) for which the
129 *     spectral axis is linear, grisms excluded (see below).
130 *
131 *     For non-grism axes, the X-type is encoded as the sixth character of
132 *     CTYPEia.
133 *
134 *   Grisms: Grism axes have normal S-, and P-types but the axis is linear,
135 *     not in any spectral variable, but in a special "grism parameter".
136 *     The X-type spectral variable is either W or A for grisms in vacuo or
137 *     air respectively, but is encoded as 'w' or 'a' to indicate that an
138 *     additional transformation is required to convert to or from the
139 *     grism parameter.  The spectral algorithm code for grisms also has a
140 *     special encoding in CTYPEia, either 'GRI' (in vacuo) or 'GRA' (in air).
141 *
142 * In the algorithm chain, the non-linear transformation occurs between the
143 * X-type and the P-type variables; the transformation between P-type and
144 * S-type variables is always linear.
145 *
146 * When the P-type and X-type variables are the same, the spectral axis is
147 * linear in the S-type variable and the second four characters of CTYPEia
148 * are blank.  This can never happen for grism axes.
149 *
150 * As an example, correlating radio spectrometers always produce spectra that
151 * are regularly gridded in frequency; a redshift scale on such a spectrum is
152 * non-linear.  The required value of CTYPEia would be 'ZOPT-F2W', where the
153 * desired S-type is 'ZOPT' (redshift), the P-type is necessarily 'W'
154 * (wavelength), and the X-type is 'F' (frequency) by the nature of the
155 * instrument.
156 *
157 * Air-to-vacuum wavelength conversion:
158 * -----------------------------------
159 * Please refer to the prologue of spx.h for important comments relating to the
160 * air-to-vacuum wavelength conversion.
161 *
162 * Argument checking:
163 * ------------------
164 * The input spectral values are only checked for values that would result in
165 * floating point exceptions.  In particular, negative frequencies and
166 * wavelengths are allowed, as are velocities greater than the speed of
167 * light.  The same is true for the spectral parameters - rest frequency and
168 * wavelength.
169 *
170 * Accuracy:
```

```
171 * ---------
172 * No warranty is given for the accuracy of these routines (refer to the
173 * copyright notice); intending users must satisfy for themselves their
174 * adequacy for the intended purpose.  However, closure effectively to within
175 * double precision rounding error was demonstrated by test routine tspc.c
176 * which accompanies this software.
177 *
178 *
179 * spcini() - Default constructor for the spcprm struct
180 * ----------------------------------------------------
181 * spcini() sets all members of a spcprm struct to default values.  It should
182 * be used to initialize every spcprm struct.
183 *
184 * PLEASE NOTE: If the spcprm struct has already been initialized, then before
185 * reinitializing, it spcfree() should be used to free any memory that may have
186 * been allocated to store an error message.  A memory leak may otherwise
187 * result.
188 *
189 * Given and returned:
190 *   spc       struct spcprm*
191 *                       Spectral transformation parameters.
192 *
193 * Function return value:
194 *             int       Status return value:
195 *                         0: Success.
196 *                         1: Null spcprm pointer passed.
197 *
198 *
199 * spcfree() - Destructor for the spcprm struct
200 * --------------------------------------------
201 * spcfree() frees any memory that may have been allocated to store an error
202 * message in the spcprm struct.
203 *
204 * Given:
205 *   spc       struct spcprm*
206 *                       Spectral transformation parameters.
207 *
208 * Function return value:
209 *             int       Status return value:
210 *                         0: Success.
211 *                         1: Null spcprm pointer passed.
212 *
213 *
214 * spcsize() - Compute the size of a spcprm struct
215 * -----------------------------------------------
216 * spcsize() computes the full size of a spcprm struct, including allocated
217 * memory.
218 *
219 * Given:
220 *   spc       const struct spcprm*
221 *                       Spectral transformation parameters.
222 *
223 *                       If NULL, the base size of the struct and the allocated
224 *                       size are both set to zero.
225 *
226 * Returned:
227 *   sizes     int[2]    The first element is the base size of the struct as
228 *                       returned by sizeof(struct spcprm).  The second element
229 *                       is the total allocated size, in bytes.  This figure
230 *                       includes memory allocated for the constituent struct,
231 *                       spcprm::err.
232 *
233 *                       It is not an error for the struct not to have been set
234 *                       up via spcset().
235 *
236 * Function return value:
237 *             int       Status return value:
238 *                         0: Success.
239 *
240 *
241 * spcprt() - Print routine for the spcprm struct
242 * ----------------------------------------------
243 * spcprt() prints the contents of a spcprm struct using wcsprintf().  Mainly
244 * intended for diagnostic purposes.
245 *
246 * Given:
247 *   spc       const struct spcprm*
248 *                       Spectral transformation parameters.
249 *
250 * Function return value:
251 *             int       Status return value:
252 *                         0: Success.
253 *                         1: Null spcprm pointer passed.
254 *
255 *
256 * spcperr() - Print error messages from a spcprm struct
257 * -----------------------------------------------------
```

```
258 * spcperr() prints the error message(s) (if any) stored in a spcprm struct.
259 * If there are no errors then nothing is printed.  It uses wcserr_prt(), q.v.
260 *
261 * Given:
262 *   spc       const struct spcprm*
263 *                       Spectral transformation parameters.
264 *
265 *   prefix    const char *
266 *                       If non-NULL, each output line will be prefixed with
267 *                       this string.
268 *
269 * Function return value:
270 *             int       Status return value:
271 *                         0: Success.
272 *                         1: Null spcprm pointer passed.
273 *
274 *
275 * spcset() - Setup routine for the spcprm struct
276 * ----------------------------------------------
277 * spcset() sets up a spcprm struct according to information supplied within
278 * it.
279 *
280 * Note that this routine need not be called directly; it will be invoked by
281 * spcx2s() and spcs2x() if spcprm::flag is anything other than a predefined
282 * magic value.
283 *
284 * Given and returned:
285 *   spc       struct spcprm*
286 *                       Spectral transformation parameters.
287 *
288 * Function return value:
289 *             int       Status return value:
290 *                         0: Success.
291 *                         1: Null spcprm pointer passed.
292 *                         2: Invalid spectral parameters.
293 *
294 *                       For returns > 1, a detailed error message is set in
295 *                       spcprm::err if enabled, see wcserr_enable().
296 *
297 *
298 * spcx2s() - Transform to spectral coordinates
299 * --------------------------------------------
300 * spcx2s() transforms intermediate world coordinates to spectral coordinates.
301 *
302 * Given and returned:
303 *   spc       struct spcprm*
304 *                       Spectral transformation parameters.
305 *
306 * Given:
307 *   nx        int       Vector length.
308 *
309 *   sx        int       Vector stride.
310 *
311 *   sspec     int       Vector stride.
312 *
313 *   x         const double[]
314 *                       Intermediate world coordinates, in SI units.
315 *
316 * Returned:
317 *   spec      double[]  Spectral coordinates, in SI units.
318 *
319 *   stat      int[]     Status return value status for each vector element:
320 *                         0: Success.
321 *                         1: Invalid value of x.
322 *
323 * Function return value:
324 *             int       Status return value:
325 *                         0: Success.
326 *                         1: Null spcprm pointer passed.
327 *                         2: Invalid spectral parameters.
328 *                         3: One or more of the x coordinates were invalid,
329 *                            as indicated by the stat vector.
330 *
331 *                       For returns > 1, a detailed error message is set in
332 *                       spcprm::err if enabled, see wcserr_enable().
333 *
334 *
335 * spcs2x() - Transform spectral coordinates
336 * -----------------------------------------
337 * spcs2x() transforms spectral world coordinates to intermediate world
338 * coordinates.
339 *
340 * Given and returned:
341 *   spc       struct spcprm*
342 *                       Spectral transformation parameters.
343 *
344 * Given:
```

```
345 *   nspec     int       Vector length.
346 *
347 *   sspec     int       Vector stride.
348 *
349 *   sx        int       Vector stride.
350 *
351 *   spec      const double[]
352 *                       Spectral coordinates, in SI units.
353 *
354 * Returned:
355 *   x         double[]  Intermediate world coordinates, in SI units.
356 *
357 *   stat      int[]     Status return value status for each vector element:
358 *                         0: Success.
359 *                         1: Invalid value of spec.
360 *
361 * Function return value:
362 *             int       Status return value:
363 *                         0: Success.
364 *                         1: Null spcprm pointer passed.
365 *                         2: Invalid spectral parameters.
366 *                         4: One or more of the spec coordinates were
367 *                            invalid, as indicated by the stat vector.
368 *
369 *                       For returns > 1, a detailed error message is set in
370 *                       spcprm::err if enabled, see wcserr_enable().
371 *
372 *
373 * spctype() - Spectral CTYPEia keyword analysis
374 * ---------------------------------------------
375 * spctype() checks whether a CTYPEia keyvalue is a valid spectral axis type
376 * and if so returns information derived from it relating to the associated S-,
377 * P-, and X-type spectral variables (see explanation above).
378 *
379 * The return arguments are guaranteed not be modified if CTYPEia is not a
380 * valid spectral type; zero-pointers may be specified for any that are not of
381 * interest.
382 *
383 * A deprecated form of this function, spctyp(), lacks the wcserr** parameter.
384 *
385 * Given:
386 *   ctype     const char[9]
387 *                       The CTYPEia keyvalue, (eight characters with null
388 *                       termination).
389 *
390 * Returned:
391 *   stype     char[]    The four-letter name of the S-type spectral variable
392 *                       copied or translated from ctype.  If a non-zero
393 *                       pointer is given, the array must accomodate a null-
394 *                       terminated string of length 5.
395 *
396 *   scode     char[]    The three-letter spectral algorithm code copied or
397 *                       translated from ctype.  Logarithmic ('LOG') and
398 *                       tabular ('TAB') codes are also recognized.  If a
399 *                       non-zero pointer is given, the array must accomodate a
400 *                       null-terminated string of length 4.
401 *
402 *   sname     char[]    Descriptive name of the S-type spectral variable.
403 *                       If a non-zero pointer is given, the array must
404 *                       accomodate a null-terminated string of length 22.
405 *
406 *   units     char[]    SI units of the S-type spectral variable.  If a
407 *                       non-zero pointer is given, the array must accomodate a
408 *                       null-terminated string of length 8.
409 *
410 *   ptype     char*     Character code for the P-type spectral variable
411 *                       derived from ctype, one of 'F', 'W', 'A', or 'V'.
412 *
413 *   xtype     char*     Character code for the X-type spectral variable
414 *                       derived from ctype, one of 'F', 'W', 'A', or 'V'.
415 *                       Also, 'w' and 'a' are synonymous to 'W' and 'A' for
416 *                       grisms in vacuo and air respectively.  Set to 'L' or
417 *                       'T' for logarithmic ('LOG') and tabular ('TAB') axes.
418 *
419 *   restreq   int*      Multivalued flag that indicates whether rest
420 *                       frequency or wavelength is required to compute
421 *                       spectral variables for this CTYPEia:
422 *                         0: Not required.
423 *                         1: Required for the conversion between S- and
424 *                            P-types (e.g. 'ZOPT-F2W').
425 *                         2: Required for the conversion between P- and
426 *                            X-types (e.g. 'BETA-W2V').
427 *                         3: Required for the conversion between S- and
428 *                            P-types, and between P- and X-types, but not
429 *                            between S- and X-types (this applies only for
430 *                            'VRAD-V2F', 'VOPT-V2W', and 'ZOPT-V2W').
431 *                         Thus the rest frequency or wavelength is required for
```

```
432 *                              spectral coordinate computations (i.e. between S- and
433 *                              X-types) only if restreq%3 != 0.
434 *
435 *   err        struct wcserr **
436 *                              If enabled, for function return values > 1, this
437 *                              struct will contain a detailed error message, see
438 *                              wcserr_enable().  May be NULL if an error message is
439 *                              not desired.  Otherwise, the user is responsible for
440 *                              deleting the memory allocated for the wcserr struct.
441 *
442 * Function return value:
443 *              int        Status return value:
444 *                            0: Success.
445 *                            2: Invalid spectral parameters (not a spectral
446 *                               CTYPEia).
447 *
448 *
449 * spcspxe() - Spectral keyword analysis
450 * -------------------------------------
451 * spcspxe() analyses the CTYPEia and CRVALia FITS spectral axis keyword values
452 * and returns information about the associated X-type spectral variable.
453 *
454 * A deprecated form of this function, spcspx(), lacks the wcserr** parameter.
455 *
456 * Given:
457 *   ctypeS     const char[9]
458 *                              Spectral axis type, i.e. the CTYPEia keyvalue, (eight
459 *                              characters with null termination).  For non-grism
460 *                              axes, the character code for the P-type spectral
461 *                              variable in the algorithm code (i.e. the eighth
462 *                              character of CTYPEia) may be set to '?' (it will not
463 *                              be reset).
464 *
465 *   crvalS     double     Value of the S-type spectral variable at the reference
466 *                              point, i.e. the CRVALia keyvalue, SI units.
467 *
468 *   restfrq,
469 *   restwav    double     Rest frequency [Hz] and rest wavelength in vacuo [m],
470 *                              only one of which need be given, the other should be
471 *                              set to zero.
472 *
473 * Returned:
474 *   ptype      char*      Character code for the P-type spectral variable
475 *                              derived from ctypeS, one of 'F', 'W', 'A', or 'V'.
476 *
477 *   xtype      char*      Character code for the X-type spectral variable
478 *                              derived from ctypeS, one of 'F', 'W', 'A', or 'V'.
479 *                              Also, 'w' and 'a' are synonymous to 'W' and 'A' for
480 *                              grisms in vacuo and air respectively; crvalX and dXdS
481 *                              (see below) will conform to these.
482 *
483 *   restreq    int*       Multivalued flag that indicates whether rest frequency
484 *                              or wavelength is required to compute spectral
485 *                              variables for this CTYPEia, as for spctype().
486 *
487 *   crvalX     double*    Value of the X-type spectral variable at the reference
488 *                              point, SI units.
489 *
490 *   dXdS       double*    The derivative, dX/dS, evaluated at the reference
491 *                              point, SI units.  Multiply the CDELTia keyvalue by
492 *                              this to get the pixel spacing in the X-type spectral
493 *                              coordinate.
494 *
495 *   err        struct wcserr **
496 *                              If enabled, for function return values > 1, this
497 *                              struct will contain a detailed error message, see
498 *                              wcserr_enable().  May be NULL if an error message is
499 *                              not desired.  Otherwise, the user is responsible for
500 *                              deleting the memory allocated for the wcserr struct.
501 *
502 * Function return value:
503 *              int        Status return value:
504 *                            0: Success.
505 *                            2: Invalid spectral parameters.
506 *
507 *
508 * spcxpse() - Spectral keyword synthesis
509 * --------------------------------------
510 * spcxpse(), for the spectral axis type specified and the value provided for
511 * the X-type spectral variable at the reference point, deduces the value of
512 * the FITS spectral axis keyword CRVALia and also the derivative dS/dX which
513 * may be used to compute CDELTia.  See above for an explanation of the S-,
514 * P-, and X-type spectral variables.
515 *
516 * A deprecated form of this function, spcxps(), lacks the wcserr** parameter.
517 *
518 * Given:
```

```
519 *    ctypeS     const char[9]
520 *                          The required spectral axis type, i.e. the CTYPEia
521 *                          keyvalue, (eight characters with null termination).
522 *                          For non-grism axes, the character code for the P-type
523 *                          spectral variable in the algorithm code (i.e. the
524 *                          eighth character of CTYPEia) may be set to '?' (it
525 *                          will not be reset).
526 *
527 *    crvalX     double     Value of the X-type spectral variable at the reference
528 *                          point (N.B. NOT the CRVALia keyvalue), SI units.
529 *
530 *    restfrq,
531 *    restwav    double     Rest frequency [Hz] and rest wavelength in vacuo [m],
532 *                          only one of which need be given, the other should be
533 *                          set to zero.
534 *
535 * Returned:
536 *    ptype      char*      Character code for the P-type spectral variable
537 *                          derived from ctypeS, one of 'F', 'W', 'A', or 'V'.
538 *
539 *    xtype      char*      Character code for the X-type spectral variable
540 *                          derived from ctypeS, one of 'F', 'W', 'A', or 'V'.
541 *                          Also, 'w' and 'a' are synonymous to 'W' and 'A' for
542 *                          grisms; crvalX and cdeltX must conform to these.
543 *
544 *    restreq    int*       Multivalued flag that indicates whether rest frequency
545 *                          or wavelength is required to compute spectral
546 *                          variables for this CTYPEia, as for spctype().
547 *
548 *    crvalS     double*    Value of the S-type spectral variable at the reference
549 *                          point (i.e. the appropriate CRVALia keyvalue), SI
550 *                          units.
551 *
552 *    dSdX       double*    The derivative, dS/dX, evaluated at the reference
553 *                          point, SI units.  Multiply this by the pixel spacing
554 *                          in the X-type spectral coordinate to get the CDELTia
555 *                          keyvalue.
556 *
557 *    err        struct wcserr **
558 *                          If enabled, for function return values > 1, this
559 *                          struct will contain a detailed error message, see
560 *                          wcserr_enable().  May be NULL if an error message is
561 *                          not desired.  Otherwise, the user is responsible for
562 *                          deleting the memory allocated for the wcserr struct.
563 *
564 * Function return value:
565 *               int        Status return value:
566 *                            0: Success.
567 *                            2: Invalid spectral parameters.
568 *
569 *
570 * spctrne() - Spectral keyword translation
571 * ----------------------------------------
572 * spctrne() translates a set of FITS spectral axis keywords into the
573 * corresponding set for the specified spectral axis type.  For example, a
574 * 'FREQ' axis may be translated into 'ZOPT-F2W' and vice versa.
575 *
576 * A deprecated form of this function, spctrn(), lacks the wcserr** parameter.
577 *
578 * Given:
579 *    ctypeS1    const char[9]
580 *                          Spectral axis type, i.e. the CTYPEia keyvalue, (eight
581 *                          characters with null termination).  For non-grism
582 *                          axes, the character code for the P-type spectral
583 *                          variable in the algorithm code (i.e. the eighth
584 *                          character of CTYPEia) may be set to '?' (it will not
585 *                          be reset).
586 *
587 *    crvalS1    double     Value of the S-type spectral variable at the reference
588 *                          point, i.e. the CRVALia keyvalue, SI units.
589 *
590 *    cdeltS1    double     Increment of the S-type spectral variable at the
591 *                          reference point, SI units.
592 *
593 *    restfrq,
594 *    restwav    double     Rest frequency [Hz] and rest wavelength in vacuo [m],
595 *                          only one of which need be given, the other should be
596 *                          set to zero.  Neither are required if the translation
597 *                          is between wave-characteristic types, or between
598 *                          velocity-characteristic types.  E.g., required for
599 *                          'FREQ'    -> 'ZOPT-F2W', but not required for
600 *                          'VELO-F2V' -> 'ZOPT-F2W'.
601 *
602 * Given and returned:
603 *    ctypeS2    char[9]    Required spectral axis type (eight characters with
604 *                          null termination).  The first four characters are
605 *                          required to be given and are never modified.  The
```

```
606 *                        remaining four, the algorithm code, are completely
607 *                        determined by, and must be consistent with, ctypeS1
608 *                        and the first four characters of ctypeS2.  A non-zero
609 *                        status value will be returned if they are inconsistent
610 *                        (see below).  However, if the final three characters
611 *                        are specified as "???", or if just the eighth
612 *                        character is specified as '?', the correct algorithm
613 *                        code will be substituted (applies for grism axes as
614 *                        well as non-grism).
615 *
616 * Returned:
617 *   crvalS2   double*   Value of the new S-type spectral variable at the
618 *                        reference point, i.e. the new CRVALia keyvalue, SI
619 *                        units.
620 *
621 *   cdeltS2   double*   Increment of the new S-type spectral variable at the
622 *                        reference point, i.e. the new CDELTia keyvalue, SI
623 *                        units.
624 *
625 *   err       struct wcserr **
626 *                        If enabled, for function return values > 1, this
627 *                        struct will contain a detailed error message, see
628 *                        wcserr_enable().  May be NULL if an error message is
629 *                        not desired.  Otherwise, the user is responsible for
630 *                        deleting the memory allocated for the wcserr struct.
631 *
632 * Function return value:
633 *             int       Status return value:
634 *                          0: Success.
635 *                          2: Invalid spectral parameters.
636 *
637 *                        A status value of 2 will be returned if restfrq or
638 *                        restwav are not specified when required, or if ctypeS1
639 *                        or ctypeS2 are self-inconsistent, or have different
640 *                        spectral X-type variables.
641 *
642 *
643 * spcaips() - Translate AIPS-convention spectral keywords
644 * -------------------------------------------------------
645 * spcaips() translates AIPS-convention spectral CTYPEia and VELREF keyvalues.
646 *
647 * Given:
648 *   ctypeA    const char[9]
649 *                        CTYPEia keyvalue possibly containing an
650 *                        AIPS-convention spectral code (eight characters, need
651 *                        not be null-terminated).
652 *
653 *   velref    int       AIPS-convention VELREF code.  It has the following
654 *                        integer values:
655 *                          1: LSR kinematic, originally described simply as
656 *                             "LSR" without distinction between the kinematic
657 *                             and dynamic definitions.
658 *                          2: Barycentric, originally described as "HEL"
659 *                             meaning heliocentric.
660 *                          3: Topocentric, originally described as "OBS"
661 *                             meaning geocentric but widely interpreted as
662 *                             topocentric.
663 *                        AIPS++ extensions to VELREF are also recognized:
664 *                          4: LSR dynamic.
665 *                          5: Geocentric.
666 *                          6: Source rest frame.
667 *                          7: Galactocentric.
668 *
669 *                        For an AIPS 'VELO' axis, a radio convention velocity
670 *                        (VRAD) is denoted by adding 256 to VELREF, otherwise
671 *                        an optical velocity (VOPT) is indicated (this is not
672 *                        applicable to 'FREQ' or 'FELO' axes).  Setting velref
673 *                        to 0 or 256 chooses between optical and radio velocity
674 *                        without specifying a Doppler frame, provided that a
675 *                        frame is encoded in ctypeA.  If not, i.e. for
676 *                        ctypeA = 'VELO', ctype will be returned as 'VELO'.
677 *
678 *                        VELREF takes precedence over CTYPEia in defining the
679 *                        Doppler frame, e.g.
680 *
681 =                          ctypeA = 'VELO-HEL'
682 =                          velref = 1
683 *
684 *                        returns ctype = 'VOPT' with specsys set to 'LSRK'.
685 *
686 *                        If omitted from the header, the default value of
687 *                        VELREF is 0.
688 *
689 * Returned:
690 *   ctype     char[9]   Translated CTYPEia keyvalue, or a copy of ctypeA if no
691 *                        translation was performed (in which case any trailing
692 *                        blanks in ctypeA will be replaced with nulls).
```

```
693 *
694 *    specsys   char[9]   Doppler reference frame indicated by VELREF or else
695 *                        by CTYPEia with value corresponding to the SPECSYS
696 *                        keyvalue in the FITS WCS standard.  May be returned
697 *                        blank if neither specifies a Doppler frame, e.g.
698 *                        ctypeA = 'FELO' and velref%256 == 0.
699 *
700 * Function return value:
701 *              int        Status return value:
702 *                          -1: No translation required (not an error).
703 *                           0: Success.
704 *                           2: Invalid value of VELREF.
705 *
706 *
707 * spcprm struct - Spectral transformation parameters
708 * --------------------------------------------------
709 * The spcprm struct contains information required to transform spectral
710 * coordinates.  It consists of certain members that must be set by the user
711 * ("given") and others that are set by the WCSLIB routines ("returned").  Some
712 * of the latter are supplied for informational purposes while others are for
713 * internal use only.
714 *
715 *    int flag
716 *       (Given and returned) This flag must be set to zero whenever any of the
717 *       following spcprm structure members are set or changed:
718 *
719 *          - spcprm::type,
720 *          - spcprm::code,
721 *          - spcprm::crval,
722 *          - spcprm::restfrq,
723 *          - spcprm::restwav,
724 *          - spcprm::pv[].
725 *
726 *       This signals the initialization routine, spcset(), to recompute the
727 *       returned members of the spcprm struct.  spcset() will reset flag to
728 *       indicate that this has been done.
729 *
730 *    char type[8]
731 *       (Given) Four-letter spectral variable type, e.g "ZOPT" for
732 *       CTYPEia = 'ZOPT-F2W'.  (Declared as char[8] for alignment reasons.)
733 *
734 *    char code[4]
735 *       (Given) Three-letter spectral algorithm code, e.g "F2W" for
736 *       CTYPEia = 'ZOPT-F2W'.
737 *
738 *    double crval
739 *       (Given) Reference value (CRVALia), SI units.
740 *
741 *    double restfrq
742 *       (Given) The rest frequency [Hz], and ...
743 *
744 *    double restwav
745 *       (Given) ... the rest wavelength in vacuo [m], only one of which need be
746 *       given, the other should be set to zero.  Neither are required if the
747 *       X and S spectral variables are both wave-characteristic, or both
748 *       velocity-characteristic, types.
749 *
750 *    double pv[7]
751 *       (Given) Grism parameters for 'GRI' and 'GRA' algorithm codes:
752 *          - 0: G, grating ruling density.
753 *          - 1: m, interference order.
754 *          - 2: alpha, angle of incidence [deg].
755 *          - 3: n_r, refractive index at the reference wavelength, lambda_r.
756 *          - 4: n'_r, dn/dlambda at the reference wavelength, lambda_r (/m).
757 *          - 5: epsilon, grating tilt angle [deg].
758 *          - 6: theta, detector tilt angle [deg].
759 *
760 * The remaining members of the spcprm struct are maintained by spcset() and
761 * must not be modified elsewhere:
762 *
763 *    double w[6]
764 *       (Returned) Intermediate values:
765 *          - 0: Rest frequency or wavelength (SI).
766 *          - 1: The value of the X-type spectral variable at the reference point
767 *               (SI units).
768 *          - 2: dX/dS at the reference point (SI units).
769 *           The remainder are grism intermediates.
770 *
771 *    int isGrism
772 *       (Returned) Grism coordinates?
773 *          - 0: no,
774 *          - 1: in vacuum,
775 *          - 2: in air.
776 *
777 *    int padding1
778 *       (An unused variable inserted for alignment purposes only.)
779 *
```

```
780 *    struct wcserr *err
781 *       (Returned) If enabled, when an error status is returned, this struct
782 *       contains detailed information about the error, see wcserr_enable().
783 *
784 *    void *padding2
785 *       (An unused variable inserted for alignment purposes only.)
786 *    int (*spxX2P)(SPX_ARGS)
787 *       (Returned) The first and ...
788 *    int (*spxP2S)(SPX_ARGS)
789 *       (Returned) ... the second of the pointers to the transformation
790 *       functions in the two-step algorithm chain X -> P -> S in the
791 *       pixel-to-spectral direction where the non-linear transformation is from
792 *       X to P.  The argument list, SPX_ARGS, is defined in spx.h.
793 *
794 *    int (*spxS2P)(SPX_ARGS)
795 *       (Returned) The first and ...
796 *    int (*spxP2X)(SPX_ARGS)
797 *       (Returned) ... the second of the pointers to the transformation
798 *       functions in the two-step algorithm chain S -> P -> X in the
799 *       spectral-to-pixel direction where the non-linear transformation is from
800 *       P to X.  The argument list, SPX_ARGS, is defined in spx.h.
801 *
802 *
803 * Global variable: const char *spc_errmsg[] - Status return messages
804 * -----------------------------------------------------------------
805 * Error messages to match the status value returned from each function.
806 *
807 *===========================================================================*/
808
809 #ifndef WCSLIB_SPC
810 #define WCSLIB_SPC
811
812 #include "spx.h"
813
814 #ifdef __cplusplus
815 extern "C" {
816 #endif
817
818
819 extern const char *spc_errmsg[];
820
821 enum spc_errmsg_enum {
822   SPCERR_NO_CHANGE      = -1,  // No change.
823   SPCERR_SUCCESS        = 0,     // Success.
824   SPCERR_NULL_POINTER   = 1,      // Null spcprm pointer passed.
825   SPCERR_BAD_SPEC_PARAMS = 2,    // Invalid spectral parameters.
826   SPCERR_BAD_X          = 3,      // One or more of x coordinates were
827                                 // invalid.
828   SPCERR_BAD_SPEC       = 4    // One or more of the spec coordinates were
829                                 // invalid.
830 };
831
832 struct spcprm {
833   // Initialization flag (see the prologue above).
834   //------------------------------------------------------------------------
835   int    flag;                   // Set to zero to force initialization.
836
837   // Parameters to be provided (see the prologue above).
838   //------------------------------------------------------------------------
839   char   type[8];          // Four-letter spectral variable type.
840   char   code[4];          // Three-letter spectral algorithm code.
841
842   double crval;                  // Reference value (CRVALia), SI units.
843   double restfrq;            // Rest frequency, Hz.
844   double restwav;             // Rest wavelength, m.
845
846   double pv[7];                  // Grism parameters.
847                              //   0: G, grating ruling density.
848                              //   1: m, interference order.
849                              //   2: alpha, angle of incidence.
850                              //   3: n_r, refractive index at lambda_r.
851                              //   4: n'_r, dn/dlambda at lambda_r.
852                              //   5: epsilon, grating tilt angle.
853                              //   6: theta, detector tilt angle.
854
855   // Information derived from the parameters supplied.
856   //------------------------------------------------------------------------
857   double w[6];                 // Intermediate values.
858                              //   0: Rest frequency or wavelength (SI).
859                              //   1: CRVALX (SI units).
860                              //   2: CDELTX/CDELTia = dX/dS (SI units).
861                              // The remainder are grism intermediates.
862
863   int    isGrism;              // Grism coordinates?  1: vacuum, 2: air.
864   int    padding1;             // (Dummy inserted for alignment purposes.)
865
866   // Error handling
```

```
867    //--------------------------------------------------------------------------
868    struct wcserr *err;
869
870    // Private
871    //--------------------------------------------------------------------------
872    void   *padding2;               // (Dummy inserted for alignment purposes.)
873    int (*spxX2P)(SPX_ARGS);        // Pointers to the transformation functions
874    int (*spxP2S)(SPX_ARGS);        // in the two-step algorithm chain in the
875                                    // pixel-to-spectral direction.
876
877    int (*spxS2P)(SPX_ARGS);        // Pointers to the transformation functions
878    int (*spxP2X)(SPX_ARGS);        // in the two-step algorithm chain in the
879                                    // spectral-to-pixel direction.
880  };
881
882  // Size of the spcprm struct in int units, used by the Fortran wrappers.
883  #define SPCLEN (sizeof(struct spcprm)/sizeof(int))
884
885
886  int spcini(struct spcprm *spc);
887
888  int spcfree(struct spcprm *spc);
889
890  int spcsize(const struct spcprm *spc, int sizes[2]);
891
892  int spcprt(const struct spcprm *spc);
893
894  int spcperr(const struct spcprm *spc, const char *prefix);
895
896  int spcset(struct spcprm *spc);
897
898  int spcx2s(struct spcprm *spc, int nx, int sx, int sspec,
899             const double x[], double spec[], int stat[]);
900
901  int spcs2x(struct spcprm *spc, int nspec, int sspec, int sx,
902             const double spec[], double x[], int stat[]);
903
904  int spctype(const char ctype[9], char stype[], char scode[], char sname[],
905              char units[], char *ptype, char *xtype, int *restreq,
906              struct wcserr **err);
907
908  int spcspxe(const char ctypeS[9], double crvalS, double restfrq,
909              double restwav, char *ptype, char *xtype, int *restreq,
910              double *crvalX, double *dXdS, struct wcserr **err);
911
912  int spcxpse(const char ctypeS[9], double crvalX, double restfrq,
913              double restwav, char *ptype, char *xtype, int *restreq,
914              double *crvalS, double *dSdX, struct wcserr **err);
915
916  int spctrne(const char ctypeS1[9], double crvalS1, double cdeltS1,
917              double restfrq, double restwav, char ctypeS2[9], double *crvalS2,
918              double *cdeltS2, struct wcserr **err);
919
920  int spcaips(const char ctypeA[9], int velref, char ctype[9], char specsys[9]);
921
922
923  // Deprecated.
924  #define spcini_errmsg spc_errmsg
925  #define spcprt_errmsg spc_errmsg
926  #define spcset_errmsg spc_errmsg
927  #define spcx2s_errmsg spc_errmsg
928  #define spcs2x_errmsg spc_errmsg
929
930  int spctyp(const char ctype[9], char stype[], char scode[], char sname[],
931             char units[], char *ptype, char *xtype, int *restreq);
932  int spcspx(const char ctypeS[9], double crvalS, double restfrq,
933             double restwav, char *ptype, char *xtype, int *restreq,
934             double *crvalX, double *dXdS);
935  int spcxps(const char ctypeS[9], double crvalX, double restfrq,
936             double restwav, char *ptype, char *xtype, int *restreq,
937             double *crvalS, double *dSdX);
938  int spctrn(const char ctypeS1[9], double crvalS1, double cdeltS1,
939             double restfrq, double restwav, char ctypeS2[9], double *crvalS2,
940             double *cdeltS2);
941
942  #ifdef __cplusplus
943  }
944  #endif
945
946  #endif // WCSLIB_SPC
```

19.17   sph.h File Reference

Functions

- int sphx2s (const double eul[5], int nphi, int ntheta, int spt, int sxy, const double phi[ ], const double theta[ ], double lng[ ], double lat[ ])

    Rotation in the pixel-to-world direction.
- int sphs2x (const double eul[5], int nlng, int nlat, int sll, int spt, const double lng[ ], const double lat[ ], double phi[ ], double theta[ ])

    Rotation in the world-to-pixel direction.
- int sphdpa (int nfield, double lng0, double lat0, const double lng[ ], const double lat[ ], double dist[ ], double pa[ ])

    Compute angular distance and position angle.
- int sphpad (int nfield, double lng0, double lat0, const double dist[ ], const double pa[ ], double lng[ ], double lat[ ])

    Compute field points offset from a given point.

19.17.1   Detailed Description

Routines in this suite implement the spherical coordinate transformations defined by the FITS World Coordinate System (WCS) standard

```
"Representations of world coordinates in FITS",
Greisen, E.W., & Calabretta, M.R. 2002, A&A, 395, 1061 (WCS Paper I)
"Representations of celestial coordinates in FITS",
Calabretta, M.R., & Greisen, E.W. 2002, A&A, 395, 1077 (WCS Paper II)
```

The transformations are implemented via separate functions, sphx2s() and sphs2x(), for the spherical rotation in each direction.

A utility function, sphdpa(), computes the angular distances and position angles from a given point on the sky to a number of other points. sphpad() does the complementary operation - computes the coordinates of points offset by the given angular distances and position angles from a given point on the sky.

19.17.2   Function Documentation

19.17.2.1   sphx2s()   int sphx2s (
```
            const double eul[5],
            int nphi,
            int ntheta,
            int spt,
            int sxy,
            const double phi[],
            const double theta[],
            double lng[],
            double lat[] )
```

sphx2s() transforms native coordinates of a projection to celestial coordinates.

Parameters

in	*eul*	Euler angles for the transformation:
		• 0: Celestial longitude of the native pole [deg].
		• 1: Celestial colatitude of the native pole, or native colatitude of the celestial pole [deg].
		• 2: Native longitude of the celestial pole [deg].
		• 3: cos(eul[1])
		• 4: sin(eul[1])
in	*nphi,ntheta*	Vector lengths.
in	*spt,sxy*	Vector strides.
in	*phi,theta*	Longitude and latitude in the native coordinate system of the projection [deg].
out	*lng,lat*	Celestial longitude and latitude [deg]. These may refer to the same storage as *phi* and *theta* respectively.

Returns

Status return value:

• 0: Success.

19.17.2.2   sphs2x()   int sphs2x (
        const double *eul[5],*
        int *nlng,*
        int *nlat,*
        int *sll,*
        int *spt,*
        const double *lng[ ],*
        const double *lat[ ],*
        double *phi[ ],*
        double *theta[ ]*)

sphs2x() transforms celestial coordinates to the native coordinates of a projection.

Parameters

in	*eul*	Euler angles for the transformation:
		• 0: Celestial longitude of the native pole [deg].
		• 1: Celestial colatitude of the native pole, or native colatitude of the celestial pole [deg].
		• 2: Native longitude of the celestial pole [deg].
		• 3: cos(eul[1])
		• 4: sin(eul[1])
in	*nlng,nlat*	Vector lengths.
in	*sll,spt*	Vector strides.
in	*lng,lat*	Celestial longitude and latitude [deg].
out	*phi,theta*	Longitude and latitude in the native coordinate system of the projection [deg]. These may refer to the same storage as *lng* and *lat* respectively.

Returns

Status return value:

- 0: Success.

19.17.2.3 sphdpa() `int sphdpa (`
`        int nfield,`
`        double lng0,`
`        double lat0,`
`        const double lng[],`
`        const double lat[],`
`        double dist[],`
`        double pa[] )`

sphdpa() computes the angular distance and generalized position angle (see notes) from a "reference" point to a number of "field" points on the sphere. The points must be specified consistently in any spherical coordinate system.

sphdpa() is complementary to sphpad().

Parameters

in	*nfield*	The number of field points.
in	*lng0,lat0*	Spherical coordinates of the reference point [deg].
in	*lng,lat*	Spherical coordinates of the field points [deg].
out	*dist,pa*	Angular distances and position angles [deg]. These may refer to the same storage as *lng* and *lat* respectively.

Returns

Status return value:

- 0: Success.

Notes:
1. **sphdpa**() uses sphs2x() to rotate coordinates so that the reference point is at the north pole of the new system with the north pole of the old system at zero longitude in the new. The Euler angles required by sphs2x() for this rotation are
```
eul[0] = lng0;
eul[1] = 90.0 - lat0;
eul[2] =  0.0;
```

The angular distance and generalized position angle are readily obtained from the longitude and latitude of the field point in the new system. This applies even if the reference point is at one of the poles, in which case the "position angle" returned is as would be computed for a reference point at $(\alpha_0, +90° - \epsilon)$ or $(\alpha_0, -90° + \epsilon)$, in the limit as ϵ goes to zero.

It is evident that the coordinate system in which the two points are expressed is irrelevant to the determination of the angular separation between the points. However, this is not true of the generalized position angle.

The generalized position angle is here defined as the angle of intersection of the great circle containing the reference and field points with that containing the reference point and the pole. It has its normal meaning when the the reference and field points are specified in equatorial coordinates (right ascension and declination).

Interchanging the reference and field points changes the position angle in a non-intuitive way (because the sum of the angles of a spherical triangle normally exceeds $180°$).

The position angle is undefined if the reference and field points are coincident or antipodal. This may be detected by checking for a distance of $0°$ or $180°$ (within rounding tolerance). **sphdpa**() will return an arbitrary position angle in such circumstances.

19.17.2.4 sphpad() `int sphpad (`
    `int nfield,`
    `double lng0,`
    `double lat0,`
    `const double dist[],`
    `const double pa[],`
    `double lng[],`
    `double lat[] )`

sphpad() computes the coordinates of a set of points that are offset by the specified angular distances and position angles from a given "reference" point on the sky. The distances and position angles must be specified consistently in any spherical coordinate system.

sphpad() is complementary to sphdpa().

Parameters

in	*nfield*	The number of field points.
in	*lng0,lat0*	Spherical coordinates of the reference point [deg].
in	*dist,pa*	Angular distances and position angles [deg].
out	*lng,lat*	Spherical coordinates of the field points [deg]. These may refer to the same storage as *dist* and *pa* respectively.

Returns

 Status return value:

  • 0: Success.

Notes:

1. **sphpad**() is implemented analogously to sphdpa() although using sphx2s() for the inverse transformation. In particular, when the reference point is at one of the poles, "position angle" is interpreted as though the reference point was at $(\alpha_0, +90° - \epsilon)$ or $(\alpha_0, -90° + \epsilon)$, in the limit as ϵ goes to zero.

Applying **sphpad**() with the distances and position angles computed by sphdpa() should return the original field points.

19.18 sph.h

Go to the documentation of this file.
```
1 /*============================================================================
2   WCSLIB 7.11 - an implementation of the FITS WCS standard.
3   Copyright (C) 1995-2022, Mark Calabretta
```

```
4
5   This file is part of WCSLIB.
6
7   WCSLIB is free software: you can redistribute it and/or modify it under the
8   terms of the GNU Lesser General Public License as published by the Free
9   Software Foundation, either version 3 of the License, or (at your option)
10  any later version.
11
12  WCSLIB is distributed in the hope that it will be useful, but WITHOUT ANY
13  WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
14  FOR A PARTICULAR PURPOSE.  See the GNU Lesser General Public License for
15  more details.
16
17  You should have received a copy of the GNU Lesser General Public License
18  along with WCSLIB.  If not, see http://www.gnu.org/licenses.
19
20  Author: Mark Calabretta, Australia Telescope National Facility, CSIRO.
21  http://www.atnf.csiro.au/people/Mark.Calabretta
22  $Id: sph.h,v 7.11 2022/04/26 06:13:52 mcalabre Exp $
23 *=============================================================================
24 *
25 * WCSLIB 7.11 - C routines that implement the FITS World Coordinate System
26 * (WCS) standard.  Refer to the README file provided with WCSLIB for an
27 * overview of the library.
28 *
29 *
30 * Summary of the sph routines
31 * ---------------------------
32 * Routines in this suite implement the spherical coordinate transformations
33 * defined by the FITS World Coordinate System (WCS) standard
34 *
35 =   "Representations of world coordinates in FITS",
36 =   Greisen, E.W., & Calabretta, M.R. 2002, A&A, 395, 1061 (WCS Paper I)
37 =
38 =   "Representations of celestial coordinates in FITS",
39 =   Calabretta, M.R., & Greisen, E.W. 2002, A&A, 395, 1077 (WCS Paper II)
40 *
41 * The transformations are implemented via separate functions, sphx2s() and
42 * sphs2x(), for the spherical rotation in each direction.
43 *
44 * A utility function, sphdpa(), computes the angular distances and position
45 * angles from a given point on the sky to a number of other points.  sphpad()
46 * does the complementary operation - computes the coordinates of points offset
47 * by the given angular distances and position angles from a given point on the
48 * sky.
49 *
50 *
51 * sphx2s() - Rotation in the pixel-to-world direction
52 * ---------------------------------------------------
53 * sphx2s() transforms native coordinates of a projection to celestial
54 * coordinates.
55 *
56 * Given:
57 *   eul       const double[5]
58 *                       Euler angles for the transformation:
59 *                         0: Celestial longitude of the native pole [deg].
60 *                         1: Celestial colatitude of the native pole, or
61 *                            native colatitude of the celestial pole [deg].
62 *                         2: Native longitude of the celestial pole [deg].
63 *                         3: cos(eul[1])
64 *                         4: sin(eul[1])
65 *
66 *   nphi,
67 *   ntheta    int       Vector lengths.
68 *
69 *   spt,sxy   int       Vector strides.
70 *
71 *   phi,theta const double[]
72 *                       Longitude and latitude in the native coordinate
73 *                       system of the projection [deg].
74 *
75 * Returned:
76 *   lng,lat   double[]  Celestial longitude and latitude [deg].  These may
77 *                       refer to the same storage as phi and theta
78 *                       respectively.
79 *
80 * Function return value:
81 *             int       Status return value:
82 *                         0: Success.
83 *
84 *
85 * sphs2x() - Rotation in the world-to-pixel direction
86 * ---------------------------------------------------
87 * sphs2x() transforms celestial coordinates to the native coordinates of a
88 * projection.
89 *
90 * Given:
```

```
 91 *   eul       const double[5]
 92 *                     Euler angles for the transformation:
 93 *                        0: Celestial longitude of the native pole [deg].
 94 *                        1: Celestial colatitude of the native pole, or
 95 *                           native colatitude of the celestial pole [deg].
 96 *                        2: Native longitude of the celestial pole [deg].
 97 *                        3: cos(eul[1])
 98 *                        4: sin(eul[1])
 99 *
100 *   nlng,nlat int       Vector lengths.
101 *
102 *   sll,spt   int       Vector strides.
103 *
104 *   lng,lat   const double[]
105 *                     Celestial longitude and latitude [deg].
106 *
107 * Returned:
108 *   phi,theta double[]  Longitude and latitude in the native coordinate system
109 *                       of the projection [deg].  These may refer to the same
110 *                       storage as lng and lat respectively.
111 *
112 * Function return value:
113 *             int       Status return value:
114 *                         0: Success.
115 *
116 *
117 * sphdpa() - Compute angular distance and position angle
118 * ------------------------------------------------------
119 * sphdpa() computes the angular distance and generalized position angle (see
120 * notes) from a "reference" point to a number of "field" points on the sphere.
121 * The points must be specified consistently in any spherical coordinate
122 * system.
123 *
124 * sphdpa() is complementary to sphpad().
125 *
126 * Given:
127 *   nfield    int       The number of field points.
128 *
129 *   lng0,lat0 double    Spherical coordinates of the reference point [deg].
130 *
131 *   lng,lat   const double[]
132 *                     Spherical coordinates of the field points [deg].
133 *
134 * Returned:
135 *   dist,pa   double[]  Angular distances and position angles [deg].  These
136 *                       may refer to the same storage as lng and lat
137 *                       respectively.
138 *
139 * Function return value:
140 *             int       Status return value:
141 *                         0: Success.
142 *
143 * Notes:
144 *   1. sphdpa() uses sphs2x() to rotate coordinates so that the reference
145 *      point is at the north pole of the new system with the north pole of the
146 *      old system at zero longitude in the new.  The Euler angles required by
147 *      sphs2x() for this rotation are
148 *
149 =        eul[0] = lng0;
150 =        eul[1] = 90.0 - lat0;
151 =        eul[2] =  0.0;
152 *
153 *      The angular distance and generalized position angle are readily
154 *      obtained from the longitude and latitude of the field point in the new
155 *      system.  This applies even if the reference point is at one of the
156 *      poles, in which case the "position angle" returned is as would be
157 *      computed for a reference point at (lng0,+90-epsilon) or
158 *      (lng0,-90+epsilon), in the limit as epsilon goes to zero.
159 *
160 *      It is evident that the coordinate system in which the two points are
161 *      expressed is irrelevant to the determination of the angular separation
162 *      between the points.  However, this is not true of the generalized
163 *      position angle.
164 *
165 *      The generalized position angle is here defined as the angle of
166 *      intersection of the great circle containing the reference and field
167 *      points with that containing the reference point and the pole.  It has
168 *      its normal meaning when the the reference and field points are
169 *      specified in equatorial coordinates (right ascension and declination).
170 *
171 *      Interchanging the reference and field points changes the position angle
172 *      in a non-intuitive way (because the sum of the angles of a spherical
173 *      triangle normally exceeds 180 degrees).
174 *
175 *      The position angle is undefined if the reference and field points are
176 *      coincident or antipodal.  This may be detected by checking for a
177 *      distance of 0 or 180 degrees (within rounding tolerance).  sphdpa()
```

```
178 *         will return an arbitrary position angle in such circumstances.
179 *
180 *
181 * sphpad() - Compute field points offset from a given point
182 * ----------------------------------------------------------
183 * sphpad() computes the coordinates of a set of points that are offset by the
184 * specified angular distances and position angles from a given "reference"
185 * point on the sky.  The distances and position angles must be specified
186 * consistently in any spherical coordinate system.
187 *
188 * sphpad() is complementary to sphdpa().
189 *
190 * Given:
191 *   nfield    int       The number of field points.
192 *
193 *   lng0,lat0 double    Spherical coordinates of the reference point [deg].
194 *
195 *   dist,pa   const double[]
196 *                       Angular distances and position angles [deg].
197 *
198 * Returned:
199 *   lng,lat   double[]  Spherical coordinates of the field points [deg].
200 *                       These may refer to the same storage as dist and pa
201 *                       respectively.
202 *
203 * Function return value:
204 *             int       Status return value:
205 *                         0: Success.
206 *
207 * Notes:
208 *   1: sphpad() is implemented analogously to sphdpa() although using sphx2s()
209 *      for the inverse transformation.  In particular, when the reference
210 *      point is at one of the poles, "position angle" is interpreted as though
211 *      the reference point was at (lng0,+90-epsilon) or (lng0,-90+epsilon), in
212 *      the limit as epsilon goes to zero.
213 *
214 *   Applying sphpad() with the distances and position angles computed by
215 *   sphdpa() should return the original field points.
216 *
217 *===========================================================================*/
218
219 #ifndef WCSLIB_SPH
220 #define WCSLIB_SPH
221
222 #ifdef __cplusplus
223 extern "C" {
224 #endif
225
226
227 int sphx2s(const double eul[5], int nphi, int ntheta, int spt, int sxy,
228            const double phi[], const double theta[],
229            double lng[], double lat[]);
230
231 int sphs2x(const double eul[5], int nlng, int nlat, int sll , int spt,
232            const double lng[], const double lat[],
233            double phi[], double theta[]);
234
235 int sphdpa(int nfield, double lng0, double lat0,
236            const double lng[], const double lat[],
237            double dist[], double pa[]);
238
239 int sphpad(int nfield, double lng0, double lat0,
240            const double dist[], const double pa[],
241            double lng[], double lat[]);
242
243
244 #ifdef __cplusplus
245 }
246 #endif
247
248 #endif // WCSLIB_SPH
```

19.19 spx.h File Reference

Data Structures

- struct spxprm

    Spectral variables and their derivatives.

Macros

- #define SPXLEN (sizeof(struct spxprm)/sizeof(int))

    Size of the spxprm struct in int units.
- #define SPX_ARGS

    For use in declaring spectral conversion function prototypes.

Enumerations

- enum spx_errmsg {
    SPXERR_SUCCESS = 0 , SPXERR_NULL_POINTER = 1 , SPXERR_BAD_SPEC_PARAMS = 2 ,
    SPXERR_BAD_SPEC_VAR = 3 ,
    SPXERR_BAD_INSPEC_COORD = 4 }

Functions

- int specx (const char *type, double spec, double restfrq, double restwav, struct spxprm *specs)

    Spectral cross conversions (scalar).
- int spxperr (const struct spxprm *spx, const char *prefix)

    Print error messages from a spxprm struct.
- int freqafrq (SPX_ARGS)

    Convert frequency to angular frequency (vector).
- int afrqfreq (SPX_ARGS)

    Convert angular frequency to frequency (vector).
- int freqener (SPX_ARGS)

    Convert frequency to photon energy (vector).
- int enerfreq (SPX_ARGS)

    Convert photon energy to frequency (vector).
- int freqwavn (SPX_ARGS)

    Convert frequency to wave number (vector).
- int wavnfreq (SPX_ARGS)

    Convert wave number to frequency (vector).
- int freqwave (SPX_ARGS)

    Convert frequency to vacuum wavelength (vector).
- int wavefreq (SPX_ARGS)

    Convert vacuum wavelength to frequency (vector).
- int freqawav (SPX_ARGS)

    Convert frequency to air wavelength (vector).
- int awavfreq (SPX_ARGS)

    Convert air wavelength to frequency (vector).
- int waveawav (SPX_ARGS)

    Convert vacuum wavelength to air wavelength (vector).
- int awavwave (SPX_ARGS)

    Convert air wavelength to vacuum wavelength (vector).
- int velobeta (SPX_ARGS)

    Convert relativistic velocity to relativistic beta (vector).
- int betavelo (SPX_ARGS)

    Convert relativistic beta to relativistic velocity (vector).
- int freqvelo (SPX_ARGS)

    Convert frequency to relativistic velocity (vector).

- int velofreq (SPX_ARGS)

  Convert relativistic velocity to frequency (vector).

- int freqvrad (SPX_ARGS)

  Convert frequency to radio velocity (vector).

- int vradfreq (SPX_ARGS)

  Convert radio velocity to frequency (vector).

- int wavevelo (SPX_ARGS)

  Conversions between wavelength and velocity types (vector).

- int velowave (SPX_ARGS)

  Convert relativistic velocity to vacuum wavelength (vector).

- int awavvelo (SPX_ARGS)

  Convert air wavelength to relativistic velocity (vector).

- int veloawav (SPX_ARGS)

  Convert relativistic velocity to air wavelength (vector).

- int wavevopt (SPX_ARGS)

  Convert vacuum wavelength to optical velocity (vector).

- int voptwave (SPX_ARGS)

  Convert optical velocity to vacuum wavelength (vector).

- int wavezopt (SPX_ARGS)

  Convert vacuum wavelength to redshift (vector).

- int zoptwave (SPX_ARGS)

  Convert redshift to vacuum wavelength (vector).

Variables

- const char ∗ spx_errmsg [ ]

19.19.1 Detailed Description

Routines in this suite implement the spectral coordinate systems recognized by the FITS World Coordinate System (WCS) standard, as described in

```
"Representations of world coordinates in FITS",
Greisen, E.W., & Calabretta, M.R. 2002, A&A, 395, 1061 (WCS Paper I)
"Representations of spectral coordinates in FITS",
Greisen, E.W., Calabretta, M.R., Valdes, F.G., & Allen, S.L.
2006, A&A, 446, 747 (WCS Paper III)
```

specx() is a scalar routine that, given one spectral variable (e.g. frequency), computes all the others (e.g. wavelength, velocity, etc.) plus the required derivatives of each with respect to the others. The results are returned in the spxprm struct.

spxperr() prints the error message(s) (if any) stored in a spxprm struct.

The remaining routines are all vector conversions from one spectral variable to another. The API of these functions only differ in whether the rest frequency or wavelength need be supplied.

Non-linear:

- freqwave() frequency -> vacuum wavelength

- wavefreq() vacuum wavelength -> frequency

- freqawav() frequency -> air wavelength

- awavfreq() air wavelength -> frequency

- freqvelo() frequency -> relativistic velocity

- velofreq() relativistic velocity -> frequency

- waveawav() vacuum wavelength -> air wavelength

- awavwave() air wavelength -> vacuum wavelength

- wavevelo() vacuum wavelength -> relativistic velocity

- velowave() relativistic velocity -> vacuum wavelength

- awavvelo() air wavelength -> relativistic velocity

- veloawav() relativistic velocity -> air wavelength

Linear:

- freqafrq() frequency -> angular frequency

- afrqfreq() angular frequency -> frequency

- freqener() frequency -> energy

- enerfreq() energy -> frequency

- freqwavn() frequency -> wave number

- wavnfreq() wave number -> frequency

- freqvrad() frequency -> radio velocity

- vradfreq() radio velocity -> frequency

- wavevopt() vacuum wavelength -> optical velocity

- voptwave() optical velocity -> vacuum wavelength

- wavezopt() vacuum wavelength -> redshift

- zoptwave() redshift -> vacuum wavelength

- velobeta() relativistic velocity -> beta ($\beta = v/c$)

- betavelo() beta ($\beta = v/c$) -> relativistic velocity

These are the workhorse routines, to be used for fast transformations. Conversions may be done "in place" by calling the routine with the output vector set to the input.

Air-to-vacuum wavelength conversion:
The air-to-vacuum wavelength conversion in early drafts of WCS Paper III cites Cox (ed., 2000, Allen's Astrophysical Quantities, AIP Press, Springer-Verlag, New York), which itself derives from Edlén (1953, Journal of the Optical Society of America, 43, 339). This is the IAU standard, adopted in 1957 and again in 1991. No more recent IAU resolution replaces this relation, and it is the one used by WCSLIB.

However, the Cox relation was replaced in later drafts of Paper III, and as eventually published, by the IUGG relation (1999, International Union of Geodesy and Geophysics, comptes rendus of the 22nd General Assembly, Birmingham UK, p111). There is a nearly constant ratio between the two, with IUGG/Cox = 1.000015 over most of the range between 200nm and 10,000nm.

The IUGG relation itself is derived from the work of Ciddor (1996, Applied Optics, 35, 1566), which is used directly by the Sloan Digital Sky Survey. It agrees closely with Cox; longwards of 2500nm, the ratio Ciddor/Cox is fixed at 1.000000021, decreasing only slightly, to 1.000000018, at 1000nm.

The Cox, IUGG, and Ciddor relations all accurately provide the wavelength dependence of the air-to-vacuum wavelength conversion. However, for full accuracy, the atmospheric temperature, pressure, and partial pressure of water vapour must be taken into account. These will determine a small, wavelength-independent scale factor and offset, which is not considered by WCS Paper III.

WCS Paper III is also silent on the question of the range of validity of the air-to-vacuum wavelength conversion. Cox's relation would appear to be valid in the range 200nm to 10,000nm. Both the Cox and the Ciddor relations have singularities below 200nm, with Cox's at 156nm and 83nm. WCSLIB checks neither the range of validity, nor for these singularities.

Argument checking:
The input spectral values are only checked for values that would result in floating point exceptions. In particular, negative frequencies and wavelengths are allowed, as are velocities greater than the speed of light. The same is true for the spectral parameters - rest frequency and wavelength.

Accuracy:
No warranty is given for the accuracy of these routines (refer to the copyright notice); intending users must satisfy for themselves their adequacy for the intended purpose. However, closure effectively to within double precision rounding error was demonstrated by test routine tspec.c which accompanies this software.

19.19.2 Macro Definition Documentation

19.19.2.1 SPXLEN `#define SPXLEN (sizeof(struct spxprm)/sizeof(int))`

Size of the spxprm struct in *int* units, used by the Fortran wrappers.

19.19.2.2 SPX_ARGS `#define SPX_ARGS`

Value:
```
double param, int nspec, int instep, int outstep, \
const double inspec[], double outspec[], int stat[]
```

Preprocessor macro used for declaring spectral conversion function prototypes.

19.19.3 Enumeration Type Documentation

19.19.3.1 spx_errmsg `enum spx_errmsg`

Enumerator

SPXERR_SUCCESS	
SPXERR_NULL_POINTER	
SPXERR_BAD_SPEC_PARAMS	
SPXERR_BAD_SPEC_VAR	
SPXERR_BAD_INSPEC_COORD	

19.19.4   Function Documentation

19.19.4.1   specx()   `int specx (`
              `const char * type,`
              `double spec,`
              `double restfrq,`
              `double restwav,`
              `struct spxprm * specs )`

Given one spectral variable **specx**() computes all the others, plus the required derivatives of each with respect to the others.

Parameters

in	type	The type of spectral variable given by spec, **FREQ**, **AFRQ**, **ENER**, **WAVN**, **VRAD**, **WAVE**, **VOPT**, **ZOPT**, **AWAV**, **VELO**, or **BETA** (case sensitive).
in	spec	The spectral variable given, in SI units.
in	restfrq,restwav	Rest frequency [Hz] or rest wavelength in vacuo [m], only one of which need be given. The other should be set to zero. If both are zero, only a subset of the spectral variables can be computed, the remainder are set to zero. Specifically, given one of **FREQ**, **AFRQ**, **ENER**, **WAVN**, **WAVE**, or **AWAV** the others can be computed without knowledge of the rest frequency. Likewise, **VRAD**, **VOPT**, **ZOPT**, **VELO**, and **BETA**.
in,out	specs	Data structure containing all spectral variables and their derivatives, in SI units.

Returns

   Status return value:

   - 0: Success.
   - 1: Null spxprm pointer passed.
   - 2: Invalid spectral parameters.
   - 3: Invalid spectral variable.

   For returns > 1, a detailed error message is set in spxprm::err if enabled, see wcserr_enable().

freqafrq(), afrqfreq(), freqener(), enerfreq(), freqwavn(), wavnfreq(), freqwave(), wavefreq(), freqawav(), awavfreq(), waveawav(), awavwave(), velobeta(), and betavelo() implement vector conversions between wave-like or velocity-like spectral types (i.e. conversions that do not need the rest frequency or wavelength). They all have the same API.

19.19.4.2   spxperr()   `int spxperr (`
              `const struct spxprm * spx,`
              `const char * prefix )`

spxperr() prints the error message(s) (if any) stored in a spxprm struct. If there are no errors then nothing is printed. It uses wcserr_prt(), q.v.

Parameters

in	spx	Spectral variables and their derivatives.
in	prefix	If non-NULL, each output line will be prefixed with this string.

Returns

Status return value:

- 0: Success.
- 1: Null spxprm pointer passed.

19.19.4.3 freqafrq() `int freqafrq (`
`SPX_ARGS )`

freqafrq() converts frequency to angular frequency.

Parameters

in	*param*	Ignored.
in	*nspec*	Vector length.
in	*instep,outstep*	Vector strides.
in	*inspec*	Input spectral variables, in SI units.
out	*outspec*	Output spectral variables, in SI units.
out	*stat*	Status return value for each vector element: • 0: Success. • 1: Invalid value of inspec.

Returns

Status return value:

- 0: Success.
- 2: Invalid spectral parameters.
- 4: One or more of the inspec coordinates were invalid, as indicated by the stat vector.

19.19.4.4 afrqfreq() `int afrqfreq (`
`SPX_ARGS )`

afrqfreq() converts angular frequency to frequency.

See freqafrq() for a description of the API.

19.19.4.5 freqener() `int freqener (`
`SPX_ARGS )`

freqener() converts frequency to photon energy.

See freqafrq() for a description of the API.

19.19.4.6   enerfreq()   `int enerfreq (`
            `SPX_ARGS  )`

enerfreq() converts photon energy to frequency.

See freqafrq() for a description of the API.

19.19.4.7   freqwavn()   `int freqwavn (`
            `SPX_ARGS  )`

freqwavn() converts frequency to wave number.

See freqafrq() for a description of the API.

19.19.4.8   wavnfreq()   `int wavnfreq (`
            `SPX_ARGS  )`

wavnfreq() converts wave number to frequency.

See freqafrq() for a description of the API.

19.19.4.9   freqwave()   `int freqwave (`
            `SPX_ARGS  )`

freqwave() converts frequency to vacuum wavelength.

See freqafrq() for a description of the API.

19.19.4.10   wavefreq()   `int wavefreq (`
            `SPX_ARGS  )`

wavefreq() converts vacuum wavelength to frequency.

See freqafrq() for a description of the API.

19.19.4.11   freqawav()   `int freqawav (`
            `SPX_ARGS  )`

freqawav() converts frequency to air wavelength.

See freqafrq() for a description of the API.

19.19.4.12   awavfreq()   `int awavfreq (`
            `SPX_ARGS  )`

awavfreq() converts air wavelength to frequency.

See freqafrq() for a description of the API.

19.19.4.13 waveawav() `int waveawav (`
`        `[`SPX_ARGS`]` )`

waveawav() converts vacuum wavelength to air wavelength.

See [freqafrq()](#) for a description of the API.

19.19.4.14 awavwave() `int awavwave (`
`        `[`SPX_ARGS`]` )`

awavwave() converts air wavelength to vacuum wavelength.

See [freqafrq()](#) for a description of the API.

19.19.4.15 velobeta() `int velobeta (`
`        `[`SPX_ARGS`]` )`

velobeta() converts relativistic velocity to relativistic beta.

See [freqafrq()](#) for a description of the API.

19.19.4.16 betavelo() `int betavelo (`
`        `[`SPX_ARGS`]` )`

betavelo() converts relativistic beta to relativistic velocity.

See [freqafrq()](#) for a description of the API.

19.19.4.17 freqvelo() `int freqvelo (`
`        `[`SPX_ARGS`]` )`

freqvelo() converts frequency to relativistic velocity.

Parameters

in	*param*	Rest frequency [Hz].
in	*nspec*	Vector length.
in	*instep,outstep*	Vector strides.
in	*inspec*	Input spectral variables, in SI units.
out	*outspec*	Output spectral variables, in SI units.
out	*stat*	Status return value for each vector element: • 0: Success. • 1: Invalid value of inspec.

Returns

Status return value:

- 0: Success.

- 2: Invalid spectral parameters.

- 4: One or more of the inspec coordinates were invalid, as indicated by the stat vector.

19.19.4.18   velofreq()   `int velofreq (`
                `SPX_ARGS  )`

velofreq() converts relativistic velocity to frequency.

See freqvelo() for a description of the API.

19.19.4.19   freqvrad()   `int freqvrad (`
                `SPX_ARGS  )`

freqvrad() converts frequency to radio velocity.

See freqvelo() for a description of the API.

19.19.4.20   vradfreq()   `int vradfreq (`
                `SPX_ARGS  )`

vradfreq() converts radio velocity to frequency.

See freqvelo() for a description of the API.

19.19.4.21   wavevelo()   `int wavevelo (`
                `SPX_ARGS  )`

wavevelo() converts vacuum wavelength to relativistic velocity.

Parameters

in	*param*	Rest wavelength in vacuo [m].
in	*nspec*	Vector length.
in	*instep,outstep*	Vector strides.
in	*inspec*	Input spectral variables, in SI units.
out	*outspec*	Output spectral variables, in SI units.
out	*stat*	Status return value for each vector element: • 0: Success. • 1: Invalid value of inspec.

Returns

Status return value:

- 0: Success.
- 2: Invalid spectral parameters.
- 4: One or more of the inspec coordinates were invalid, as indicated by the stat vector.

19.19.4.22 velowave() `int velowave (`
  `SPX_ARGS )`

velowave() converts relativistic velocity to vacuum wavelength.

See freqvelo() for a description of the API.

19.19.4.23 awavvelo() `int awavvelo (`
  `SPX_ARGS )`

awavvelo() converts air wavelength to relativistic velocity.

See freqvelo() for a description of the API.

19.19.4.24 veloawav() `int veloawav (`
  `SPX_ARGS )`

veloawav() converts relativistic velocity to air wavelength.

See freqvelo() for a description of the API.

19.19.4.25 wavevopt() `int wavevopt (`
  `SPX_ARGS )`

wavevopt() converts vacuum wavelength to optical velocity.

See freqvelo() for a description of the API.

19.19.4.26 voptwave() `int voptwave (`
  `SPX_ARGS )`

voptwave() converts optical velocity to vacuum wavelength.

See freqvelo() for a description of the API.

19.19.4.27 wavezopt() `int wavezopt (`
  `SPX_ARGS )`

wavevopt() converts vacuum wavelength to redshift.

See freqvelo() for a description of the API.

19.19.4.28 zoptwave() `int zoptwave (`
              `SPX_ARGS )`

zoptwave() converts redshift to vacuum wavelength.

See freqvelo() for a description of the API.

19.19.5 Variable Documentation

19.19.5.1 spx_errmsg `const char*` spx_errmsg`[] [extern]`

19.20 spx.h

Go to the documentation of this file.
```
1  /*============================================================================
2    WCSLIB 7.11 - an implementation of the FITS WCS standard.
3    Copyright (C) 1995-2022, Mark Calabretta
4
5    This file is part of WCSLIB.
6
7    WCSLIB is free software: you can redistribute it and/or modify it under the
8    terms of the GNU Lesser General Public License as published by the Free
9    Software Foundation, either version 3 of the License, or (at your option)
10   any later version.
11
12   WCSLIB is distributed in the hope that it will be useful, but WITHOUT ANY
13   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
14   FOR A PARTICULAR PURPOSE.  See the GNU Lesser General Public License for
15   more details.
16
17   You should have received a copy of the GNU Lesser General Public License
18   along with WCSLIB.  If not, see http://www.gnu.org/licenses.
19
20   Author: Mark Calabretta, Australia Telescope National Facility, CSIRO.
21   http://www.atnf.csiro.au/people/Mark.Calabretta
22   $Id: spx.h,v 7.11 2022/04/26 06:13:52 mcalabre Exp $
23  *============================================================================
24  *
25  * WCSLIB 7.11 - C routines that implement the FITS World Coordinate System
26  * (WCS) standard.  Refer to the README file provided with WCSLIB for an
27  * overview of the library.
28  *
29  *
30  * Summary of the spx routines
31  * ---------------------------
32  * Routines in this suite implement the spectral coordinate systems recognized
33  * by the FITS World Coordinate System (WCS) standard, as described in
34  *
35  =   "Representations of world coordinates in FITS",
36  =   Greisen, E.W., & Calabretta, M.R. 2002, A&A, 395, 1061 (WCS Paper I)
37  =
38  =   "Representations of spectral coordinates in FITS",
39  =   Greisen, E.W., Calabretta, M.R., Valdes, F.G., & Allen, S.L.
40  =   2006, A&A, 446, 747 (WCS Paper III)
41  *
42  * specx() is a scalar routine that, given one spectral variable (e.g.
43  * frequency), computes all the others (e.g. wavelength, velocity, etc.) plus
44  * the required derivatives of each with respect to the others.  The results
45  * are returned in the spxprm struct.
46  *
47  * spxperr() prints the error message(s) (if any) stored in a spxprm struct.
48  *
49  * The remaining routines are all vector conversions from one spectral
50  * variable to another.  The API of these functions only differ in whether the
51  * rest frequency or wavelength need be supplied.
52  *
53  * Non-linear:
54  *   - freqwave()    frequency              ->  vacuum wavelength
55  *   - wavefreq()    vacuum wavelength      ->  frequency
56  *
```

```
 57 *    - freqawav()    frequency               -> air wavelength
 58 *    - awavfreq()    air wavelength          -> frequency
 59 *
 60 *    - freqvelo()    frequency               -> relativistic velocity
 61 *    - velofreq()    relativistic velocity   -> frequency
 62 *
 63 *    - waveawav()    vacuum wavelength       -> air wavelength
 64 *    - awavwave()    air wavelength          -> vacuum wavelength
 65 *
 66 *    - wavevelo()    vacuum wavelength       -> relativistic velocity
 67 *    - velowave()    relativistic velocity   -> vacuum wavelength
 68 *
 69 *    - awavvelo()    air wavelength          -> relativistic velocity
 70 *    - veloawav()    relativistic velocity   -> air wavelength
 71 *
 72 * Linear:
 73 *    - freqafrq()    frequency               -> angular frequency
 74 *    - afrqfreq()    angular frequency       -> frequency
 75 *
 76 *    - freqener()    frequency               -> energy
 77 *    - enerfreq()    energy                  -> frequency
 78 *
 79 *    - freqwavn()    frequency               -> wave number
 80 *    - wavnfreq()    wave number             -> frequency
 81 *
 82 *    - freqvrad()    frequency               -> radio velocity
 83 *    - vradfreq()    radio velocity          -> frequency
 84 *
 85 *    - wavevopt()    vacuum wavelength       -> optical velocity
 86 *    - voptwave()    optical velocity        -> vacuum wavelength
 87 *
 88 *    - wavezopt()    vacuum wavelength       -> redshift
 89 *    - zoptwave()    redshift                -> vacuum wavelength
 90 *
 91 *    - velobeta()    relativistic velocity   -> beta (= v/c)
 92 *    - betavelo()    beta (= v/c)            -> relativistic velocity
 93 *
 94 * These are the workhorse routines, to be used for fast transformations.
 95 * Conversions may be done "in place" by calling the routine with the output
 96 * vector set to the input.
 97 *
 98 * Air-to-vacuum wavelength conversion:
 99 * ------------------------------------
100 * The air-to-vacuum wavelength conversion in early drafts of WCS Paper III
101 * cites Cox (ed., 2000, Allen's Astrophysical Quantities, AIP Press,
102 * Springer-Verlag, New York), which itself derives from Edlén (1953, Journal
103 * of the Optical Society of America, 43, 339).  This is the IAU standard,
104 * adopted in 1957 and again in 1991.  No more recent IAU resolution replaces
105 * this relation, and it is the one used by WCSLIB.
106 *
107 * However, the Cox relation was replaced in later drafts of Paper III, and as
108 * eventually published, by the IUGG relation (1999, International Union of
109 * Geodesy and Geophysics, comptes rendus of the 22nd General Assembly,
110 * Birmingham UK, p111).  There is a nearly constant ratio between the two,
111 * with IUGG/Cox = 1.000015 over most of the range between 200nm and 10,000nm.
112 *
113 * The IUGG relation itself is derived from the work of Ciddor (1996, Applied
114 * Optics, 35, 1566), which is used directly by the Sloan Digital Sky Survey.
115 * It agrees closely with Cox; longwards of 2500nm, the ratio Ciddor/Cox is
116 * fixed at 1.000000021, decreasing only slightly, to 1.000000018, at 1000nm.
117 *
118 * The Cox, IUGG, and Ciddor relations all accurately provide the wavelength
119 * dependence of the air-to-vacuum wavelength conversion.  However, for full
120 * accuracy, the atmospheric temperature, pressure, and partial pressure of
121 * water vapour must be taken into account.  These will determine a small,
122 * wavelength-independent scale factor and offset, which is not considered by
123 * WCS Paper III.
124 *
125 * WCS Paper III is also silent on the question of the range of validity of the
126 * air-to-vacuum wavelength conversion.  Cox's relation would appear to be
127 * valid in the range 200nm to 10,000nm.  Both the Cox and the Ciddor relations
128 * have singularities below 200nm, with Cox's at 156nm and 83nm.  WCSLIB checks
129 * neither the range of validity, nor for these singularities.
130 *
131 * Argument checking:
132 * ------------------
133 * The input spectral values are only checked for values that would result
134 * in floating point exceptions.  In particular, negative frequencies and
135 * wavelengths are allowed, as are velocities greater than the speed of
136 * light.  The same is true for the spectral parameters - rest frequency and
137 * wavelength.
138 *
139 * Accuracy:
140 * ---------
141 * No warranty is given for the accuracy of these routines (refer to the
142 * copyright notice); intending users must satisfy for themselves their
143 * adequacy for the intended purpose.  However, closure effectively to within
```

```
144 * double precision rounding error was demonstrated by test routine tspec.c
145 * which accompanies this software.
146 *
147 *
148 * specx() - Spectral cross conversions (scalar)
149 * ---------------------------------------------
150 * Given one spectral variable specx() computes all the others, plus the
151 * required derivatives of each with respect to the others.
152 *
153 * Given:
154 *   type       const char*
155 *                         The type of spectral variable given by spec, FREQ,
156 *                         AFRQ, ENER, WAVN, VRAD, WAVE, VOPT, ZOPT, AWAV, VELO,
157 *                         or BETA (case sensitive).
158 *
159 *   spec       double     The spectral variable given, in SI units.
160 *
161 *   restfrq,
162 *   restwav    double     Rest frequency [Hz] or rest wavelength in vacuo [m],
163 *                         only one of which need be given.  The other should be
164 *                         set to zero.  If both are zero, only a subset of the
165 *                         spectral variables can be computed, the remainder are
166 *                         set to zero.  Specifically, given one of FREQ, AFRQ,
167 *                         ENER, WAVN, WAVE, or AWAV the others can be computed
168 *                         without knowledge of the rest frequency.  Likewise,
169 *                         VRAD, VOPT, ZOPT, VELO, and BETA.
170 *
171 * Given and returned:
172 *   specs      struct spxprm*
173 *                         Data structure containing all spectral variables and
174 *                         their derivatives, in SI units.
175 *
176 * Function return value:
177 *              int        Status return value:
178 *                           0: Success.
179 *                           1: Null spxprm pointer passed.
180 *                           2: Invalid spectral parameters.
181 *                           3: Invalid spectral variable.
182 *
183 *                         For returns > 1, a detailed error message is set in
184 *                         spxprm::err if enabled, see wcserr_enable().
185 *
186 * freqafrq(), afrqfreq(), freqener(), enerfreq(), freqwavn(), wavnfreq(),
187 * freqwave(), wavefreq(), freqawav(), awavfreq(), waveawav(), awavwave(),
188 * velobeta(), and betavelo() implement vector conversions between wave-like
189 * or velocity-like spectral types (i.e. conversions that do not need the rest
190 * frequency or wavelength).  They all have the same API.
191 *
192 *
193 * spxperr() - Print error messages from a spxprm struct
194 * -----------------------------------------------------
195 * spxperr() prints the error message(s) (if any) stored in a spxprm struct.
196 * If there are no errors then nothing is printed.  It uses wcserr_prt(), q.v.
197 *
198 * Given:
199 *   spx        const struct spxprm*
200 *                         Spectral variables and their derivatives.
201 *
202 *   prefix     const char *
203 *                         If non-NULL, each output line will be prefixed with
204 *                         this string.
205 *
206 * Function return value:
207 *              int        Status return value:
208 *                           0: Success.
209 *                           1: Null spxprm pointer passed.
210 *
211 *
212 * freqafrq() - Convert frequency to angular frequency (vector)
213 * -----------------------------------------------------------
214 * freqafrq() converts frequency to angular frequency.
215 *
216 * Given:
217 *   param      double     Ignored.
218 *
219 *   nspec      int        Vector length.
220 *
221 *   instep,
222 *   outstep    int        Vector strides.
223 *
224 *   inspec     const double[]
225 *                         Input spectral variables, in SI units.
226 *
227 * Returned:
228 *   outspec    double[]   Output spectral variables, in SI units.
229 *
230 *   stat       int[]      Status return value for each vector element:
```

```
231 *                          0: Success.
232 *                          1: Invalid value of inspec.
233 *
234 * Function return value:
235 *             int        Status return value:
236 *                          0: Success.
237 *                          2: Invalid spectral parameters.
238 *                          4: One or more of the inspec coordinates were
239 *                             invalid, as indicated by the stat vector.
240 *
241 *
242 * freqvelo(), velofreq(), freqvrad(), and vradfreq() implement vector
243 * conversions between frequency and velocity spectral types.  They all have
244 * the same API.
245 *
246 *
247 * freqvelo() - Convert frequency to relativistic velocity (vector)
248 * -------------------------------------------------------------
249 * freqvelo() converts frequency to relativistic velocity.
250 *
251 * Given:
252 *   param     double    Rest frequency [Hz].
253 *
254 *   nspec     int       Vector length.
255 *
256 *   instep,
257 *   outstep   int       Vector strides.
258 *
259 *   inspec    const double[]
260 *                       Input spectral variables, in SI units.
261 *
262 * Returned:
263 *   outspec   double[]  Output spectral variables, in SI units.
264 *
265 *   stat      int[]     Status return value for each vector element:
266 *                          0: Success.
267 *                          1: Invalid value of inspec.
268 *
269 * Function return value:
270 *             int        Status return value:
271 *                          0: Success.
272 *                          2: Invalid spectral parameters.
273 *                          4: One or more of the inspec coordinates were
274 *                             invalid, as indicated by the stat vector.
275 *
276 *
277 * wavevelo(), velowave(), awavvelo(), veloawav(), wavevopt(), voptwave(),
278 * wavezopt(), and zoptwave() implement vector conversions between wavelength
279 * and velocity spectral types.  They all have the same API.
280 *
281 *
282 * wavevelo() - Conversions between wavelength and velocity types (vector)
283 * ----------------------------------------------------------------------
284 * wavevelo() converts vacuum wavelength to relativistic velocity.
285 *
286 * Given:
287 *   param     double    Rest wavelength in vacuo [m].
288 *
289 *   nspec     int       Vector length.
290 *
291 *   instep,
292 *   outstep   int       Vector strides.
293 *
294 *   inspec    const double[]
295 *                       Input spectral variables, in SI units.
296 *
297 * Returned:
298 *   outspec   double[]  Output spectral variables, in SI units.
299 *
300 *   stat      int[]     Status return value for each vector element:
301 *                          0: Success.
302 *                          1: Invalid value of inspec.
303 *
304 * Function return value:
305 *             int        Status return value:
306 *                          0: Success.
307 *                          2: Invalid spectral parameters.
308 *                          4: One or more of the inspec coordinates were
309 *                             invalid, as indicated by the stat vector.
310 *
311 *
312 * spxprm struct - Spectral variables and their derivatives
313 * --------------------------------------------------------
314 * The spxprm struct contains the value of all spectral variables and their
315 * derivatives.   It is used solely by specx() which constructs it from
316 * information provided via its function arguments.
317 *
```

```
318 * This struct should be considered read-only, no members need ever be set nor
319 * should ever be modified by the user.
320 *
321 *   double restfrq
322 *      (Returned) Rest frequency [Hz].
323 *
324 *   double restwav
325 *      (Returned) Rest wavelength [m].
326 *
327 *   int wavetype
328 *      (Returned) True if wave types have been computed, and ...
329 *
330 *   int velotype
331 *      (Returned) ... true if velocity types have been computed; types are
332 *      defined below.
333 *
334 *      If one or other of spxprm::restfrq and spxprm::restwav is given
335 *      (non-zero) then all spectral variables may be computed.  If both are
336 *      given, restfrq is used.  If restfrq and restwav are both zero, only wave
337 *      characteristic xor velocity type spectral variables may be computed
338 *      depending on the variable given.   These flags indicate what is
339 *      available.
340 *
341 *   double freq
342 *      (Returned) Frequency [Hz] (wavetype).
343 *
344 *   double afrq
345 *      (Returned) Angular frequency [rad/s] (wavetype).
346 *
347 *   double ener
348 *      (Returned) Photon energy [J] (wavetype).
349 *
350 *   double wavn
351 *      (Returned) Wave number [/m] (wavetype).
352 *
353 *   double vrad
354 *      (Returned) Radio velocity [m/s] (velotype).
355 *
356 *   double wave
357 *      (Returned) Vacuum wavelength [m] (wavetype).
358 *
359 *   double vopt
360 *      (Returned) Optical velocity [m/s] (velotype).
361 *
362 *   double zopt
363 *      (Returned) Redshift [dimensionless] (velotype).
364 *
365 *   double awav
366 *      (Returned) Air wavelength [m] (wavetype).
367 *
368 *   double velo
369 *      (Returned) Relativistic velocity [m/s] (velotype).
370 *
371 *   double beta
372 *      (Returned) Relativistic beta [dimensionless] (velotype).
373 *
374 *   double dfreqafrq
375 *      (Returned) Derivative of frequency with respect to angular frequency
376 *      [/rad] (constant, = 1 / 2*pi), and ...
377 *   double dafrqfreq
378 *      (Returned) ... vice versa [rad] (constant, = 2*pi, always available).
379 *
380 *   double dfreqener
381 *      (Returned) Derivative of frequency with respect to photon energy
382 *      [/J/s] (constant, = 1/h), and ...
383 *   double denerfreq
384 *      (Returned) ... vice versa [Js] (constant, = h, Planck's constant,
385 *      always available).
386 *
387 *   double dfreqwavn
388 *      (Returned) Derivative of frequency with respect to wave number [m/s]
389 *      (constant, = c, the speed of light in vacuo), and ...
390 *   double dwavnfreq
391 *      (Returned) ... vice versa [s/m] (constant, = 1/c, always available).
392 *
393 *   double dfreqvrad
394 *      (Returned) Derivative of frequency with respect to radio velocity [/m],
395 *      and ...
396 *   double dvradfreq
397 *      (Returned) ... vice versa [m] (wavetype && velotype).
398 *
399 *   double dfreqwave
400 *      (Returned) Derivative of frequency with respect to vacuum wavelength
401 *      [/m/s], and ...
402 *   double dwavefreq
403 *      (Returned) ... vice versa [m s] (wavetype).
404 *
```

```
405 *    double dfreqawav
406 *       (Returned) Derivative of frequency with respect to air wavelength,
407 *       [/m/s], and ...
408 *    double dawavfreq
409 *       (Returned) ... vice versa [m s] (wavetype).
410 *
411 *    double dfreqvelo
412 *       (Returned) Derivative of frequency with respect to relativistic
413 *       velocity [/m], and ...
414 *    double dvelofreq
415 *       (Returned) ... vice versa [m] (wavetype && velotype).
416 *
417 *    double dwavevopt
418 *       (Returned) Derivative of vacuum wavelength with respect to optical
419 *       velocity [s], and ...
420 *    double dvoptwave
421 *       (Returned) ... vice versa [/s] (wavetype && velotype).
422 *
423 *    double dwavezopt
424 *       (Returned) Derivative of vacuum wavelength with respect to redshift [m],
425 *       and ...
426 *    double dzoptwave
427 *       (Returned) ... vice versa [/m] (wavetype && velotype).
428 *
429 *    double dwaveawav
430 *       (Returned) Derivative of vacuum wavelength with respect to air
431 *       wavelength [dimensionless], and ...
432 *    double dawavwave
433 *       (Returned) ... vice versa [dimensionless] (wavetype).
434 *
435 *    double dwavevelo
436 *       (Returned) Derivative of vacuum wavelength with respect to relativistic
437 *       velocity [s], and ...
438 *    double dvelowave
439 *       (Returned) ... vice versa [/s] (wavetype && velotype).
440 *
441 *    double dawavvelo
442 *       (Returned) Derivative of air wavelength with respect to relativistic
443 *       velocity [s], and ...
444 *    double dveloawav
445 *       (Returned) ... vice versa [/s] (wavetype && velotype).
446 *
447 *    double dvelobeta
448 *       (Returned) Derivative of relativistic velocity with respect to
449 *       relativistic beta [m/s] (constant, = c, the speed of light in vacuo),
450 *       and ...
451 *    double dbetavelo
452 *       (Returned) ... vice versa [s/m] (constant, = 1/c, always available).
453 *
454 *    struct wcserr *err
455 *       (Returned) If enabled, when an error status is returned, this struct
456 *       contains detailed information about the error, see wcserr_enable().
457 *
458 *    void *padding
459 *       (An unused variable inserted for alignment purposes only.)
460 *
461 * Global variable: const char *spx_errmsg[] - Status return messages
462 * ----------------------------------------------------------------
463 * Error messages to match the status value returned from each function.
464 *
465 *=========================================================================*/
466
467 #ifndef WCSLIB_SPEC
468 #define WCSLIB_SPEC
469
470 #ifdef __cplusplus
471 extern "C" {
472 #endif
473
474 extern const char *spx_errmsg[];
475
476 enum spx_errmsg {
477   SPXERR_SUCCESS        = 0,    // Success.
478   SPXERR_NULL_POINTER   = 1,       // Null spxprm pointer passed.
479   SPXERR_BAD_SPEC_PARAMS = 2,    // Invalid spectral parameters.
480   SPXERR_BAD_SPEC_VAR   = 3,       // Invalid spectral variable.
481   SPXERR_BAD_INSPEC_COORD = 4   // One or more of the inspec coordinates were
482                                 // invalid.
483 };
484
485 struct spxprm {
486   double restfrq, restwav;       // Rest frequency [Hz] and wavelength [m].
487
488   int wavetype, velotype;       // True if wave/velocity types have been
489                                 // computed; types are defined below.
490
491   // Spectral variables computed by specx().
```

```
492  //---------------------------------------------------------------------
493  double freq,                          // wavetype: Frequency [Hz].
494         afrq,                          // wavetype: Angular frequency [rad/s].
495         ener,                          // wavetype: Photon energy [J].
496         wavn,                          // wavetype: Wave number [/m].
497         vrad,                          // velotype: Radio velocity [m/s].
498         wave,                          // wavetype: Vacuum wavelength [m].
499         vopt,                          // velotype: Optical velocity [m/s].
500         zopt,                          // velotype: Redshift.
501         awav,                          // wavetype: Air wavelength [m].
502         velo,                          // velotype: Relativistic velocity [m/s].
503         beta;                          // velotype: Relativistic beta.
504
505  // Derivatives of spectral variables computed by specx().
506  //---------------------------------------------------------------------
507  double dfreqafrq, dafrqfreq,          // Constant, always available.
508         dfreqener, denerfreq,          // Constant, always available.
509         dfreqwavn, dwavnfreq,          // Constant, always available.
510         dfreqvrad, dvradfreq,          // wavetype && velotype.
511         dfreqwave, dwavefreq,          // wavetype.
512         dfreqawav, dawavfreq,          // wavetype.
513         dfreqvelo, dvelofreq,          // wavetype && velotype.
514         dwavevopt, dvoptwave,          // wavetype && velotype.
515         dwavezopt, dzoptwave,          // wavetype && velotype.
516         dwaveawav, dawavwave,          // wavetype.
517         dwavevelo, dvelowave,          // wavetype && velotype.
518         dawavvelo, dveloawav,          // wavetype && velotype.
519         dvelobeta, dbetavelo;          // Constant, always available.
520
521  // Error handling
522  //---------------------------------------------------------------------
523  struct wcserr *err;
524
525  // Private
526  //---------------------------------------------------------------------
527  void   *padding;                      // (Dummy inserted for alignment purposes.)
528 };
529
530 // Size of the spxprm struct in int units, used by the Fortran wrappers.
531 #define SPXLEN (sizeof(struct spxprm)/sizeof(int))
532
533
534 int specx(const char *type, double spec, double restfrq, double restwav,
535           struct spxprm *specs);
536
537 int spxperr(const struct spxprm *spx, const char *prefix);
538
539 // For use in declaring function prototypes, e.g. in spcprm.
540 #define SPX_ARGS double param, int nspec, int instep, int outstep, \
541     const double inspec[], double outspec[], int stat[]
542
543 int freqafrq(SPX_ARGS);
544 int afrqfreq(SPX_ARGS);
545
546 int freqener(SPX_ARGS);
547 int enerfreq(SPX_ARGS);
548
549 int freqwavn(SPX_ARGS);
550 int wavnfreq(SPX_ARGS);
551
552 int freqwave(SPX_ARGS);
553 int wavefreq(SPX_ARGS);
554
555 int freqawav(SPX_ARGS);
556 int awavfreq(SPX_ARGS);
557
558 int waveawav(SPX_ARGS);
559 int awavwave(SPX_ARGS);
560
561 int velobeta(SPX_ARGS);
562 int betavelo(SPX_ARGS);
563
564
565 int freqvelo(SPX_ARGS);
566 int velofreq(SPX_ARGS);
567
568 int freqvrad(SPX_ARGS);
569 int vradfreq(SPX_ARGS);
570
571
572 int wavevelo(SPX_ARGS);
573 int velowave(SPX_ARGS);
574
575 int awavvelo(SPX_ARGS);
576 int veloawav(SPX_ARGS);
577
578 int wavevopt(SPX_ARGS);
```

```
579 int voptwave(SPX_ARGS);
580
581 int wavezopt(SPX_ARGS);
582 int zoptwave(SPX_ARGS);
583
584
585 #ifdef __cplusplus
586 }
587 #endif
588
589 #endif // WCSLIB_SPEC
```

19.21 tab.h File Reference

Data Structures

- struct tabprm

    Tabular transformation parameters.

Macros

- #define TABLEN (sizeof(struct tabprm)/sizeof(int))

    Size of the tabprm struct in int units.

- #define tabini_errmsg tab_errmsg

    Deprecated.

- #define tabcpy_errmsg tab_errmsg

    Deprecated.

- #define tabfree_errmsg tab_errmsg

    Deprecated.

- #define tabprt_errmsg tab_errmsg

    Deprecated.

- #define tabset_errmsg tab_errmsg

    Deprecated.

- #define tabx2s_errmsg tab_errmsg

    Deprecated.

- #define tabs2x_errmsg tab_errmsg

    Deprecated.

Enumerations

- enum tab_errmsg_enum {
  TABERR_SUCCESS = 0 , TABERR_NULL_POINTER = 1 , TABERR_MEMORY = 2 , TABERR_BAD_PARAMS
  = 3 ,
  TABERR_BAD_X = 4 , TABERR_BAD_WORLD = 5 }

Functions

- int tabini (int alloc, int M, const int K[ ], struct tabprm ∗tab)

  Default constructor for the tabprm struct.
- int tabmem (struct tabprm ∗tab)

  Acquire tabular memory.
- int tabcpy (int alloc, const struct tabprm ∗tabsrc, struct tabprm ∗tabdst)

  Copy routine for the tabprm struct.
- int tabcmp (int cmp, double tol, const struct tabprm ∗tab1, const struct tabprm ∗tab2, int ∗equal)

  Compare two tabprm structs for equality.
- int tabfree (struct tabprm ∗tab)

  Destructor for the tabprm struct.
- int tabsize (const struct tabprm ∗tab, int size[2])

  Compute the size of a tabprm struct.
- int tabprt (const struct tabprm ∗tab)

  Print routine for the tabprm struct.
- int tabperr (const struct tabprm ∗tab, const char ∗prefix)

  Print error messages from a tabprm struct.
- int tabset (struct tabprm ∗tab)

  Setup routine for the tabprm struct.
- int tabx2s (struct tabprm ∗tab, int ncoord, int nelem, const double x[ ], double world[ ], int stat[ ])

  Pixel-to-world transformation.
- int tabs2x (struct tabprm ∗tab, int ncoord, int nelem, const double world[ ], double x[ ], int stat[ ])

  World-to-pixel transformation.

Variables

- const char ∗ tab_errmsg [ ]

  Status return messages.

19.21.1 Detailed Description

Routines in this suite implement the part of the FITS World Coordinate System (WCS) standard that deals with tabular coordinates, i.e. coordinates that are defined via a lookup table, as described in

```
"Representations of world coordinates in FITS",
Greisen, E.W., & Calabretta, M.R. 2002, A&A, 395, 1061 (WCS Paper I)
"Representations of spectral coordinates in FITS",
Greisen, E.W., Calabretta, M.R., Valdes, F.G., & Allen, S.L.
2006, A&A, 446, 747 (WCS Paper III)
```

These routines define methods to be used for computing tabular world coordinates from intermediate world coordinates (a linear transformation of image pixel coordinates), and vice versa. They are based on the tabprm struct which contains all information needed for the computations. The struct contains some members that must be set by the user, and others that are maintained by these routines, somewhat like a C++ class but with no encapsulation.

tabini(), tabmem(), tabcpy(), and tabfree() are provided to manage the tabprm struct, tabsize() computes its total size including allocated memory, and tabprt() prints its contents.

tabperr() prints the error message(s) (if any) stored in a tabprm struct.

A setup routine, tabset(), computes intermediate values in the tabprm struct from parameters in it that were supplied by the user. The struct always needs to be set up by tabset() but it need not be called explicitly - refer to the explanation of tabprm::flag.

tabx2s() and tabs2x() implement the WCS tabular coordinate transformations.

Accuracy:

No warranty is given for the accuracy of these routines (refer to the copyright notice); intending users must satisfy for themselves their adequacy for the intended purpose. However, closure effectively to within double precision rounding error was demonstrated by test routine ttab.c which accompanies this software.

19.21.2 Macro Definition Documentation

19.21.2.1 TABLEN `#define TABLEN (sizeof(struct tabprm)/sizeof(int))`

Size of the tabprm struct in *int* units, used by the Fortran wrappers.

19.21.2.2 tabini_errmsg `#define tabini_errmsg tab_errmsg`

Deprecated Added for backwards compatibility, use tab_errmsg directly now instead.

19.21.2.3 tabcpy_errmsg `#define tabcpy_errmsg tab_errmsg`

Deprecated Added for backwards compatibility, use tab_errmsg directly now instead.

19.21.2.4 tabfree_errmsg `#define tabfree_errmsg tab_errmsg`

Deprecated Added for backwards compatibility, use tab_errmsg directly now instead.

19.21.2.5 tabprt_errmsg `#define tabprt_errmsg tab_errmsg`

Deprecated Added for backwards compatibility, use tab_errmsg directly now instead.

19.21.2.6 tabset_errmsg `#define tabset_errmsg tab_errmsg`

Deprecated Added for backwards compatibility, use tab_errmsg directly now instead.

19.21.2.7 tabx2s_errmsg `#define tabx2s_errmsg tab_errmsg`

Deprecated Added for backwards compatibility, use tab_errmsg directly now instead.

19.21.2.8 tabs2x_errmsg `#define tabs2x_errmsg tab_errmsg`

Deprecated Added for backwards compatibility, use tab_errmsg directly now instead.

19.21.3 Enumeration Type Documentation

19.21.3.1 tab_errmsg_enum `enum tab_errmsg_enum`

Enumerator

TABERR_SUCCESS	
TABERR_NULL_POINTER	
TABERR_MEMORY	
TABERR_BAD_PARAMS	
TABERR_BAD_X	
TABERR_BAD_WORLD	

19.21.4   Function Documentation

19.21.4.1   tabini()   `int tabini (`
            `int alloc,`
            `int M,`
            `const int K[ ],`
            `struct tabprm * tab )`

tabini() allocates memory for arrays in a tabprm struct and sets all members of the struct to default values.

PLEASE NOTE: every tabprm struct should be initialized by **tabini**(), possibly repeatedly. On the first invokation, and only the first invokation, the flag member of the tabprm struct must be set to -1 to initialize memory management, regardless of whether **tabini**() will actually be used to allocate memory.

Parameters

in	*alloc*	If true, allocate memory unconditionally for arrays in the tabprm struct.
		If false, it is assumed that pointers to these arrays have been set by the user except if they are null pointers in which case memory will be allocated for them regardless. (In other words, setting alloc true saves having to initalize these pointers to zero.)
in	*M*	The number of tabular coordinate axes.
in	*K*	Vector of length M whose elements $(K_1, K_2, ...K_M)$ record the lengths of the axes of the coordinate array and of each indexing vector. M and K[] are used to determine the length of the various tabprm arrays and therefore the amount of memory to allocate for them. Their values are copied into the tabprm struct.
		It is permissible to set K (i.e. the address of the array) to zero which has the same effect as setting each element of K[] to zero. In this case no memory will be allocated for the index vectors or coordinate array in the tabprm struct. These together with the K vector must be set separately before calling tabset().
in,out	*tab*	Tabular transformation parameters. Note that, in order to initialize memory management tabprm::flag should be set to -1 when tab is initialized for the first time (memory leaks may result if it had already been initialized).

Returns

> Status return value:
>
> - 0: Success.
>
> - 1: Null tabprm pointer passed.
>
> - 2: Memory allocation failed.

- 3: Invalid tabular parameters.

For returns > 1, a detailed error message is set in tabprm::err if enabled, see wcserr_enable().

19.21.4.2 tabmem() `int tabmem (`
          `struct tabprm * tab )`

tabmem() takes control of memory allocated by the user for arrays in the tabprm struct.

Parameters

in,out	*tab*	Tabular transformation parameters.

Returns

Status return value:

- 0: Success.
- 1: Null tabprm pointer passed.
- 2: Memory allocation failed.

For returns > 1, a detailed error message is set in tabprm::err if enabled, see wcserr_enable().

19.21.4.3 tabcpy() `int tabcpy (`
          `int alloc,`
          `const struct tabprm * tabsrc,`
          `struct tabprm * tabdst )`

tabcpy() does a deep copy of one tabprm struct to another, using tabini() to allocate memory for its arrays if required. Only the "information to be provided" part of the struct is copied; a call to tabset() is required to set up the remainder.

Parameters

in	*alloc*	If true, allocate memory unconditionally for arrays in the tabprm struct. If false, it is assumed that pointers to these arrays have been set by the user except if they are null pointers in which case memory will be allocated for them regardless. (In other words, setting alloc true saves having to initalize these pointers to zero.)
in	*tabsrc*	Struct to copy from.
in,out	*tabdst*	Struct to copy to. tabprm::flag should be set to -1 if tabdst was not previously initialized (memory leaks may result if it was previously initialized).

Returns

Status return value:

- 0: Success.
- 1: Null tabprm pointer passed.

- 2: Memory allocation failed.

For returns > 1, a detailed error message is set in tabprm::err (associated with tabdst) if enabled, see
wcserr_enable().

19.21.4.4   tabcmp()   int tabcmp (
            int *cmp,*
            double *tol,*
            const struct tabprm * *tab1,*
            const struct tabprm * *tab2,*
            int * *equal*)

tabcmp() compares two tabprm structs for equality.

Parameters

in	*cmp*	A bit field controlling the strictness of the comparison. At present, this value must always be 0, indicating a strict comparison. In the future, other options may be added.
in	*tol*	Tolerance for comparison of floating-point values. For example, for tol == 1e-6, all floating-point values in the structs must be equal to the first 6 decimal places. A value of 0 implies exact equality.
in	*tab1*	The first tabprm struct to compare.
in	*tab2*	The second tabprm struct to compare.
out	*equal*	Non-zero when the given structs are equal.

Returns

Status return value:

- 0: Success.
- 1: Null pointer passed.

19.21.4.5   tabfree()   int tabfree (
            struct tabprm * *tab*)

tabfree() frees memory allocated for the tabprm arrays by tabini(). tabini() records the memory it allocates and
tabfree() will only attempt to free this.

PLEASE NOTE: tabfree() must not be invoked on a tabprm struct that was not initialized by tabini().

Parameters

out	*tab*	Coordinate transformation parameters.

Returns

Status return value:

- 0: Success.

- 1: Null tabprm pointer passed.

19.21.4.6 tabsize() `int tabsize (`
          `const struct tabprm * tab,`
          `int size[2] )`

tabsize() computes the full size of a tabprm struct, including allocated memory.

Parameters

in	*tab*	Tabular transformation parameters.
		If NULL, the base size of the struct and the allocated size are both set to zero.
out	*sizes*	The first element is the base size of the struct as returned by sizeof(struct tabprm). The second element is the total allocated size, in bytes, assuming that the allocation was done by tabini(). This figure includes memory allocated for the constituent struct, tabprm::err. It is not an error for the struct not to have been set up via tabset(), which normally results in additional memory allocation.

Returns

Status return value:

- 0: Success.

19.21.4.7 tabprt() `int tabprt (`
          `const struct tabprm * tab )`

tabprt() prints the contents of a tabprm struct using wcsprintf(). Mainly intended for diagnostic purposes.

Parameters

in	*tab*	Tabular transformation parameters.

Returns

Status return value:

- 0: Success.

- 1: Null tabprm pointer passed.

19.21.4.8 tabperr() `int tabperr (`
          `const struct tabprm * tab,`
          `const char * prefix )`

tabperr() prints the error message(s) (if any) stored in a tabprm struct. If there are no errors then nothing is printed. It uses wcserr_prt(), q.v.

Parameters

in	*tab*	Tabular transformation parameters.
in	*prefix*	If non-NULL, each output line will be prefixed with this string.

Returns

Status return value:

- 0: Success.
- 1: Null tabprm pointer passed.

19.21.4.9 **tabset()** `int tabset (`
`            struct tabprm * tab )`

tabset() allocates memory for work arrays in the tabprm struct and sets up the struct according to information supplied within it.

Note that this routine need not be called directly; it will be invoked by tabx2s() and tabs2x() if tabprm::flag is anything other than a predefined magic value.

Parameters

in,out	*tab*	Tabular transformation parameters.

Returns

Status return value:

- 0: Success.
- 1: Null tabprm pointer passed.
- 3: Invalid tabular parameters.

For returns > 1, a detailed error message is set in tabprm::err if enabled, see wcserr_enable().

19.21.4.10 **tabx2s()** `int tabx2s (`
`            struct tabprm * tab,`
`            int ncoord,`
`            int nelem,`
`            const double x[],`
`            double world[],`
`            int stat[] )`

tabx2s() transforms intermediate world coordinates to world coordinates using coordinate lookup.

Parameters

in,out	*tab*	Tabular transformation parameters.
in	*ncoord,nelem*	The number of coordinates, each of vector length nelem.
in	*x*	Array of intermediate world coordinates, SI units.
out	*world*	Array of world coordinates, in SI units.
out	*stat*	Status return value status for each coordinate: • 0: Success. • 1: Invalid intermediate world coordinate.

Returns

Status return value:

- 0: Success.
- 1: Null tabprm pointer passed.
- 3: Invalid tabular parameters.
- 4: One or more of the x coordinates were invalid, as indicated by the stat vector.

For returns > 1, a detailed error message is set in tabprm::err if enabled, see wcserr_enable().

19.21.4.11 tabs2x() `int tabs2x (`
`        struct tabprm * tab,`
`        int ncoord,`
`        int nelem,`
`        const double world[],`
`        double x[],`
`        int stat[] )`

tabs2x() transforms world coordinates to intermediate world coordinates.

Parameters

in,out	*tab*	Tabular transformation parameters.
in	*ncoord,nelem*	The number of coordinates, each of vector length nelem.
in	*world*	Array of world coordinates, in SI units.
out	*x*	Array of intermediate world coordinates, SI units.
out	*stat*	Status return value status for each vector element: • 0: Success. • 1: Invalid world coordinate.

Returns

Status return value:

- 0: Success.

- 1: Null tabprm pointer passed.

- 3: Invalid tabular parameters.

- 5: One or more of the world coordinates were invalid, as indicated by the stat vector.

For returns > 1, a detailed error message is set in tabprm::err if enabled, see wcserr_enable().

19.21.5 Variable Documentation

19.21.5.1 tab_errmsg `const char * tab_errmsg[]` `[extern]`

Error messages to match the status value returned from each function.

19.22 tab.h

Go to the documentation of this file.
```
1  /*============================================================================
2    WCSLIB 7.11 - an implementation of the FITS WCS standard.
3    Copyright (C) 1995-2022, Mark Calabretta
4
5    This file is part of WCSLIB.
6
7    WCSLIB is free software: you can redistribute it and/or modify it under the
8    terms of the GNU Lesser General Public License as published by the Free
9    Software Foundation, either version 3 of the License, or (at your option)
10   any later version.
11
12   WCSLIB is distributed in the hope that it will be useful, but WITHOUT ANY
13   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
14   FOR A PARTICULAR PURPOSE.  See the GNU Lesser General Public License for
15   more details.
16
17   You should have received a copy of the GNU Lesser General Public License
18   along with WCSLIB.  If not, see http://www.gnu.org/licenses.
19
20   Author: Mark Calabretta, Australia Telescope National Facility, CSIRO.
21   http://www.atnf.csiro.au/people/Mark.Calabretta
22   $Id: tab.h,v 7.11 2022/04/26 06:13:52 mcalabre Exp $
23  *============================================================================
24  *
25  * WCSLIB 7.11 - C routines that implement the FITS World Coordinate System
26  * (WCS) standard.  Refer to the README file provided with WCSLIB for an
27  * overview of the library.
28  *
29  *
30  * Summary of the tab routines
31  * ---------------------------
32  * Routines in this suite implement the part of the FITS World Coordinate
33  * System (WCS) standard that deals with tabular coordinates, i.e. coordinates
34  * that are defined via a lookup table, as described in
35  *
36  =   "Representations of world coordinates in FITS",
37  =   Greisen, E.W., & Calabretta, M.R. 2002, A&A, 395, 1061 (WCS Paper I)
38  =
39  =   "Representations of spectral coordinates in FITS",
40  =   Greisen, E.W., Calabretta, M.R., Valdes, F.G., & Allen, S.L.
41  =   2006, A&A, 446, 747 (WCS Paper III)
42  *
43  * These routines define methods to be used for computing tabular world
44  * coordinates from intermediate world coordinates (a linear transformation
45  * of image pixel coordinates), and vice versa.  They are based on the tabprm
46  * struct which contains all information needed for the computations.  The
47  * struct contains some members that must be set by the user, and others that
48  * are maintained by these routines, somewhat like a C++ class but with no
49  * encapsulation.
50  *
51  * tabini(), tabmem(), tabcpy(), and tabfree() are provided to manage the
52  * tabprm struct, tabsize() computes its total size including allocated memory,
53  * and tabprt() prints its contents.
54  *
```

```
55 * tabperr() prints the error message(s) (if any) stored in a tabprm struct.
56 *
57 * A setup routine, tabset(), computes intermediate values in the tabprm struct
58 * from parameters in it that were supplied by the user.  The struct always
59 * needs to be set up by tabset() but it need not be called explicitly - refer
60 * to the explanation of tabprm::flag.
61 *
62 * tabx2s() and tabs2x() implement the WCS tabular coordinate transformations.
63 *
64 * Accuracy:
65 * ---------
66 * No warranty is given for the accuracy of these routines (refer to the
67 * copyright notice); intending users must satisfy for themselves their
68 * adequacy for the intended purpose.  However, closure effectively to within
69 * double precision rounding error was demonstrated by test routine ttab.c
70 * which accompanies this software.
71 *
72 *
73 * tabini() - Default constructor for the tabprm struct
74 * ----------------------------------------------------
75 * tabini() allocates memory for arrays in a tabprm struct and sets all members
76 * of the struct to default values.
77 *
78 * PLEASE NOTE: every tabprm struct should be initialized by tabini(), possibly
79 * repeatedly.  On the first invokation, and only the first invokation, the
80 * flag member of the tabprm struct must be set to -1 to initialize memory
81 * management, regardless of whether tabini() will actually be used to allocate
82 * memory.
83 *
84 * Given:
85 *   alloc     int        If true, allocate memory unconditionally for arrays in
86 *                        the tabprm struct.
87 *
88 *                        If false, it is assumed that pointers to these arrays
89 *                        have been set by the user except if they are null
90 *                        pointers in which case memory will be allocated for
91 *                        them regardless.  (In other words, setting alloc true
92 *                        saves having to initalize these pointers to zero.)
93 *
94 *   M         int        The number of tabular coordinate axes.
95 *
96 *   K         const int[]
97 *                        Vector of length M whose elements (K_1, K_2,... K_M)
98 *                        record the lengths of the axes of the coordinate array
99 *                        and of each indexing vector.  M and K[] are used to
100 *                        determine the length of the various tabprm arrays and
101 *                        therefore the amount of memory to allocate for them.
102 *                        Their values are copied into the tabprm struct.
103 *
104 *                        It is permissible to set K (i.e. the address of the
105 *                        array) to zero which has the same effect as setting
106 *                        each element of K[] to zero.  In this case no memory
107 *                        will be allocated for the index vectors or coordinate
108 *                        array in the tabprm struct.  These together with the
109 *                        K vector must be set separately before calling
110 *                        tabset().
111 *
112 * Given and returned:
113 *   tab       struct tabprm*
114 *                        Tabular transformation parameters.  Note that, in
115 *                        order to initialize memory management tabprm::flag
116 *                        should be set to -1 when tab is initialized for the
117 *                        first time (memory leaks may result if it had already
118 *                        been initialized).
119 *
120 * Function return value:
121 *             int        Status return value:
122 *                          0: Success.
123 *                          1: Null tabprm pointer passed.
124 *                          2: Memory allocation failed.
125 *                          3: Invalid tabular parameters.
126 *
127 *                        For returns > 1, a detailed error message is set in
128 *                        tabprm::err if enabled, see wcserr_enable().
129 *
130 *
131 * tabmem() - Acquire tabular memory
132 * ---------------------------------
133 * tabmem() takes control of memory allocated by the user for arrays in the
134 * tabprm struct.
135 *
136 * Given and returned:
137 *   tab       struct tabprm*
138 *                        Tabular transformation parameters.
139 *
140 * Function return value:
141 *             int        Status return value:
```

```
142 *                            0: Success.
143 *                            1: Null tabprm pointer passed.
144 *                            2: Memory allocation failed.
145 *
146 *                      For returns > 1, a detailed error message is set in
147 *                      tabprm::err if enabled, see wcserr_enable().
148 *
149 *
150 * tabcpy() - Copy routine for the tabprm struct
151 * ---------------------------------------------
152 * tabcpy() does a deep copy of one tabprm struct to another, using tabini() to
153 * allocate memory for its arrays if required.  Only the "information to be
154 * provided" part of the struct is copied; a call to tabset() is required to
155 * set up the remainder.
156 *
157 * Given:
158 *   alloc     int       If true, allocate memory unconditionally for arrays in
159 *                       the tabprm struct.
160 *
161 *                       If false, it is assumed that pointers to these arrays
162 *                       have been set by the user except if they are null
163 *                       pointers in which case memory will be allocated for
164 *                       them regardless.  (In other words, setting alloc true
165 *                       saves having to initalize these pointers to zero.)
166 *
167 *   tabsrc    const struct tabprm*
168 *                       Struct to copy from.
169 *
170 * Given and returned:
171 *   tabdst    struct tabprm*
172 *                       Struct to copy to.  tabprm::flag should be set to -1
173 *                       if tabdst was not previously initialized (memory leaks
174 *                       may result if it was previously initialized).
175 *
176 * Function return value:
177 *             int       Status return value:
178 *                            0: Success.
179 *                            1: Null tabprm pointer passed.
180 *                            2: Memory allocation failed.
181 *
182 *                       For returns > 1, a detailed error message is set in
183 *                       tabprm::err (associated with tabdst) if enabled, see
184 *                       wcserr_enable().
185 *
186 *
187 * tabcmp() - Compare two tabprm structs for equality
188 * --------------------------------------------------
189 * tabcmp() compares two tabprm structs for equality.
190 *
191 * Given:
192 *   cmp       int       A bit field controlling the strictness of the
193 *                       comparison.  At present, this value must always be 0,
194 *                       indicating a strict comparison.  In the future, other
195 *                       options may be added.
196 *
197 *   tol       double    Tolerance for comparison of floating-point values.
198 *                       For example, for tol == 1e-6, all floating-point
199 *                       values in the structs must be equal to the first 6
200 *                       decimal places.  A value of 0 implies exact equality.
201 *
202 *   tab1      const struct tabprm*
203 *                       The first tabprm struct to compare.
204 *
205 *   tab2      const struct tabprm*
206 *                       The second tabprm struct to compare.
207 *
208 * Returned"
209 *   equal     int*      Non-zero when the given structs are equal.
210 *
211 * Function return value:
212 *             int       Status return value:
213 *                            0: Success.
214 *                            1: Null pointer passed.
215 *
216 *
217 * tabfree() - Destructor for the tabprm struct
218 * --------------------------------------------
219 * tabfree() frees memory allocated for the tabprm arrays by tabini().
220 * tabini() records the memory it allocates and tabfree() will only attempt to
221 * free this.
222 *
223 * PLEASE NOTE: tabfree() must not be invoked on a tabprm struct that was not
224 * initialized by tabini().
225 *
226 * Returned:
227 *   tab       struct tabprm*
228 *                       Coordinate transformation parameters.
```

```
229 *
230 * Function return value:
231 *            int        Status return value:
232 *                          0: Success.
233 *                          1: Null tabprm pointer passed.
234 *
235 *
236 * tabsize() - Compute the size of a tabprm struct
237 * -----------------------------------------------
238 * tabsize() computes the full size of a tabprm struct, including allocated
239 * memory.
240 *
241 * Given:
242 *   tab        const struct tabprm*
243 *                          Tabular transformation parameters.
244 *
245 *                          If NULL, the base size of the struct and the allocated
246 *                          size are both set to zero.
247 *
248 * Returned:
249 *   sizes      int[2]     The first element is the base size of the struct as
250 *                          returned by sizeof(struct tabprm).  The second element
251 *                          is the total allocated size, in bytes, assuming that
252 *                          the allocation was done by tabini().  This figure
253 *                          includes memory allocated for the constituent struct,
254 *                          tabprm::err.
255 *
256 *                          It is not an error for the struct not to have been set
257 *                          up via tabset(), which normally results in additional
258 *                          memory allocation.
259 *
260 * Function return value:
261 *            int        Status return value:
262 *                          0: Success.
263 *
264 *
265 * tabprt() - Print routine for the tabprm struct
266 * ----------------------------------------------
267 * tabprt() prints the contents of a tabprm struct using wcsprintf().  Mainly
268 * intended for diagnostic purposes.
269 *
270 * Given:
271 *   tab        const struct tabprm*
272 *                          Tabular transformation parameters.
273 *
274 * Function return value:
275 *            int        Status return value:
276 *                          0: Success.
277 *                          1: Null tabprm pointer passed.
278 *
279 *
280 * tabperr() - Print error messages from a tabprm struct
281 * -----------------------------------------------------
282 * tabperr() prints the error message(s) (if any) stored in a tabprm struct.
283 * If there are no errors then nothing is printed.  It uses wcserr_prt(), q.v.
284 *
285 * Given:
286 *   tab        const struct tabprm*
287 *                          Tabular transformation parameters.
288 *
289 *   prefix     const char *
290 *                          If non-NULL, each output line will be prefixed with
291 *                          this string.
292 *
293 * Function return value:
294 *            int        Status return value:
295 *                          0: Success.
296 *                          1: Null tabprm pointer passed.
297 *
298 *
299 * tabset() - Setup routine for the tabprm struct
300 * ----------------------------------------------
301 * tabset() allocates memory for work arrays in the tabprm struct and sets up
302 * the struct according to information supplied within it.
303 *
304 * Note that this routine need not be called directly; it will be invoked by
305 * tabx2s() and tabs2x() if tabprm::flag is anything other than a predefined
306 * magic value.
307 *
308 * Given and returned:
309 *   tab        struct tabprm*
310 *                          Tabular transformation parameters.
311 *
312 * Function return value:
313 *            int        Status return value:
314 *                          0: Success.
315 *                          1: Null tabprm pointer passed.
```

```
316 *                        3: Invalid tabular parameters.
317 *
318 *                    For returns > 1, a detailed error message is set in
319 *                    tabprm::err if enabled, see wcserr_enable().
320 *
321 *
322 * tabx2s() - Pixel-to-world transformation
323 * ----------------------------------------
324 * tabx2s() transforms intermediate world coordinates to world coordinates
325 * using coordinate lookup.
326 *
327 * Given and returned:
328 *   tab       struct tabprm*
329 *                    Tabular transformation parameters.
330 *
331 * Given:
332 *   ncoord,
333 *   nelem     int       The number of coordinates, each of vector length
334 *                    nelem.
335 *
336 *   x         const double[ncoord][nelem]
337 *                    Array of intermediate world coordinates, SI units.
338 *
339 * Returned:
340 *   world     double[ncoord][nelem]
341 *                    Array of world coordinates, in SI units.
342 *
343 *   stat      int[ncoord]
344 *                    Status return value status for each coordinate:
345 *                      0: Success.
346 *                      1: Invalid intermediate world coordinate.
347 *
348 * Function return value:
349 *             int       Status return value:
350 *                      0: Success.
351 *                      1: Null tabprm pointer passed.
352 *                      3: Invalid tabular parameters.
353 *                      4: One or more of the x coordinates were invalid,
354 *                        as indicated by the stat vector.
355 *
356 *                    For returns > 1, a detailed error message is set in
357 *                    tabprm::err if enabled, see wcserr_enable().
358 *
359 *
360 * tabs2x() - World-to-pixel transformation
361 * ----------------------------------------
362 * tabs2x() transforms world coordinates to intermediate world coordinates.
363 *
364 * Given and returned:
365 *   tab       struct tabprm*
366 *                    Tabular transformation parameters.
367 *
368 * Given:
369 *   ncoord,
370 *   nelem     int       The number of coordinates, each of vector length
371 *                    nelem.
372 *   world     const double[ncoord][nelem]
373 *                    Array of world coordinates, in SI units.
374 *
375 * Returned:
376 *   x         double[ncoord][nelem]
377 *                    Array of intermediate world coordinates, SI units.
378 *   stat      int[ncoord]
379 *                    Status return value status for each vector element:
380 *                      0: Success.
381 *                      1: Invalid world coordinate.
382 *
383 * Function return value:
384 *             int       Status return value:
385 *                      0: Success.
386 *                      1: Null tabprm pointer passed.
387 *                      3: Invalid tabular parameters.
388 *                      5: One or more of the world coordinates were
389 *                        invalid, as indicated by the stat vector.
390 *
391 *                    For returns > 1, a detailed error message is set in
392 *                    tabprm::err if enabled, see wcserr_enable().
393 *
394 *
395 * tabprm struct - Tabular transformation parameters
396 * -------------------------------------------------
397 * The tabprm struct contains information required to transform tabular
398 * coordinates.  It consists of certain members that must be set by the user
399 * ("given") and others that are set by the WCSLIB routines ("returned").  Some
400 * of the latter are supplied for informational purposes while others are for
401 * internal use only.
402 *
```

```
403 *    int flag
404 *      (Given and returned) This flag must be set to zero whenever any of the
405 *      following tabprm structure members are set or changed:
406 *
407 *        - tabprm::M (q.v., not normally set by the user),
408 *        - tabprm::K (q.v., not normally set by the user),
409 *        - tabprm::map,
410 *        - tabprm::crval,
411 *        - tabprm::index,
412 *        - tabprm::coord.
413 *
414 *      This signals the initialization routine, tabset(), to recompute the
415 *      returned members of the tabprm struct.  tabset() will reset flag to
416 *      indicate that this has been done.
417 *
418 *      PLEASE NOTE: flag should be set to -1 when tabini() is called for the
419 *      first time for a particular tabprm struct in order to initialize memory
420 *      management.  It must ONLY be used on the first initialization otherwise
421 *      memory leaks may result.
422 *
423 *    int M
424 *      (Given or returned) Number of tabular coordinate axes.
425 *
426 *      If tabini() is used to initialize the tabprm struct (as would normally
427 *      be the case) then it will set M from the value passed to it as a
428 *      function argument.  The user should not subsequently modify it.
429 *
430 *    int *K
431 *      (Given or returned) Pointer to the first element of a vector of length
432 *      tabprm::M whose elements (K_1, K_2,... K_M) record the lengths of the
433 *      axes of the coordinate array and of each indexing vector.
434 *
435 *      If tabini() is used to initialize the tabprm struct (as would normally
436 *      be the case) then it will set K from the array passed to it as a
437 *      function argument.  The user should not subsequently modify it.
438 *
439 *    int *map
440 *      (Given) Pointer to the first element of a vector of length tabprm::M
441 *      that defines the association between axis m in the M-dimensional
442 *      coordinate array (1 <= m <= M) and the indices of the intermediate world
443 *      coordinate and world coordinate arrays, x[] and world[], in the argument
444 *      lists for tabx2s() and tabs2x().
445 *
446 *      When x[] and world[] contain the full complement of coordinate elements
447 *      in image-order, as will usually be the case, then map[m-1] == i-1 for
448 *      axis i in the N-dimensional image (1 <= i <= N).  In terms of the FITS
449 *      keywords
450 *
451 *        map[PVi_3a - 1] == i - 1.
452 *
453 *      However, a different association may result if x[], for example, only
454 *      contains a (relevant) subset of intermediate world coordinate elements.
455 *      For example, if M == 1 for an image with N > 1, it is possible to fill
456 *      x[] with the relevant coordinate element with nelem set to 1.  In this
457 *      case map[0] = 0 regardless of the value of i.
458 *
459 *    double *crval
460 *      (Given) Pointer to the first element of a vector of length tabprm::M
461 *      whose elements contain the index value for the reference pixel for each
462 *      of the tabular coordinate axes.
463 *
464 *    double **index
465 *      (Given) Pointer to the first element of a vector of length tabprm::M of
466 *      pointers to vectors of lengths (K_1, K_2,... K_M) of 0-relative indexes
467 *      (see tabprm::K).
468 *
469 *      The address of any or all of these index vectors may be set to zero,
470 *      i.e.
471 *
472 =        index[m] == 0;
473 *
474 *      this is interpreted as default indexing, i.e.
475 *
476 =        index[m][k] = k;
477 *
478 *    double *coord
479 *      (Given) Pointer to the first element of the tabular coordinate array,
480 *      treated as though it were defined as
481 *
482 =        double coord[K_M]...[K_2][K_1][M];
483 *
484 *      (see tabprm::K) i.e. with the M dimension varying fastest so that the
485 *      M elements of a coordinate vector are stored contiguously in memory.
486 *
487 *    int nc
488 *      (Returned) Total number of coordinate vectors in the coordinate array
489 *      being the product K_1 * K_2 * ... * K_M (see tabprm::K).
```

```
490 *
491 *   int padding
492 *      (An unused variable inserted for alignment purposes only.)
493 *
494 *   int *sense
495 *      (Returned) Pointer to the first element of a vector of length tabprm::M
496 *      whose elements indicate whether the corresponding indexing vector is
497 *      monotonic increasing (+1), or decreasing (-1).
498 *
499 *   int *p0
500 *      (Returned) Pointer to the first element of a vector of length tabprm::M
501 *      of interpolated indices into the coordinate array such that Upsilon_m,
502 *      as defined in Paper III, is equal to (p0[m] + 1) + tabprm::delta[m].
503 *
504 *   double *delta
505 *      (Returned) Pointer to the first element of a vector of length tabprm::M
506 *      of interpolated indices into the coordinate array such that Upsilon_m,
507 *      as defined in Paper III, is equal to (tabprm::p0[m] + 1) + delta[m].
508 *
509 *   double *extrema
510 *      (Returned) Pointer to the first element of an array that records the
511 *      minimum and maximum value of each element of the coordinate vector in
512 *      each row of the coordinate array, treated as though it were defined as
513 *
514 =          double extrema[K_M]...[K_2][2][M]
515 *
516 *      (see tabprm::K).  The minimum is recorded in the first element of the
517 *      compressed K_1 dimension, then the maximum.  This array is used by the
518 *      inverse table lookup function, tabs2x(), to speed up table searches.
519 *
520 *   struct wcserr *err
521 *      (Returned) If enabled, when an error status is returned, this struct
522 *      contains detailed information about the error, see wcserr_enable().
523 *
524 *   int m_flag
525 *      (For internal use only.)
526 *   int m_M
527 *      (For internal use only.)
528 *   int m_N
529 *      (For internal use only.)
530 *   int set_M
531 *      (For internal use only.)
532 *   int m_K
533 *      (For internal use only.)
534 *   int m_map
535 *      (For internal use only.)
536 *   int m_crval
537 *      (For internal use only.)
538 *   int m_index
539 *      (For internal use only.)
540 *   int m_indxs
541 *      (For internal use only.)
542 *   int m_coord
543 *      (For internal use only.)
544 *
545 *
546 * Global variable: const char *tab_errmsg[] - Status return messages
547 * -------------------------------------------------------------------
548 * Error messages to match the status value returned from each function.
549 *
550 *===========================================================================*/
551
552 #ifndef WCSLIB_TAB
553 #define WCSLIB_TAB
554
555 #ifdef __cplusplus
556 extern "C" {
557 #endif
558
559
560 extern const char *tab_errmsg[];
561
562 enum tab_errmsg_enum {
563   TABERR_SUCCESS      = 0,       // Success.
564   TABERR_NULL_POINTER = 1,   // Null tabprm pointer passed.
565   TABERR_MEMORY       = 2, // Memory allocation failed.
566   TABERR_BAD_PARAMS   = 3,     // Invalid tabular parameters.
567   TABERR_BAD_X        = 4,  // One or more of the x coordinates were
568                                 // invalid.
569   TABERR_BAD_WORLD    = 5       // One or more of the world coordinates were
570                                 // invalid.
571 };
572
573 struct tabprm {
574   // Initialization flag (see the prologue above).
575   //------------------------------------------------------------------------
576   int    flag;                   // Set to zero to force initialization.
```

```
577
578   // Parameters to be provided (see the prologue above).
579   //------------------------------------------------------------------------
580   int    M;                       // Number of tabular coordinate axes.
581   int    *K;                      // Vector of length M whose elements
582                                   // (K_1, K_2,... K_M) record the lengths of
583                                   // the axes of the coordinate array and of
584                                   // each indexing vector.
585   int    *map;                         // Vector of length M usually such that
586                                   // map[m-1] == i-1 for coordinate array
587                                   // axis m and image axis i (see above).
588   double *crval;            // Vector of length M containing the index
589                                   // value for the reference pixel for each
590                                   // of the tabular coordinate axes.
591   double **index;           // Vector of pointers to M indexing vectors
592                                   // of lengths (K_1, K_2,... K_M).
593   double *coord;            // (1+M)-dimensional tabular coordinate
594                                   // array (see above).
595
596   // Information derived from the parameters supplied.
597   //------------------------------------------------------------------------
598   int    nc;                      // Number of coordinate vectors (of length
599                                   // M) in the coordinate array.
600   int    padding;                  // (Dummy inserted for alignment purposes.)
601   int    *sense;            // Vector of M flags that indicate whether
602                                   // the Mth indexing vector is monotonic
603                                   // increasing, or else decreasing.
604   int    *p0;               // Vector of M indices.
605   double *delta;            // Vector of M increments.
606   double *extrema;                 // (1+M)-dimensional array of coordinate
607                                   // extrema.
608
609   // Error handling
610   //------------------------------------------------------------------------
611   struct wcserr *err;
612
613   // Private - the remainder are for memory management.
614   //------------------------------------------------------------------------
615   int    m_flag, m_M, m_N;
616   int    set_M;
617   int    *m_K, *m_map;
618   double *m_crval, **m_index, **m_indxs, *m_coord;
619 };
620
621 // Size of the tabprm struct in int units, used by the Fortran wrappers.
622 #define TABLEN (sizeof(struct tabprm)/sizeof(int))
623
624
625 int tabini(int alloc, int M, const int K[], struct tabprm *tab);
626
627 int tabmem(struct tabprm *tab);
628
629 int tabcpy(int alloc, const struct tabprm *tabsrc, struct tabprm *tabdst);
630
631 int tabcmp(int cmp, double tol, const struct tabprm *tab1,
632           const struct tabprm *tab2, int *equal);
633
634 int tabfree(struct tabprm *tab);
635
636 int tabsize(const struct tabprm *tab, int size[2]);
637
638 int tabprt(const struct tabprm *tab);
639
640 int tabperr(const struct tabprm *tab, const char *prefix);
641
642 int tabset(struct tabprm *tab);
643
644 int tabx2s(struct tabprm *tab, int ncoord, int nelem, const double x[],
645           double world[], int stat[]);
646
647 int tabs2x(struct tabprm *tab, int ncoord, int nelem, const double world[],
648           double x[], int stat[]);
649
650
651 // Deprecated.
652 #define tabini_errmsg tab_errmsg
653 #define tabcpy_errmsg tab_errmsg
654 #define tabfree_errmsg tab_errmsg
655 #define tabprt_errmsg tab_errmsg
656 #define tabset_errmsg tab_errmsg
657 #define tabx2s_errmsg tab_errmsg
658 #define tabs2x_errmsg tab_errmsg
659
660 #ifdef __cplusplus
661 }
662 #endif
663
```

```
664 #endif // WCSLIB_TAB
```

19.23 wcs.h File Reference

```
#include "lin.h"
#include "cel.h"
#include "spc.h"
```

Data Structures

- struct pvcard

    Store for ***PV****i_ma keyrecords.*

- struct pscard

    Store for ***PS****i_ma keyrecords.*

- struct auxprm

    Additional auxiliary parameters.

- struct wcsprm

    Coordinate transformation parameters.

Macros

- #define WCSSUB_LONGITUDE 0x1001

    Mask for extraction of longitude axis by wcssub().

- #define WCSSUB_LATITUDE 0x1002

    Mask for extraction of latitude axis by wcssub().

- #define WCSSUB_CUBEFACE 0x1004

    Mask for extraction of ***CUBEFACE*** *axis by wcssub().*

- #define WCSSUB_CELESTIAL 0x1007

    Mask for extraction of celestial axes by wcssub().

- #define WCSSUB_SPECTRAL 0x1008

    Mask for extraction of spectral axis by wcssub().

- #define WCSSUB_STOKES 0x1010

    Mask for extraction of ***STOKES*** *axis by wcssub().*

- #define WCSSUB_TIME 0x1020
- #define WCSCOMPARE_ANCILLARY 0x0001
- #define WCSCOMPARE_TILING 0x0002
- #define WCSCOMPARE_CRPIX 0x0004
- #define PVLEN (sizeof(struct pvcard)/sizeof(int))
- #define PSLEN (sizeof(struct pscard)/sizeof(int))
- #define AUXLEN (sizeof(struct auxprm)/sizeof(int))
- #define WCSLEN (sizeof(struct wcsprm)/sizeof(int))

    Size of the wcsprm struct in int units.

- #define wcscopy(alloc, wcssrc, wcsdst) wcssub(alloc, wcssrc, 0x0, 0x0, wcsdst)

    Copy routine for the wcsprm struct.

- #define wcsini_errmsg wcs_errmsg

    Deprecated.

- #define wcssub_errmsg wcs_errmsg

    Deprecated.

- #define wcscopy_errmsg wcs_errmsg

    Deprecated.
- #define wcsfree_errmsg wcs_errmsg

    Deprecated.
- #define wcsprt_errmsg wcs_errmsg

    Deprecated.
- #define wcsset_errmsg wcs_errmsg

    Deprecated.
- #define wcsp2s_errmsg wcs_errmsg

    Deprecated.
- #define wcss2p_errmsg wcs_errmsg

    Deprecated.
- #define wcsmix_errmsg wcs_errmsg

    Deprecated.

Enumerations

- enum wcs_errmsg_enum {
  WCSERR_SUCCESS = 0 , WCSERR_NULL_POINTER = 1 , WCSERR_MEMORY = 2 , WCSERR_SINGULAR_MTX
  = 3 ,
  WCSERR_BAD_CTYPE = 4 , WCSERR_BAD_PARAM = 5 , WCSERR_BAD_COORD_TRANS = 6 ,
  WCSERR_ILL_COORD_TRANS = 7 ,
  WCSERR_BAD_PIX = 8 , WCSERR_BAD_WORLD = 9 , WCSERR_BAD_WORLD_COORD = 10 ,
  WCSERR_NO_SOLUTION = 11 ,
  WCSERR_BAD_SUBIMAGE = 12 , WCSERR_NON_SEPARABLE = 13 , WCSERR_UNSET = 14 }

Functions

- int wcsnpv (int n)

    Memory allocation for **PV**i_ma.
- int wcsnps (int n)

    Memory allocation for **PS**i_ma.
- int wcsini (int alloc, int naxis, struct wcsprm *wcs)

    Default constructor for the wcsprm struct.
- int wcsinit (int alloc, int naxis, struct wcsprm *wcs, int npvmax, int npsmax, int ndpmax)

    Default constructor for the wcsprm struct.
- int wcsauxi (int alloc, struct wcsprm *wcs)

    Default constructor for the auxprm struct.
- int wcssub (int alloc, const struct wcsprm *wcssrc, int *nsub, int axes[ ], struct wcsprm *wcsdst)

    Subimage extraction routine for the wcsprm struct.
- int wcscompare (int cmp, double tol, const struct wcsprm *wcs1, const struct wcsprm *wcs2, int *equal)

    Compare two wcsprm structs for equality.
- int wcsfree (struct wcsprm *wcs)

    Destructor for the wcsprm struct.
- int wcstrim (struct wcsprm *wcs)

    Free unused arrays in the wcsprm struct.
- int wcssize (const struct wcsprm *wcs, int sizes[2])

    Compute the size of a wcsprm struct.
- int auxsize (const struct auxprm *aux, int sizes[2])

    Compute the size of a auxprm struct.

- int wcsprt (const struct wcsprm ∗wcs)

    Print routine for the wcsprm struct.

- int wcsperr (const struct wcsprm ∗wcs, const char ∗prefix)

    Print error messages from a wcsprm struct.

- int wcsbchk (struct wcsprm ∗wcs, int bounds)

    Enable/disable bounds checking.

- int wcsset (struct wcsprm ∗wcs)

    Setup routine for the wcsprm struct.

- int wcsp2s (struct wcsprm ∗wcs, int ncoord, int nelem, const double pixcrd[ ], double imgcrd[ ], double phi[ ], double theta[ ], double world[ ], int stat[ ])

    Pixel-to-world transformation.

- int wcss2p (struct wcsprm ∗wcs, int ncoord, int nelem, const double world[ ], double phi[ ], double theta[ ], double imgcrd[ ], double pixcrd[ ], int stat[ ])

    World-to-pixel transformation.

- int wcsmix (struct wcsprm ∗wcs, int mixpix, int mixcel, const double vspan[2], double vstep, int viter, double world[ ], double phi[ ], double theta[ ], double imgcrd[ ], double pixcrd[ ])

    Hybrid coordinate transformation.

- int wcsccs (struct wcsprm ∗wcs, double lng2p1, double lat2p1, double lng1p2, const char ∗clng, const char ∗clat, const char ∗radesys, double equinox, const char ∗alt)

    Change celestial coordinate system.

- int wcssptr (struct wcsprm ∗wcs, int ∗i, char ctype[9])

    Spectral axis translation.

- const char ∗ wcslib_version (int vers[3])

Variables

- const char ∗ wcs_errmsg [ ]

    Status return messages.

19.23.1 Detailed Description

Routines in this suite implement the FITS World Coordinate System (WCS) standard which defines methods to be used for computing world coordinates from image pixel coordinates, and vice versa. The standard, and proposed extensions for handling distortions, are described in

```
"Representations of world coordinates in FITS",
Greisen, E.W., & Calabretta, M.R. 2002, A&A, 395, 1061 (WCS Paper I)
"Representations of celestial coordinates in FITS",
Calabretta, M.R., & Greisen, E.W. 2002, A&A, 395, 1077 (WCS Paper II)
"Representations of spectral coordinates in FITS",
Greisen, E.W., Calabretta, M.R., Valdes, F.G., & Allen, S.L.
2006, A&A, 446, 747 (WCS Paper III)
"Representations of distortions in FITS world coordinate systems",
Calabretta, M.R. et al. (WCS Paper IV, draft dated 2004/04/22),
available from http://www.atnf.csiro.au/people/Mark.Calabretta
"Mapping on the HEALPix grid",
Calabretta, M.R., & Roukema, B.F. 2007, MNRAS, 381, 865 (WCS Paper V)
"Representing the 'Butterfly' Projection in FITS -- Projection Code XPH",
Calabretta, M.R., & Lowe, S.R. 2013, PASA, 30, e050 (WCS Paper VI)
"Representations of time coordinates in FITS -
 Time and relative dimension in space",
Rots, A.H., Bunclark, P.S., Calabretta, M.R., Allen, S.L.,
Manchester, R.N., & Thompson, W.T. 2015, A&A, 574, A36 (WCS Paper VII)
```

These routines are based on the wcsprm struct which contains all information needed for the computations. The struct contains some members that must be set by the user, and others that are maintained by these routines, somewhat like a C++ class but with no encapsulation.

wcsnpv(), wcsnps(), wcsini(), wcsinit(), wcssub(), wcsfree(), and wcstrim(), are provided to manage the wcsprm struct, wcssize() computes its total size including allocated memory, and wcsprt() prints its contents. Refer to the

description of the wcsprm struct for an explanation of the anticipated usage of these routines. wcscopy(), which does a deep copy of one wcsprm struct to another, is defined as a preprocessor macro function that invokes wcssub().

wcsperr() prints the error message(s) (if any) stored in a wcsprm struct, and the linprm, celprm, prjprm, spcprm, and tabprm structs that it contains.

A setup routine, wcsset(), computes intermediate values in the wcsprm struct from parameters in it that were supplied by the user. The struct always needs to be set up by wcsset() but this need not be called explicitly - refer to the explanation of wcsprm::flag.

wcsp2s() and wcss2p() implement the WCS world coordinate transformations. In fact, they are high level driver routines for the WCS linear, logarithmic, celestial, spectral and tabular transformation routines described in lin.h, log.h, cel.h, spc.h and tab.h.

Given either the celestial longitude or latitude plus an element of the pixel coordinate a hybrid routine, wcsmix(), iteratively solves for the unknown elements.

wcsccs() changes the celestial coordinate system of a wcsprm struct, for example, from equatorial to galactic, and wcssptr() translates the spectral axis. For example, a **'FREQ'** axis may be translated into **'ZOPT-F2W'** and vice versa.

wcslib_version() returns the WCSLIB version number.

Quadcube projections:
The quadcube projections (**TSC**, **CSC**, **QSC**) may be represented in FITS in either of two ways:

a: The six faces may be laid out in one plane and numbered as follows:

```
          0
4  3  2   1  4  3  2
          5
```

Faces 2, 3 and 4 may appear on one side or the other (or both). The world-to-pixel routines map faces 2, 3 and 4 to the left but the pixel-to-world routines accept them on either side.

b: The "COBE" convention in which the six faces are stored in a three-dimensional structure using a **CUBEFACE** axis indexed from 0 to 5 as above.

These routines support both methods; wcsset() determines which is being used by the presence or absence of a **CUBEFACE** axis in ctype[]. wcsp2s() and wcss2p() translate the **CUBEFACE** axis representation to the single plane representation understood by the lower-level WCSLIB projection routines.

19.23.2 Macro Definition Documentation

19.23.2.1 WCSSUB_LONGITUDE `#define WCSSUB_LONGITUDE 0x1001`

Mask to use for extracting the longitude axis when sub-imaging, refer to the *axes* argument of wcssub().

19.23.2.2 WCSSUB_LATITUDE `#define WCSSUB_LATITUDE 0x1002`

Mask to use for extracting the latitude axis when sub-imaging, refer to the *axes* argument of wcssub().

19.23.2.3 WCSSUB_CUBEFACE `#define WCSSUB_CUBEFACE 0x1004`

Mask to use for extracting the **CUBEFACE** axis when sub-imaging, refer to the *axes* argument of wcssub().

19.23.2.4 WCSSUB_CELESTIAL `#define WCSSUB_CELESTIAL 0x1007`

Mask to use for extracting the celestial axes (longitude, latitude and cubeface) when sub-imaging, refer to the *axes* argument of wcssub().

19.23.2.5 WCSSUB_SPECTRAL `#define WCSSUB_SPECTRAL 0x1008`

Mask to use for extracting the spectral axis when sub-imaging, refer to the *axes* argument of wcssub().

19.23.2.6 WCSSUB_STOKES `#define WCSSUB_STOKES 0x1010`

Mask to use for extracting the **STOKES** axis when sub-imaging, refer to the *axes* argument of wcssub().

19.23.2.7 WCSSUB_TIME `#define WCSSUB_TIME 0x1020`

19.23.2.8 WCSCOMPARE_ANCILLARY `#define WCSCOMPARE_ANCILLARY 0x0001`

19.23.2.9 WCSCOMPARE_TILING `#define WCSCOMPARE_TILING 0x0002`

19.23.2.10 WCSCOMPARE_CRPIX `#define WCSCOMPARE_CRPIX 0x0004`

19.23.2.11 PVLEN `#define PVLEN (sizeof(struct pvcard)/sizeof(int))`

19.23.2.12 PSLEN `#define PSLEN (sizeof(struct pscard)/sizeof(int))`

19.23.2.13 AUXLEN `#define AUXLEN (sizeof(struct auxprm)/sizeof(int))`

19.23.2.14 WCSLEN `#define WCSLEN (sizeof(struct wcsprm)/sizeof(int))`

Size of the wcsprm struct in *int* units, used by the Fortran wrappers.

19.23.2.15 wcscopy `#define wcscopy(`
        alloc,
        wcssrc,
        wcsdst `)` `wcssub(alloc, wcssrc, 0x0, 0x0, wcsdst)`

wcscopy() does a deep copy of one wcsprm struct to another. As of WCSLIB 3.6, it is implemented as a preprocessor macro that invokes wcssub() with the nsub and axes pointers both set to zero.

19.23.2.16 wcsini_errmsg `#define wcsini_errmsg` wcs_errmsg

Deprecated Added for backwards compatibility, use wcs_errmsg directly now instead.

19.23.2.17 wcssub_errmsg `#define wcssub_errmsg` wcs_errmsg

Deprecated Added for backwards compatibility, use wcs_errmsg directly now instead.

19.23.2.18 wcscopy_errmsg `#define wcscopy_errmsg` wcs_errmsg

Deprecated Added for backwards compatibility, use wcs_errmsg directly now instead.

19.23.2.19 wcsfree_errmsg `#define wcsfree_errmsg` wcs_errmsg

Deprecated Added for backwards compatibility, use wcs_errmsg directly now instead.

19.23.2.20 wcsprt_errmsg `#define wcsprt_errmsg` wcs_errmsg

Deprecated Added for backwards compatibility, use wcs_errmsg directly now instead.

19.23.2.21 wcsset_errmsg `#define wcsset_errmsg` `wcs_errmsg`

Deprecated Added for backwards compatibility, use wcs_errmsg directly now instead.

19.23.2.22 wcsp2s_errmsg `#define wcsp2s_errmsg` `wcs_errmsg`

Deprecated Added for backwards compatibility, use wcs_errmsg directly now instead.

19.23.2.23 wcss2p_errmsg `#define wcss2p_errmsg` `wcs_errmsg`

Deprecated Added for backwards compatibility, use wcs_errmsg directly now instead.

19.23.2.24 wcsmix_errmsg `#define wcsmix_errmsg` `wcs_errmsg`

Deprecated Added for backwards compatibility, use wcs_errmsg directly now instead.

19.23.3 Enumeration Type Documentation

19.23.3.1 wcs_errmsg_enum `enum` `wcs_errmsg_enum`

Enumerator

WCSERR_SUCCESS	
WCSERR_NULL_POINTER	
WCSERR_MEMORY	
WCSERR_SINGULAR_MTX	
WCSERR_BAD_CTYPE	
WCSERR_BAD_PARAM	
WCSERR_BAD_COORD_TRANS	
WCSERR_ILL_COORD_TRANS	
WCSERR_BAD_PIX	
WCSERR_BAD_WORLD	
WCSERR_BAD_WORLD_COORD	
WCSERR_NO_SOLUTION	
WCSERR_BAD_SUBIMAGE	
WCSERR_NON_SEPARABLE	
WCSERR_UNSET	

19.23.4 Function Documentation

19.23.4.1 wcsnpv() `int wcsnpv (`
          `int n )`

wcsnpv() sets or gets the value of NPVMAX (default 64). This global variable controls the number of pvcard structs, for holding **PV**i_ma keyvalues, that wcsini() should allocate space for. It is also used by wcsinit() as the default value of npvmax.

PLEASE NOTE: This function is not thread-safe.

Parameters

in	*n*	Value of NPVMAX; ignored if < 0. Use a value less than zero to get the current value.

Returns

> Current value of NPVMAX.

19.23.4.2 wcsnps() `int wcsnps (`
          `int n )`

wcsnps() sets or gets the value of NPSMAX (default 8). This global variable controls the number of pscard structs, for holding **PS**i_ma keyvalues, that wcsini() should allocate space for. It is also used by wcsinit() as the default value of npsmax.

PLEASE NOTE: This function is not thread-safe.

Parameters

in	*n*	Value of NPSMAX; ignored if < 0. Use a value less than zero to get the current value.

Returns

> Current value of NPSMAX.

19.23.4.3 wcsini() `int wcsini (`
          `int alloc,`
          `int naxis,`
          `struct wcsprm * wcs )`

wcsini() is a thin wrapper on **wcsinit**(). It invokes it with npvmax, npsmax, and ndpmax set to -1 which causes it to use the values of the global variables NDPMAX, NPSMAX, and NDPMAX. It is thereby potentially thread-unsafe if these variables are altered dynamically via wcsnpv(), wcsnps(), and disndp(). Use **wcsinit**() for a thread-safe alternative in this case.

19.23.4.4 wcsinit() `int wcsinit (`
          `int alloc,`
          `int naxis,`
          `struct wcsprm * wcs,`
          `int npvmax,`
          `int npsmax,`
          `int ndpmax )`

wcsinit() optionally allocates memory for arrays in a wcsprm struct and sets all members of the struct to default values.

PLEASE NOTE: every wcsprm struct should be initialized by **wcsinit**(), possibly repeatedly. On the first invokation, and only the first invokation, wcsprm::flag must be set to -1 to initialize memory management, regardless of whether **wcsinit**() will actually be used to allocate memory.

Parameters

in	alloc	If true, allocate memory unconditionally for the crpix, etc. arrays. Please note that memory is never allocated by **wcsinit**() for the auxprm, tabprm, nor wtbarr structs. If false, it is assumed that pointers to these arrays have been set by the user except if they are null pointers in which case memory will be allocated for them regardless. (In other words, setting alloc true saves having to initalize these pointers to zero.)
in	naxis	The number of world coordinate axes. This is used to determine the length of the various wcsprm vectors and matrices and therefore the amount of memory to allocate for them.
in,out	wcs	Coordinate transformation parameters. Note that, in order to initialize memory management, wcsprm::flag should be set to -1 when wcs is initialized for the first time (memory leaks may result if it had already been initialized).
in	npvmax	The number of **PV**i_ma keywords to allocate space for. If set to -1, the value of the global variable NPVMAX will be used. This is potentially thread-unsafe if wcsnpv() is being used dynamically to alter its value.
in	npsmax	The number of **PS**i_ma keywords to allocate space for. If set to -1, the value of the global variable NPSMAX will be used. This is potentially thread-unsafe if wcsnps() is being used dynamically to alter its value.
in	ndpmax	The number of DPja or DQia keywords to allocate space for. If set to -1, the value of the global variable NDPMAX will be used. This is potentially thread-unsafe if disndp() is being used dynamically to alter its value.

Returns

Status return value:

- 0: Success.
- 1: Null wcsprm pointer passed.
- 2: Memory allocation failed.

For returns > 1, a detailed error message is set in wcsprm::err if enabled, see wcserr_enable().

19.23.4.5 wcsauxi() `int wcsauxi (`
          `int alloc,`
          `struct wcsprm * wcs )`

wcsauxi() optionally allocates memory for an auxprm struct, attaches it to wcsprm, and sets all members of the struct to default values.

Parameters

in	*alloc*	If true, allocate memory unconditionally for the auxprm struct.
		If false, it is assumed that wcsprm::aux has already been set to point to an auxprm struct, in which case the user is responsible for managing that memory. However, if wcsprm::aux is a null pointer, memory will be allocated regardless. (In other words, setting alloc true saves having to initalize the pointer to zero.)
in,out	*wcs*	Coordinate transformation parameters.

Returns

Status return value:

- 0: Success.
- 1: Null wcsprm pointer passed.
- 2: Memory allocation failed.

19.23.4.6   wcssub()   `int wcssub (`
          `int alloc,`
          `const struct wcsprm * wcssrc,`
          `int * nsub,`
          `int axes[],`
          `struct wcsprm * wcsdst )`

wcssub() extracts the coordinate description for a subimage from a wcsprm struct. It does a deep copy, using wcsinit() to allocate memory for its arrays if required. Only the "information to be provided" part of the struct is extracted. Consequently, wcsset() need not have been, and won't be invoked on the struct from which the subimage is extracted. A call to wcsset() is required to set up the subimage struct.

The world coordinate system of the subimage must be separable in the sense that the world coordinates at any point in the subimage must depend only on the pixel coordinates of the axes extracted. In practice, this means that the linear transformation matrix of the original image must not contain non-zero off-diagonal terms that associate any of the subimage axes with any of the non-subimage axes. Likewise, if any distortions are associated with the subimage axes, they must not depend on any of the axes that are not being extracted.

Note that while the required elements of the tabprm array are extracted, the wtbarr array is not. (Thus it is not appropriate to call **wcssub**() after wcstab() but before filling the tabprm structs - refer to wcshdr.h.)

wcssub() can also add axes to a wcsprm struct. The new axes will be created using the defaults set by wcsinit() which produce a simple, unnamed, linear axis with world coordinate equal to the pixel coordinate. These default values can be changed afterwards, before invoking wcsset().

Parameters

in	*alloc*	If true, allocate memory for the crpix, etc. arrays in the destination. Otherwise, it is assumed that pointers to these arrays have been set by the user except if they are null pointers in which case memory will be allocated for them regardless.
in	*wcssrc*	Struct to extract from.
in,out	*nsub*	

Parameters

in,out	*axes*	Vector of length ∗nsub containing the image axis numbers (1-relative) to extract. Order is significant; axes[0] is the axis number of the input image that corresponds to the first axis in the subimage, etc.
		Use an axis number of 0 to create a new axis using the defaults set by [wcsinit()](#). They can be changed later.
		nsub (the pointer) may be set to zero, and so also may ∗nsub, which is interpreted to mean all axes in the input image; the number of axes will be returned if nsub != 0x0.
		axes itself (the pointer) may be set to zero to indicate the first ∗nsub axes in their original order.
		Set both nsub (or ∗nsub) and axes to zero to do a deep copy of one wcsprm struct to another.
		Subimage extraction by coordinate axis type may be done by setting the elements of axes[] to the following special preprocessor macro values:
		• WCSSUB_LONGITUDE: Celestial longitude.
		• WCSSUB_LATITUDE: Celestial latitude.
		• WCSSUB_CUBEFACE: Quadcube **CUBEFACE** axis.
		• WCSSUB_SPECTRAL: Spectral axis.
		• WCSSUB_STOKES: Stokes axis.
		• WCSSUB_TIME: Time axis.
		Refer to the notes (below) for further usage examples.
		On return, ∗nsub will be set to the number of axes in the subimage; this may be zero if there were no axes of the required type(s) (in which case no memory will be allocated). axes[] will contain the axis numbers that were extracted, or 0 for newly created axes. The vector length must be sufficient to contain all axis numbers. No checks are performed to verify that the coordinate axes are consistent, this is done by [wcsset()](#).
in,out	*wcsdst*	Struct describing the subimage. [wcsprm::flag](#) should be set to -1 if wcsdst was not previously initialized (memory leaks may result if it was previously initialized).

Returns

Status return value:

- 0: Success.
- 1: Null wcsprm pointer passed.
- 2: Memory allocation failed.
- 12: Invalid subimage specification.
- 13: Non-separable subimage coordinate system.

For returns > 1, a detailed error message is set in [wcsprm::err](#) if enabled, see [wcserr_enable()](#).

Notes:

1. Combinations of subimage axes of particular types may be extracted in the same order as they occur in the input image by combining preprocessor codes, for example
   ```
   *nsub = 1;
   axes[0] = WCSSUB_LONGITUDE | WCSSUB_LATITUDE | WCSSUB_SPECTRAL;
   ```

would extract the longitude, latitude, and spectral axes in the same order as the input image. If one of each were present, *nsub = 3 would be returned.

For convenience, WCSSUB_CELESTIAL is defined as the combination WCSSUB_LONGITUDE | WCSSUB_LATITUDE | WCSSUB_CUBEFACE.

The codes may also be negated to extract all but the types specified, for example

```
*nsub = 4;
axes[0] = WCSSUB_LONGITUDE;
axes[1] = WCSSUB_LATITUDE;
axes[2] = WCSSUB_CUBEFACE;
axes[3] = -(WCSSUB_SPECTRAL | WCSSUB_STOKES);
```

The last of these specifies all axis types other than spectral or Stokes. Extraction is done in the order specified by axes[] a longitude axis (if present) would be extracted first (via axes[0]) and not subsequently (via axes[3]). Likewise for the latitude and cubeface axes in this example.

From the foregoing, it is apparent that the value of *nsub returned may be less than or greater than that given. However, it will never exceed the number of axes in the input image (plus the number of newly-created axes if any were specified on input).

19.23.4.7 wcscompare() ```int wcscompare (```
```
        int cmp,
        double tol,
        const struct wcsprm * wcs1,
        const struct wcsprm * wcs2,
        int * equal )
```

wcscompare() compares two wcsprm structs for equality.

Parameters

in	*cmp*	A bit field controlling the strictness of the comparison. When 0, all fields must be identical. The following constants may be or'ed together to relax the comparison: • W**CSC**OMPARE_ANCILLARY: Ignore ancillary keywords that don't change the WCS transformation, such as **DATE−OBS** or EQUINOX. • W**CSC**OMPARE_TILING: Ignore integral differences in **CRPIX**ja. This is the 'tiling' condition, where two WCSes cover different regions of the same map projection and align on the same map grid. • W**CSC**OMPARE_CRPIX: Ignore any differences at all in **CRPIX**ja. The two WCSes cover different regions of the same map projection but may not align on the same map grid. Overrides W**CSC**OMPARE_TILING.
in	*tol*	Tolerance for comparison of floating-point values. For example, for tol == 1e-6, all floating-point values in the structs must be equal to the first 6 decimal places. A value of 0 implies exact equality.
in	*wcs1*	The first wcsprm struct to compare.
in	*wcs2*	The second wcsprm struct to compare.
out	*equal*	Non-zero when the given structs are equal.

Returns

Status return value:

• 0: Success.

- 1: Null pointer passed.

19.23.4.8 wcsfree() `int wcsfree (`
    `struct wcsprm * wcs )`

wcsfree() frees memory allocated for the wcsprm arrays by wcsinit() and/or wcsset(). wcsinit() records the memory it allocates and **wcsfree**() will only attempt to free this.

PLEASE NOTE: wcsfree() must not be invoked on a wcsprm struct that was not initialized by wcsinit().

Parameters

in,out	*wcs*	Coordinate transformation parameters.

Returns

  Status return value:

- 0: Success.
- 1: Null wcsprm pointer passed.

19.23.4.9 wcstrim() `int wcstrim (`
    `struct wcsprm * wcs )`

wcstrim() frees memory allocated by wcsinit() for arrays in the wcsprm struct that remains unused after it has been set up by wcsset().

The free'd array members are associated with FITS WCS keyrecords that are rarely used and usually just bloat the struct: wcsprm::crota, wcsprm::colax, wcsprm::cname, wcsprm::crder, wcsprm::csyer, wcsprm::czphs, and wcsprm::cperi. If unused, wcsprm::pv, wcsprm::ps, and wcsprm::cd are also freed.

Once these arrays have been freed, a test such as
`if (!undefined(wcs->cname[i])) {...}`

must be protected as follows
`if (wcs->cname && !undefined(wcs->cname[i])) {...}`

In addition, if wcsprm::npv is non-zero but less than wcsprm::npvmax, then the unused space in wcsprm::pv will be recovered (using realloc()). Likewise for wcsprm::ps.

Parameters

in,out	*wcs*	Coordinate transformation parameters.

Returns

  Status return value:

- 0: Success.

- 1: Null wcsprm pointer passed.

- 14: wcsprm struct is unset.

19.23.4.10 wcssize()   `int wcssize (`
        `const struct wcsprm * wcs,`
        `int sizes[2] )`

wcssize() computes the full size of a wcsprm struct, including allocated memory.

Parameters

in	*wcs*	Coordinate transformation parameters.
		If NULL, the base size of the struct and the allocated size are both set to zero.
out	*sizes*	The first element is the base size of the struct as returned by sizeof(struct wcsprm). The second element is the total allocated size, in bytes, assuming that the allocation was done by wcsini(). This figure includes memory allocated for members of constituent structs, such as wcsprm::lin.
		It is not an error for the struct not to have been set up via wcsset(), which normally results in additional memory allocation.

Returns

Status return value:

- 0: Success.

19.23.4.11 auxsize()   `int auxsize (`
        `const struct auxprm * aux,`
        `int sizes[2] )`

auxsize() computes the full size of a auxprm struct, including allocated memory.

Parameters

in	*aux*	Auxiliary coordinate information.
		If NULL, the base size of the struct and the allocated size are both set to zero.
out	*sizes*	The first element is the base size of the struct as returned by sizeof(struct auxprm). The second element is the total allocated size, in bytes, currently zero.

Returns

Status return value:

- 0: Success.

19.23.4.12 wcsprt() `int wcsprt (`
`const struct wcsprm * wcs )`

wcsprt() prints the contents of a wcsprm struct using wcsprintf(). Mainly intended for diagnostic purposes.

Parameters

in	*wcs*	Coordinate transformation parameters.

Returns

Status return value:

- 0: Success.
- 1: Null wcsprm pointer passed.

19.23.4.13 wcsperr() `int wcsperr (`
`const struct wcsprm * wcs,`
`const char * prefix )`

wcsperr() prints the error message(s), if any, stored in a wcsprm struct, and the linprm, celprm, prjprm, spcprm, and tabprm structs that it contains. If there are no errors then nothing is printed. It uses wcserr_prt(), q.v.

Parameters

in	*wcs*	Coordinate transformation parameters.
in	*prefix*	If non-NULL, each output line will be prefixed with this string.

Returns

Status return value:

- 0: Success.
- 1: Null wcsprm pointer passed.

19.23.4.14 wcsbchk() `int wcsbchk (`
`struct wcsprm * wcs,`
`int bounds )`

wcsbchk() is used to control bounds checking in the projection routines. Note that wcsset() always enables bounds checking. **wcsbchk**() will invoke wcsset() on the wcsprm struct beforehand if necessary.

Parameters

in,out	*wcs*	Coordinate transformation parameters.

Parameters

in	*bounds*	If bounds&1 then enable strict bounds checking for the spherical-to-Cartesian (s2x) transformation for the AZP, SZP, TAN, **SIN**, ZPN, and COP projections.
		If bounds&2 then enable strict bounds checking for the Cartesian-to-spherical (x2s) transformation for the HPX and XPH projections.
		If bounds&4 then enable bounds checking on the native coordinates returned by the Cartesian-to-spherical (x2s) transformations using prjchk().
		Zero it to disable all checking.

Returns

Status return value:

- 0: Success.
- 1: Null wcsprm pointer passed.

19.23.4.15   wcsset()   int wcsset (
            struct wcsprm * *wcs*)

wcsset() sets up a wcsprm struct according to information supplied within it (refer to the description of the wcsprm struct).

wcsset() recognizes the **NCP** projection and converts it to the equivalent **SIN** projection and likewise translates **GLS** into **SFL**. It also translates the AIPS spectral types (**'FREQ-LSR'**, **'FELO-HEL'**, etc.), possibly changing the input header keywords wcsprm::ctype and/or wcsprm::specsys if necessary.

Note that this routine need not be called directly; it will be invoked by wcsp2s() and wcss2p() if the wcsprm::flag is anything other than a predefined magic value.

Parameters

in,out	*wcs*	Coordinate transformation parameters.

Returns

Status return value:

- 0: Success.
- 1: Null wcsprm pointer passed.
- 2: Memory allocation failed.
- 3: Linear transformation matrix is singular.
- 4: Inconsistent or unrecognized coordinate axis types.
- 5: Invalid parameter value.
- 6: Invalid coordinate transformation parameters.
- 7: Ill-conditioned coordinate transformation parameters.

For returns > 1, a detailed error message is set in wcsprm::err if enabled, see wcserr_enable().

Notes:

1. **wcsset**() always enables strict bounds checking in the projection routines (via a call to prjini()). Use wcsbchk() to modify bounds-checking after **wcsset**() is invoked.

19.23.4.16 wcsp2s() `int wcsp2s (`
```
        struct wcsprm * wcs,
        int ncoord,
        int nelem,
        const double pixcrd[],
        double imgcrd[],
        double phi[],
        double theta[],
        double world[],
        int stat[] )
```

wcsp2s() transforms pixel coordinates to world coordinates.

Parameters

in,out	*wcs*	Coordinate transformation parameters.
in	*ncoord,nelem*	The number of coordinates, each of vector length nelem but containing wcs.naxis coordinate elements. Thus nelem must equal or exceed the value of the **NAXIS** keyword unless ncoord == 1, in which case nelem is not used.
in	*pixcrd*	Array of pixel coordinates.
out	*imgcrd*	Array of intermediate world coordinates. For celestial axes, imgcrd[][wcs.lng] and imgcrd[][wcs.lat] are the projected x-, and y-coordinates in pseudo "degrees". For spectral axes, imgcrd[][wcs.spec] is the intermediate spectral coordinate, in SI units.
out	*phi,theta*	Longitude and latitude in the native coordinate system of the projection [deg].
out	*world*	Array of world coordinates. For celestial axes, world[][wcs.lng] and world[][wcs.lat] are the celestial longitude and latitude [deg]. For spectral axes, imgcrd[][wcs.spec] is the intermediate spectral coordinate, in SI units.
out	*stat*	Status return value for each coordinate: • 0: Success. 1+: A bit mask indicating invalid pixel coordinate element(s).

Returns

Status return value:

• 0: Success.

• 1: Null wcsprm pointer passed.

• 2: Memory allocation failed.

• 3: Linear transformation matrix is singular.

• 4: Inconsistent or unrecognized coordinate axis types.

• 5: Invalid parameter value.

• 6: Invalid coordinate transformation parameters.

• 7: Ill-conditioned coordinate transformation parameters.

• 8: One or more of the pixel coordinates were invalid, as indicated by the stat vector.

For returns > 1, a detailed error message is set in wcsprm::err if enabled, see wcserr_enable().

19.23.4.17 wcss2p() `int wcss2p (`
```
        struct wcsprm * wcs,
        int ncoord,
        int nelem,
        const double world[],
        double phi[],
        double theta[],
        double imgcrd[],
        double pixcrd[],
        int stat[] )
```

wcss2p() transforms world coordinates to pixel coordinates.

Parameters

`in,out`	*wcs*	Coordinate transformation parameters.
`in`	*ncoord,nelem*	The number of coordinates, each of vector length nelem but containing wcs.naxis coordinate elements. Thus nelem must equal or exceed the value of the **NAXIS** keyword unless ncoord == 1, in which case nelem is not used.
`in`	*world*	Array of world coordinates. For celestial axes, world[][wcs.lng] and world[][wcs.lat] are the celestial longitude and latitude [deg]. For spectral axes, world[][wcs.spec] is the spectral coordinate, in SI units.
`out`	*phi,theta*	Longitude and latitude in the native coordinate system of the projection [deg].
`out`	*imgcrd*	Array of intermediate world coordinates. For celestial axes, imgcrd[][wcs.lng] and imgcrd[][wcs.lat] are the projected x-, and y-coordinates in pseudo "degrees". For quadcube projections with a **CUBEFACE** axis the face number is also returned in imgcrd[][wcs.cubeface]. For spectral axes, imgcrd[][wcs.spec] is the intermediate spectral coordinate, in SI units.
`out`	*pixcrd*	Array of pixel coordinates.
`out`	*stat*	Status return value for each coordinate: • 0: Success. 1+: A bit mask indicating invalid world coordinate element(s).

Returns

Status return value:

- 0: Success.
- 1: Null wcsprm pointer passed.
- 2: Memory allocation failed.
- 3: Linear transformation matrix is singular.
- 4: Inconsistent or unrecognized coordinate axis types.
- 5: Invalid parameter value.
- 6: Invalid coordinate transformation parameters.
- 7: Ill-conditioned coordinate transformation parameters.
- 9: One or more of the world coordinates were invalid, as indicated by the stat vector.

For returns $>$ 1, a detailed error message is set in wcsprm::err if enabled, see wcserr_enable().

19.23.4.18 wcsmix() `int wcsmix (`
```
         struct wcsprm * wcs,
         int mixpix,
         int mixcel,
         const double vspan[2],
         double vstep,
         int viter,
         double world[],
         double phi[],
         double theta[],
         double imgcrd[],
         double pixcrd[] )
```

wcsmix(), given either the celestial longitude or latitude plus an element of the pixel coordinate, solves for the remaining elements by iterating on the unknown celestial coordinate element using wcss2p(). Refer also to the notes below.

Parameters

in,out	wcs	Indices for the celestial coordinates obtained by parsing the wcsprm::ctype[].
in	mixpix	Which element of the pixel coordinate is given.
in	mixcel	Which element of the celestial coordinate is given: • 1: Celestial longitude is given in world[wcs.lng], latitude returned in world[wcs.lat]. • 2: Celestial latitude is given in world[wcs.lat], longitude returned in world[wcs.lng].
in	vspan	Solution interval for the celestial coordinate [deg]. The ordering of the two limits is irrelevant. Longitude ranges may be specified with any convenient normalization, for example [-120,+120] is the same as [240,480], except that the solution will be returned with the same normalization, i.e. lie within the interval specified.
in	vstep	Step size for solution search [deg]. If zero, a sensible, although perhaps non-optimal default will be used.
in	viter	If a solution is not found then the step size will be halved and the search recommenced. viter controls how many times the step size is halved. The allowed range is 5 - 10.
in,out	world	World coordinate elements. world[wcs.lng] and world[wcs.lat] are the celestial longitude and latitude [deg]. Which is given and which returned depends on the value of mixcel. All other elements are given.
out	phi,theta	Longitude and latitude in the native coordinate system of the projection [deg].
out	imgcrd	Image coordinate elements. imgcrd[wcs.lng] and imgcrd[wcs.lat] are the projected x-, and y-coordinates in pseudo "degrees".
in,out	pixcrd	Pixel coordinate. The element indicated by mixpix is given and the remaining elements are returned.

Returns

Status return value:

- 0: Success.

- 1: Null wcsprm pointer passed.

- 2: Memory allocation failed.

- 3: Linear transformation matrix is singular.

- 4: Inconsistent or unrecognized coordinate axis types.

- 5: Invalid parameter value.

- 6: Invalid coordinate transformation parameters.

- 7: Ill-conditioned coordinate transformation parameters.

- 10: Invalid world coordinate.

- 11: No solution found in the specified interval.

For returns > 1, a detailed error message is set in wcsprm::err if enabled, see wcserr_enable().

Notes:

1. Initially the specified solution interval is checked to see if it's a "crossing" interval. If it isn't, a search is made for a crossing solution by iterating on the unknown celestial coordinate starting at the upper limit of the solution interval and decrementing by the specified step size. A crossing is indicated if the trial value of the pixel coordinate steps through the value specified. If a crossing interval is found then the solution is determined by a modified form of "regula falsi" division of the crossing interval. If no crossing interval was found within the specified solution interval then a search is made for a "non-crossing" solution as may arise from a point of tangency. The process is complicated by having to make allowance for the discontinuities that occur in all map projections.

    Once one solution has been determined others may be found by subsequent invokations of **wcsmix**() with suitably restricted solution intervals.

    Note the circumstance that arises when the solution point lies at a native pole of a projection in which the pole is represented as a finite curve, for example the zenithals and conics. In such cases two or more valid solutions may exist but **wcsmix**() only ever returns one.

    Because of its generality **wcsmix**() is very compute-intensive. For compute-limited applications more efficient special-case solvers could be written for simple projections, for example non-oblique cylindrical projections.

19.23.4.19   wcsccs()   `int wcsccs (`
`        struct wcsprm * wcs,`
`        double lng2p1,`
`        double lat2p1,`
`        double lng1p2,`
`        const char * clng,`
`        const char * clat,`
`        const char * radesys,`
`        double equinox,`
`        const char * alt )`

wcsccs() changes the celestial coordinate system of a wcsprm struct. For example, from equatorial to galactic coordinates.

Parameters that define the spherical coordinate transformation, essentially being three Euler angles, must be provided. Thereby **wcsccs**() does not need prior knowledge of specific celestial coordinate systems. It also has the advantage of making it completely general.

Auxiliary members of the wcsprm struct relating to equatorial celestial coordinate systems may also be changed.

Only orthodox spherical coordinate systems are supported. That is, they must be right-handed, with latitude increasing from zero at the equator to +90 degrees at the pole. This precludes systems such as aziumuth and zenith distance, which, however, could be handled as negative azimuth and elevation.

PLEASE NOTE: Information in the wcsprm struct relating to the original coordinate system will be overwritten and therefore lost. If this is undesirable, invoke **wcsccs**() on a copy of the struct made with wcssub(). The wcsprm struct is reset on return with an explicit call to wcsset().

Parameters

in,out	*wcs*	Coordinate transformation parameters. Particular "values to be given" elements of the wcsprm struct are modified.
in	*lng2p1,lat2p1*	Longitude and latitude in the new celestial coordinate system of the pole (i.e. latitude +90) of the original system [deg]. See notes 1 and 2 below.
in	*lng1p2*	Longitude in the original celestial coordinate system of the pole (i.e. latitude +90) of the new system [deg]. See note 1 below.
in	*clng,clat*	Longitude and latitude identifiers of the new **CTYPE**ia celestial axis codes, without trailing dashes. For example, "RA" and "DEC" or "GLON" and "GLAT". Up to four characters are used, longer strings need not be null-terminated.
in	*radesys*	Used when transforming to equatorial coordinates, identified by clng == "RA" and clat = "DEC". May be set to the null pointer to preserve the current value. Up to 71 characters are used, longer strings need not be null-terminated. If the new coordinate system is anything other than equatorial, then wcsprm::radesys will be cleared.
in	*equinox*	Used when transforming to equatorial coordinates. May be set to zero to preserve the current value. If the new coordinate system is not equatorial, then wcsprm::equinox will be marked as undefined.
in	*alt*	Character code for alternate coordinate descriptions (i.e. the '**a**' in keyword names such as **CTYPE**ia). This is blank for the primary coordinate description, or one of the 26 upper-case letters, A-Z. May be set to the null pointer, or null string if no change is required.

Returns

Status return value:

- 0: Success.
- 1: Null wcsprm pointer passed.
- 12: Invalid subimage specification (no celestial axes).

Notes:

1. Follows the prescription given in WCS Paper II, Sect. 2.7 for changing celestial coordinates.

   The implementation takes account of indeterminacies that arise in that prescription in the particular cases where one of the poles of the new system is at the fiducial point, or one of them is at the native pole.

2. If lat2p1 == +90, i.e. where the poles of the two coordinate systems coincide, then the spherical coordinate transformation becomes a simple change in origin of longitude given by lng2 = lng1 + (lng2p1 - lng1p2 - 180), and lat2 = lat1, where (lng2,lat2) are coordinates in the new system, and (lng1,lat1) are coordinates in the original system.

   Likewise, if lat2p1 == -90, then lng2 = -lng1 + (lng2p1 + lng1p2), and lat2 = -lat1.

3. For example, if the original coordinate system is B1950 equatorial and the desired new coordinate system is galactic, then

   - (lng2p1,lat2p1) are the galactic coordinates of the B1950 celestial pole, defined by the IAU to be (123.↩ 0,+27.4), and lng1p2 is the B1950 right ascension of the galactic pole, defined as 192.25. Clearly these coordinates are fixed for a particular coordinate transformation.

   - (clng,clat) would be '**GLON**' and '**GLAT**', these being the FITS standard identifiers for galactic coordinates.

   - Since the new coordinate system is not equatorial, wcsprm::radesys and wcsprm::equinox will be cleared.

4. The coordinates required for some common transformations (obtained from `https://ned.ipac.↩ caltech.edu/coordinate_calculator`) are as follows:
```
(123.0000,+27.4000) galactic coordinates of B1950 celestial pole,
(192.2500,+27.4000) B1950 equatorial coordinates of galactic pole.
(122.9319,+27.1283) galactic coordinates of J2000 celestial pole,
(192.8595,+27.1283) J2000 equatorial coordinates of galactic pole.
(359.6774,+89.7217) B1950 equatorial coordinates of J2000 pole,
(180.3162,+89.7217) J2000 equatorial coordinates of B1950 pole.
(270.0000,+66.5542) B1950 equatorial coordinates of B1950 ecliptic pole,
( 90.0000,+66.5542) B1950 ecliptic coordinates of B1950 celestial pole.
(270.0000,+66.5607) J2000 equatorial coordinates of J2000 ecliptic pole,
( 90.0000,+66.5607) J2000 ecliptic coordinates of J2000 celestial pole.
( 26.7315,+15.6441) supergalactic coordinates of B1950 celestial pole,
(283.1894,+15.6441) B1950 equatorial coordinates of supergalactic pole.
( 26.4505,+15.7089) supergalactic coordinates of J2000 celestial pole,
(283.7542,+15.7089) J2000 equatorial coordinates of supergalactic pole.
```

19.23.4.20   wcssptr()   `int wcssptr (`
```
struct wcsprm * wcs,
int * i,
char ctype[9] )
```

wcssptr() translates the spectral axis in a wcsprm struct. For example, a '**FREQ**' axis may be translated into '**ZOPT-F2W**' and vice versa.

PLEASE NOTE: Information in the wcsprm struct relating to the original coordinate system will be overwritten and therefore lost. If this is undesirable, invoke **wcssptr**() on a copy of the struct made with wcssub(). The wcsprm struct is reset on return with an explicit call to wcsset().

Parameters

in,out	*wcs*	Coordinate transformation parameters.
in,out	*i*	Index of the spectral axis (0-relative). If given < 0 it will be set to the first spectral axis identified from the ctype[] keyvalues in the wcsprm struct.
in,out	*ctype*	Desired spectral **CTYPE**ia. Wildcarding may be used as for the ctypeS2 argument to spctrn() as described in the prologue of spc.h, i.e. if the final three characters are specified as "???", or if just the eighth character is specified as '?', the correct algorithm code will be substituted and returned.

Returns

Status return value:

- 0: Success.
- 1: Null wcsprm pointer passed.
- 2: Memory allocation failed.

- 3: Linear transformation matrix is singular.

- 4: Inconsistent or unrecognized coordinate axis types.

- 5: Invalid parameter value.

- 6: Invalid coordinate transformation parameters.

- 7: Ill-conditioned coordinate transformation parameters.

- 12: Invalid subimage specification (no spectral axis).

For returns > 1, a detailed error message is set in wcsprm::err if enabled, see wcserr_enable().

19.23.4.21 wcslib_version() const char ∗ wcslib_version (
int *vers[3]*)

19.23.5 Variable Documentation

19.23.5.1 wcs_errmsg const char ∗ wcs_errmsg[ ] [extern]

Error messages to match the status value returned from each function.

19.24 wcs.h

Go to the documentation of this file.
```
1  /*============================================================================
2    WCSLIB 7.11 - an implementation of the FITS WCS standard.
3    Copyright (C) 1995-2022, Mark Calabretta
4
5    This file is part of WCSLIB.
6
7    WCSLIB is free software: you can redistribute it and/or modify it under the
8    terms of the GNU Lesser General Public License as published by the Free
9    Software Foundation, either version 3 of the License, or (at your option)
10   any later version.
11
12   WCSLIB is distributed in the hope that it will be useful, but WITHOUT ANY
13   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
14   FOR A PARTICULAR PURPOSE.  See the GNU Lesser General Public License for
15   more details.
16
17   You should have received a copy of the GNU Lesser General Public License
18   along with WCSLIB.  If not, see http://www.gnu.org/licenses.
19
20   Author: Mark Calabretta, Australia Telescope National Facility, CSIRO.
21   http://www.atnf.csiro.au/people/Mark.Calabretta
22   $Id: wcs.h,v 7.11 2022/04/26 06:13:52 mcalabre Exp $
23  *============================================================================
24  *
25  * WCSLIB 7.11 - C routines that implement the FITS World Coordinate System
26  * (WCS) standard.  Refer to the README file provided with WCSLIB for an
27  * overview of the library.
28  *
29  *
30  * Summary of the wcs routines
31  * ---------------------------
32  * Routines in this suite implement the FITS World Coordinate System (WCS)
33  * standard which defines methods to be used for computing world coordinates
34  * from image pixel coordinates, and vice versa.  The standard, and proposed
35  * extensions for handling distortions, are described in
36  *
37  =    "Representations of world coordinates in FITS",
38  =    Greisen, E.W., & Calabretta, M.R. 2002, A&A, 395, 1061 (WCS Paper I)
```

```
 39  =
 40  =     "Representations of celestial coordinates in FITS",
 41  =     Calabretta, M.R., & Greisen, E.W. 2002, A&A, 395, 1077 (WCS Paper II)
 42  =
 43  =     "Representations of spectral coordinates in FITS",
 44  =     Greisen, E.W., Calabretta, M.R., Valdes, F.G., & Allen, S.L.
 45  =     2006, A&A, 446, 747 (WCS Paper III)
 46  =
 47  =     "Representations of distortions in FITS world coordinate systems",
 48  =     Calabretta, M.R. et al. (WCS Paper IV, draft dated 2004/04/22),
 49  =     available from http://www.atnf.csiro.au/people/Mark.Calabretta
 50  =
 51  =     "Mapping on the HEALPix grid",
 52  =     Calabretta, M.R., & Roukema, B.F. 2007, MNRAS, 381, 865 (WCS Paper V)
 53  =
 54  =     "Representing the 'Butterfly' Projection in FITS -- Projection Code XPH",
 55  =     Calabretta, M.R., & Lowe, S.R. 2013, PASA, 30, e050 (WCS Paper VI)
 56  =
 57  =     "Representations of time coordinates in FITS -
 58  =      Time and relative dimension in space",
 59  =     Rots, A.H., Bunclark, P.S., Calabretta, M.R., Allen, S.L.,
 60  =     Manchester, R.N., & Thompson, W.T. 2015, A&A, 574, A36 (WCS Paper VII)
 61  *
 62  * These routines are based on the wcsprm struct which contains all information
 63  * needed for the computations.  The struct contains some members that must be
 64  * set by the user, and others that are maintained by these routines, somewhat
 65  * like a C++ class but with no encapsulation.
 66  *
 67  * wcsnpv(), wcsnps(), wcsini(), wcsinit(), wcssub(), wcsfree(), and wcstrim(),
 68  * are provided to manage the wcsprm struct, wcssize() computes its total size
 69  * including allocated memory, and wcsprt() prints its contents.  Refer to the
 70  * description of the wcsprm struct for an explanation of the anticipated usage
 71  * of these routines.  wcscopy(), which does a deep copy of one wcsprm struct
 72  * to another, is defined as a preprocessor macro function that invokes
 73  * wcssub().
 74  *
 75  * wcsperr() prints the error message(s) (if any) stored in a wcsprm struct,
 76  * and the linprm, celprm, prjprm, spcprm, and tabprm structs that it contains.
 77  *
 78  * A setup routine, wcsset(), computes intermediate values in the wcsprm struct
 79  * from parameters in it that were supplied by the user.  The struct always
 80  * needs to be set up by wcsset() but this need not be called explicitly -
 81  * refer to the explanation of wcsprm::flag.
 82  *
 83  * wcsp2s() and wcss2p() implement the WCS world coordinate transformations.
 84  * In fact, they are high level driver routines for the WCS linear,
 85  * logarithmic, celestial, spectral and tabular transformation routines
 86  * described in lin.h, log.h, cel.h, spc.h and tab.h.
 87  *
 88  * Given either the celestial longitude or latitude plus an element of the
 89  * pixel coordinate a hybrid routine, wcsmix(), iteratively solves for the
 90  * unknown elements.
 91  *
 92  * wcsccs() changes the celestial coordinate system of a wcsprm struct, for
 93  * example, from equatorial to galactic, and wcssptr() translates the spectral
 94  * axis.  For example, a 'FREQ' axis may be translated into 'ZOPT-F2W' and vice
 95  * versa.
 96  *
 97  * wcslib_version() returns the WCSLIB version number.
 98  *
 99  * Quadcube projections:
100  * ---------------------
101  *    The quadcube projections (TSC, CSC, QSC) may be represented in FITS in
102  *    either of two ways:
103  *
104  *      a: The six faces may be laid out in one plane and numbered as follows:
105  *
106  =                                    0
107  =
108  =                             4  3  2  1  4  3  2
109  =
110  =                                    5
111  *
112  *         Faces 2, 3 and 4 may appear on one side or the other (or both).  The
113  *         world-to-pixel routines map faces 2, 3 and 4 to the left but the
114  *         pixel-to-world routines accept them on either side.
115  *
116  *      b: The "COBE" convention in which the six faces are stored in a
117  *         three-dimensional structure using a CUBEFACE axis indexed from
118  *         0 to 5 as above.
119  *
120  *    These routines support both methods; wcsset() determines which is being
121  *    used by the presence or absence of a CUBEFACE axis in ctype[].  wcsp2s()
122  *    and wcss2p() translate the CUBEFACE axis representation to the single
123  *    plane representation understood by the lower-level WCSLIB projection
124  *    routines.
125  *
```

```
126 *
127 * wcsnpv() - Memory allocation for PVi_ma
128 * ----------------------------------------
129 * wcsnpv() sets or gets the value of NPVMAX (default 64).  This global
130 * variable controls the number of pvcard structs, for holding PVi_ma
131 * keyvalues, that wcsini() should allocate space for.  It is also used by
132 * wcsinit() as the default value of npvmax.
133 *
134 * PLEASE NOTE: This function is not thread-safe.
135 *
136 * Given:
137 *   n          int       Value of NPVMAX; ignored if < 0.  Use a value less
138 *                        than zero to get the current value.
139 *
140 * Function return value:
141 *              int       Current value of NPVMAX.
142 *
143 *
144 * wcsnps() - Memory allocation for PSi_ma
145 * ----------------------------------------
146 * wcsnps() sets or gets the value of NPSMAX (default 8).  This global variable
147 * controls the number of pscard structs, for holding PSi_ma keyvalues, that
148 * wcsini() should allocate space for.  It is also used by wcsinit() as the
149 * default value of npsmax.
150 *
151 * PLEASE NOTE: This function is not thread-safe.
152 *
153 * Given:
154 *   n          int       Value of NPSMAX; ignored if < 0.  Use a value less
155 *                        than zero to get the current value.
156 *
157 * Function return value:
158 *              int       Current value of NPSMAX.
159 *
160 *
161 * wcsini() - Default constructor for the wcsprm struct
162 * -----------------------------------------------------
163 * wcsini() is a thin wrapper on wcsinit().  It invokes it with npvmax,
164 * npsmax, and ndpmax set to -1 which causes it to use the values of the
165 * global variables NDPMAX, NPSMAX, and NDPMAX.  It is thereby potentially
166 * thread-unsafe if these variables are altered dynamically via wcsnpv(),
167 * wcsnps(), and disndp().  Use wcsinit() for a thread-safe alternative in
168 * this case.
169 *
170 *
171 * wcsinit() - Default constructor for the wcsprm struct
172 * -----------------------------------------------------
173 * wcsinit() optionally allocates memory for arrays in a wcsprm struct and sets
174 * all members of the struct to default values.
175 *
176 * PLEASE NOTE: every wcsprm struct should be initialized by wcsinit(),
177 * possibly repeatedly.  On the first invokation, and only the first
178 * invokation, wcsprm::flag must be set to -1 to initialize memory management,
179 * regardless of whether wcsinit() will actually be used to allocate memory.
180 *
181 * Given:
182 *   alloc      int       If true, allocate memory unconditionally for the
183 *                        crpix, etc. arrays.  Please note that memory is never
184 *                        allocated by wcsinit() for the auxprm, tabprm, nor
185 *                        wtbarr structs.
186 *
187 *                        If false, it is assumed that pointers to these arrays
188 *                        have been set by the user except if they are null
189 *                        pointers in which case memory will be allocated for
190 *                        them regardless.  (In other words, setting alloc true
191 *                        saves having to initalize these pointers to zero.)
192 *
193 *   naxis      int       The number of world coordinate axes.  This is used to
194 *                        determine the length of the various wcsprm vectors and
195 *                        matrices and therefore the amount of memory to
196 *                        allocate for them.
197 *
198 * Given and returned:
199 *   wcs        struct wcsprm*
200 *                        Coordinate transformation parameters.
201 *
202 *                        Note that, in order to initialize memory management,
203 *                        wcsprm::flag should be set to -1 when wcs is
204 *                        initialized for the first time (memory leaks may
205 *                        result if it had already been initialized).
206 *
207 * Given:
208 *   npvmax     int       The number of PVi_ma keywords to allocate space for.
209 *                        If set to -1, the value of the global variable NPVMAX
210 *                        will be used.  This is potentially thread-unsafe if
211 *                        wcsnpv() is being used dynamically to alter its value.
212 *
```

```
213 *    npsmax     int        The number of PSi_ma keywords to allocate space for.
214 *                          If set to -1, the value of the global variable NPSMAX
215 *                          will be used.  This is potentially thread-unsafe if
216 *                          wcsnps() is being used dynamically to alter its value.
217 *
218 *    ndpmax     int        The number of DPja or DQia keywords to allocate space
219 *                          for.  If set to -1, the value of the global variable
220 *                          NDPMAX will be used.  This is potentially
221 *                          thread-unsafe if disndp() is being used dynamically to
222 *                          alter its value.
223 *
224 * Function return value:
225 *               int        Status return value:
226 *                            0: Success.
227 *                            1: Null wcsprm pointer passed.
228 *                            2: Memory allocation failed.
229 *
230 *                          For returns > 1, a detailed error message is set in
231 *                          wcsprm::err if enabled, see wcserr_enable().
232 *
233 *
234 * wcsauxi() - Default constructor for the auxprm struct
235 * -----------------------------------------------------
236 * wcsauxi() optionally allocates memory for an auxprm struct, attaches it to
237 * wcsprm, and sets all members of the struct to default values.
238 *
239 * Given:
240 *    alloc      int        If true, allocate memory unconditionally for the
241 *                          auxprm struct.
242 *
243 *                          If false, it is assumed that wcsprm::aux has already
244 *                          been set to point to an auxprm struct, in which case
245 *                          the user is responsible for managing that memory.
246 *                          However, if wcsprm::aux is a null pointer, memory will
247 *                          be allocated regardless.  (In other words, setting
248 *                          alloc true saves having to initalize the pointer to
249 *                          zero.)
250 *
251 * Given and returned:
252 *    wcs        struct wcsprm*
253 *                          Coordinate transformation parameters.
254 *
255 * Function return value:
256 *               int        Status return value:
257 *                            0: Success.
258 *                            1: Null wcsprm pointer passed.
259 *                            2: Memory allocation failed.
260 *
261 *
262 * wcssub() - Subimage extraction routine for the wcsprm struct
263 * -----------------------------------------------------------
264 * wcssub() extracts the coordinate description for a subimage from a wcsprm
265 * struct.  It does a deep copy, using wcsinit() to allocate memory for its
266 * arrays if required.  Only the "information to be provided" part of the
267 * struct is extracted.  Consequently, wcsset() need not have been, and won't
268 * be invoked on the struct from which the subimage is extracted.  A call to
269 * wcsset() is required to set up the subimage struct.
270 *
271 * The world coordinate system of the subimage must be separable in the sense
272 * that the world coordinates at any point in the subimage must depend only on
273 * the pixel coordinates of the axes extracted.  In practice, this means that
274 * the linear transformation matrix of the original image must not contain
275 * non-zero off-diagonal terms that associate any of the subimage axes with any
276 * of the non-subimage axes.  Likewise, if any distortions are associated with
277 * the subimage axes, they must not depend on any of the axes that are not
278 * being extracted.
279 *
280 * Note that while the required elements of the tabprm array are extracted, the
281 * wtbarr array is not.  (Thus it is not appropriate to call wcssub() after
282 * wcstab() but before filling the tabprm structs - refer to wcshdr.h.)
283 *
284 * wcssub() can also add axes to a wcsprm struct.  The new axes will be created
285 * using the defaults set by wcsinit() which produce a simple, unnamed, linear
286 * axis with world coordinate equal to the pixel coordinate.  These default
287 * values can be changed afterwards, before invoking wcsset().
288 *
289 * Given:
290 *    alloc      int        If true, allocate memory for the crpix, etc. arrays in
291 *                          the destination.  Otherwise, it is assumed that
292 *                          pointers to these arrays have been set by the user
293 *                          except if they are null pointers in which case memory
294 *                          will be allocated for them regardless.
295 *
296 *    wcssrc     const struct wcsprm*
297 *                          Struct to extract from.
298 *
299 * Given and returned:
```

```
300 *   nsub       int*
301 *   axes       int[]      Vector of length *nsub containing the image axis
302 *                         numbers (1-relative) to extract.  Order is
303 *                         significant; axes[0] is the axis number of the input
304 *                         image that corresponds to the first axis in the
305 *                         subimage, etc.
306 *
307 *                         Use an axis number of 0 to create a new axis using
308 *                         the defaults set by wcsinit().  They can be changed
309 *                         later.
310 *
311 *                         nsub (the pointer) may be set to zero, and so also may
312 *                         *nsub, which is interpreted to mean all axes in the
313 *                         input image; the number of axes will be returned if
314 *                         nsub != 0x0.  axes itself (the pointer) may be set to
315 *                         zero to indicate the first *nsub axes in their
316 *                         original order.
317 *
318 *                         Set both nsub (or *nsub) and axes to zero to do a deep
319 *                         copy of one wcsprm struct to another.
320 *
321 *                         Subimage extraction by coordinate axis type may be
322 *                         done by setting the elements of axes[] to the
323 *                         following special preprocessor macro values:
324 *
325 *                           WCSSUB_LONGITUDE: Celestial longitude.
326 *                           WCSSUB_LATITUDE:  Celestial latitude.
327 *                           WCSSUB_CUBEFACE:  Quadcube CUBEFACE axis.
328 *                           WCSSUB_SPECTRAL:  Spectral axis.
329 *                           WCSSUB_STOKES:    Stokes axis.
330 *                           WCSSUB_TIME:      Time axis.
331 *
332 *                         Refer to the notes (below) for further usage examples.
333 *
334 *                         On return, *nsub will be set to the number of axes in
335 *                         the subimage; this may be zero if there were no axes
336 *                         of the required type(s) (in which case no memory will
337 *                         be allocated).  axes[] will contain the axis numbers
338 *                         that were extracted, or 0 for newly created axes.  The
339 *                         vector length must be sufficient to contain all axis
340 *                         numbers.  No checks are performed to verify that the
341 *                         coordinate axes are consistent, this is done by
342 *                         wcsset().
343 *
344 *   wcsdst     struct wcsprm*
345 *                         Struct describing the subimage.  wcsprm::flag should
346 *                         be set to -1 if wcsdst was not previously initialized
347 *                         (memory leaks may result if it was previously
348 *                         initialized).
349 *
350 * Function return value:
351 *              int        Status return value:
352 *                           0: Success.
353 *                           1: Null wcsprm pointer passed.
354 *                           2: Memory allocation failed.
355 *                          12: Invalid subimage specification.
356 *                          13: Non-separable subimage coordinate system.
357 *
358 *                         For returns > 1, a detailed error message is set in
359 *                         wcsprm::err if enabled, see wcserr_enable().
360 *
361 * Notes:
362 *   1: Combinations of subimage axes of particular types may be extracted in
363 *      the same order as they occur in the input image by combining
364 *      preprocessor codes, for example
365 *
366 =          *nsub = 1;
367 =          axes[0] = WCSSUB_LONGITUDE | WCSSUB_LATITUDE | WCSSUB_SPECTRAL;
368 *
369 *      would extract the longitude, latitude, and spectral axes in the same
370 *      order as the input image.  If one of each were present, *nsub = 3 would
371 *      be returned.
372 *
373 *      For convenience, WCSSUB_CELESTIAL is defined as the combination
374 *      WCSSUB_LONGITUDE | WCSSUB_LATITUDE | WCSSUB_CUBEFACE.
375 *
376 *      The codes may also be negated to extract all but the types specified,
377 *      for example
378 *
379 =          *nsub = 4;
380 =          axes[0] = WCSSUB_LONGITUDE;
381 =          axes[1] = WCSSUB_LATITUDE;
382 =          axes[2] = WCSSUB_CUBEFACE;
383 =          axes[3] = -(WCSSUB_SPECTRAL | WCSSUB_STOKES);
384 *
385 *      The last of these specifies all axis types other than spectral or
386 *      Stokes.  Extraction is done in the order specified by axes[] a
```

```
387 *      longitude axis (if present) would be extracted first (via axes[0]) and
388 *      not subsequently (via axes[3]).  Likewise for the latitude and cubeface
389 *      axes in this example.
390 *
391 *      From the foregoing, it is apparent that the value of *nsub returned may
392 *      be less than or greater than that given.  However, it will never exceed
393 *      the number of axes in the input image (plus the number of newly-created
394 *      axes if any were specified on input).
395 *
396 *
397 * wcscompare() - Compare two wcsprm structs for equality
398 * ------------------------------------------------------
399 * wcscompare() compares two wcsprm structs for equality.
400 *
401 * Given:
402 *   cmp       int       A bit field controlling the strictness of the
403 *                       comparison.  When 0, all fields must be identical.
404 *
405 *                       The following constants may be or'ed together to
406 *                       relax the comparison:
407 *                         WCSCOMPARE_ANCILLARY: Ignore ancillary keywords
408 *                           that don't change the WCS transformation, such
409 *                           as DATE-OBS or EQUINOX.
410 *                         WCSCOMPARE_TILING: Ignore integral differences in
411 *                           CRPIXja.  This is the 'tiling' condition, where
412 *                           two WCSes cover different regions of the same
413 *                           map projection and align on the same map grid.
414 *                         WCSCOMPARE_CRPIX: Ignore any differences at all in
415 *                           CRPIXja.  The two WCSes cover different regions
416 *                           of the same map projection but may not align on
417 *                           the same map grid.  Overrides WCSCOMPARE_TILING.
418 *
419 *   tol       double    Tolerance for comparison of floating-point values.
420 *                       For example, for tol == 1e-6, all floating-point
421 *                       values in the structs must be equal to the first 6
422 *                       decimal places.  A value of 0 implies exact equality.
423 *
424 *   wcs1      const struct wcsprm*
425 *                       The first wcsprm struct to compare.
426 *
427 *   wcs2      const struct wcsprm*
428 *                       The second wcsprm struct to compare.
429 *
430 * Returned:
431 *   equal     int*      Non-zero when the given structs are equal.
432 *
433 * Function return value:
434 *             int       Status return value:
435 *                         0: Success.
436 *                         1: Null pointer passed.
437 *
438 *
439 * wcscopy() macro - Copy routine for the wcsprm struct
440 * ----------------------------------------------------
441 * wcscopy() does a deep copy of one wcsprm struct to another.  As of
442 * WCSLIB 3.6, it is implemented as a preprocessor macro that invokes
443 * wcssub() with the nsub and axes pointers both set to zero.
444 *
445 *
446 * wcsfree() - Destructor for the wcsprm struct
447 * --------------------------------------------
448 * wcsfree() frees memory allocated for the wcsprm arrays by wcsinit() and/or
449 * wcsset().  wcsinit() records the memory it allocates and wcsfree() will only
450 * attempt to free this.
451 *
452 * PLEASE NOTE: wcsfree() must not be invoked on a wcsprm struct that was not
453 * initialized by wcsinit().
454 *
455 * Given and returned:
456 *   wcs       struct wcsprm*
457 *                       Coordinate transformation parameters.
458 *
459 * Function return value:
460 *             int       Status return value:
461 *                         0: Success.
462 *                         1: Null wcsprm pointer passed.
463 *
464 *
465 * wcstrim() - Free unused arrays in the wcsprm struct
466 * ---------------------------------------------------
467 * wcstrim() frees memory allocated by wcsinit() for arrays in the wcsprm
468 * struct that remains unused after it has been set up by wcsset().
469 *
470 * The free'd array members are associated with FITS WCS keyrecords that are
471 * rarely used and usually just bloat the struct: wcsprm::crota, wcsprm::colax,
472 * wcsprm::cname, wcsprm::crder, wcsprm::csyer, wcsprm::czphs, and
473 * wcsprm::cperi.  If unused, wcsprm::pv, wcsprm::ps, and wcsprm::cd are also
```

```
474 * freed.
475 *
476 * Once these arrays have been freed, a test such as
477 =
478 =         if (!undefined(wcs->cname[i])) {...}
479 =
480 * must be protected as follows
481 =
482 =         if (wcs->cname && !undefined(wcs->cname[i])) {...}
483 =
484 * In addition, if wcsprm::npv is non-zero but less than wcsprm::npvmax, then
485 * the unused space in wcsprm::pv will be recovered (using realloc()).
486 * Likewise for wcsprm::ps.
487 *
488 * Given and returned:
489 *   wcs       struct wcsprm*
490 *                     Coordinate transformation parameters.
491 *
492 * Function return value:
493 *             int       Status return value:
494 *                        0: Success.
495 *                        1: Null wcsprm pointer passed.
496 *                       14: wcsprm struct is unset.
497 *
498 *
499 * wcssize() - Compute the size of a wcsprm struct
500 * -----------------------------------------------
501 * wcssize() computes the full size of a wcsprm struct, including allocated
502 * memory.
503 *
504 * Given:
505 *   wcs       const struct wcsprm*
506 *                     Coordinate transformation parameters.
507 *
508 *                     If NULL, the base size of the struct and the allocated
509 *                     size are both set to zero.
510 *
511 * Returned:
512 *   sizes     int[2]   The first element is the base size of the struct as
513 *                     returned by sizeof(struct wcsprm).  The second element
514 *                     is the total allocated size, in bytes, assuming that
515 *                     the allocation was done by wcsini().  This figure
516 *                     includes memory allocated for members of constituent
517 *                     structs, such as wcsprm::lin.
518 *
519 *                     It is not an error for the struct not to have been set
520 *                     up via wcsset(), which normally results in additional
521 *                     memory allocation.
522 *
523 * Function return value:
524 *             int       Status return value:
525 *                        0: Success.
526 *
527 *
528 * auxsize() - Compute the size of a auxprm struct
529 * -----------------------------------------------
530 * auxsize() computes the full size of a auxprm struct, including allocated
531 * memory.
532 *
533 * Given:
534 *   aux       const struct auxprm*
535 *                     Auxiliary coordinate information.
536 *
537 *                     If NULL, the base size of the struct and the allocated
538 *                     size are both set to zero.
539 *
540 * Returned:
541 *   sizes     int[2]   The first element is the base size of the struct as
542 *                     returned by sizeof(struct auxprm).  The second element
543 *                     is the total allocated size, in bytes, currently zero.
544 *
545 * Function return value:
546 *             int       Status return value:
547 *                        0: Success.
548 *
549 *
550 * wcsprt() - Print routine for the wcsprm struct
551 * ----------------------------------------------
552 * wcsprt() prints the contents of a wcsprm struct using wcsprintf().  Mainly
553 * intended for diagnostic purposes.
554 *
555 * Given:
556 *   wcs       const struct wcsprm*
557 *                     Coordinate transformation parameters.
558 *
559 * Function return value:
560 *             int       Status return value:
```

```
561 *                               0: Success.
562 *                               1: Null wcsprm pointer passed.
563 *
564 *
565 * wcsperr() - Print error messages from a wcsprm struct
566 * -----------------------------------------------------
567 * wcsperr() prints the error message(s), if any, stored in a wcsprm struct,
568 * and the linprm, celprm, prjprm, spcprm, and tabprm structs that it contains.
569 * If there are no errors then nothing is printed.  It uses wcserr_prt(), q.v.
570 *
571 * Given:
572 *   wcs       const struct wcsprm*
573 *                       Coordinate transformation parameters.
574 *
575 *   prefix    const char *
576 *                       If non-NULL, each output line will be prefixed with
577 *                       this string.
578 *
579 * Function return value:
580 *             int       Status return value:
581 *                         0: Success.
582 *                         1: Null wcsprm pointer passed.
583 *
584 *
585 * wcsbchk() - Enable/disable bounds checking
586 * ------------------------------------------
587 * wcsbchk() is used to control bounds checking in the projection routines.
588 * Note that wcsset() always enables bounds checking.  wcsbchk() will invoke
589 * wcsset() on the wcsprm struct beforehand if necessary.
590 *
591 * Given and returned:
592 *   wcs       struct wcsprm*
593 *                       Coordinate transformation parameters.
594 *
595 * Given:
596 *   bounds    int       If bounds&1 then enable strict bounds checking for the
597 *                       spherical-to-Cartesian (s2x) transformation for the
598 *                       AZP, SZP, TAN, SIN, ZPN, and COP projections.
599 *
600 *                       If bounds&2 then enable strict bounds checking for the
601 *                       Cartesian-to-spherical (x2s) transformation for the
602 *                       HPX and XPH projections.
603 *
604 *                       If bounds&4 then enable bounds checking on the native
605 *                       coordinates returned by the Cartesian-to-spherical
606 *                       (x2s) transformations using prjchk().
607 *
608 *                       Zero it to disable all checking.
609 *
610 * Function return value:
611 *             int       Status return value:
612 *                         0: Success.
613 *                         1: Null wcsprm pointer passed.
614 *
615 *
616 * wcsset() - Setup routine for the wcsprm struct
617 * ----------------------------------------------
618 * wcsset() sets up a wcsprm struct according to information supplied within
619 * it (refer to the description of the wcsprm struct).
620 *
621 * wcsset() recognizes the NCP projection and converts it to the equivalent SIN
622 * projection and likewise translates GLS into SFL.  It also translates the
623 * AIPS spectral types ('FREQ-LSR', 'FELO-HEL', etc.), possibly changing the
624 * input header keywords wcsprm::ctype and/or wcsprm::specsys if necessary.
625 *
626 * Note that this routine need not be called directly; it will be invoked by
627 * wcsp2s() and wcss2p() if the wcsprm::flag is anything other than a
628 * predefined magic value.
629 *
630 * Given and returned:
631 *   wcs       struct wcsprm*
632 *                       Coordinate transformation parameters.
633 *
634 * Function return value:
635 *             int       Status return value:
636 *                         0: Success.
637 *                         1: Null wcsprm pointer passed.
638 *                         2: Memory allocation failed.
639 *                         3: Linear transformation matrix is singular.
640 *                         4: Inconsistent or unrecognized coordinate axis
641 *                            types.
642 *                         5: Invalid parameter value.
643 *                         6: Invalid coordinate transformation parameters.
644 *                         7: Ill-conditioned coordinate transformation
645 *                            parameters.
646 *
647 *                       For returns > 1, a detailed error message is set in
```

```
648 *                      wcsprm::err if enabled, see wcserr_enable().
649 *
650 * Notes:
651 *   1: wcsset() always enables strict bounds checking in the projection
652 *      routines (via a call to prjini()).  Use wcsbchk() to modify
653 *      bounds-checking after wcsset() is invoked.
654 *
655 *
656 * wcsp2s() - Pixel-to-world transformation
657 * ----------------------------------------
658 * wcsp2s() transforms pixel coordinates to world coordinates.
659 *
660 * Given and returned:
661 *   wcs       struct wcsprm*
662 *                      Coordinate transformation parameters.
663 *
664 * Given:
665 *   ncoord,
666 *   nelem     int      The number of coordinates, each of vector length
667 *                      nelem but containing wcs.naxis coordinate elements.
668 *                      Thus nelem must equal or exceed the value of the
669 *                      NAXIS keyword unless ncoord == 1, in which case nelem
670 *                      is not used.
671 *
672 *   pixcrd    const double[ncoord][nelem]
673 *                      Array of pixel coordinates.
674 *
675 * Returned:
676 *   imgcrd    double[ncoord][nelem]
677 *                      Array of intermediate world coordinates.  For
678 *                      celestial axes, imgcrd[][wcs.lng] and
679 *                      imgcrd[][wcs.lat] are the projected x-, and
680 *                      y-coordinates in pseudo "degrees".  For spectral
681 *                      axes, imgcrd[][wcs.spec] is the intermediate spectral
682 *                      coordinate, in SI units.
683 *
684 *   phi,theta double[ncoord]
685 *                      Longitude and latitude in the native coordinate system
686 *                      of the projection [deg].
687 *
688 *   world     double[ncoord][nelem]
689 *                      Array of world coordinates.  For celestial axes,
690 *                      world[][wcs.lng] and world[][wcs.lat] are the
691 *                      celestial longitude and latitude [deg].  For
692 *                      spectral axes, imgcrd[][wcs.spec] is the intermediate
693 *                      spectral coordinate, in SI units.
694 *
695 *   stat      int[ncoord]
696 *                      Status return value for each coordinate:
697 *                        0: Success.
698 *                       1+: A bit mask indicating invalid pixel coordinate
699 *                           element(s).
700 *
701 * Function return value:
702 *             int       Status return value:
703 *                        0: Success.
704 *                        1: Null wcsprm pointer passed.
705 *                        2: Memory allocation failed.
706 *                        3: Linear transformation matrix is singular.
707 *                        4: Inconsistent or unrecognized coordinate axis
708 *                           types.
709 *                        5: Invalid parameter value.
710 *                        6: Invalid coordinate transformation parameters.
711 *                        7: Ill-conditioned coordinate transformation
712 *                           parameters.
713 *                        8: One or more of the pixel coordinates were
714 *                           invalid, as indicated by the stat vector.
715 *
716 *                      For returns > 1, a detailed error message is set in
717 *                      wcsprm::err if enabled, see wcserr_enable().
718 *
719 *
720 * wcss2p() - World-to-pixel transformation
721 * ----------------------------------------
722 * wcss2p() transforms world coordinates to pixel coordinates.
723 *
724 * Given and returned:
725 *   wcs       struct wcsprm*
726 *                      Coordinate transformation parameters.
727 *
728 * Given:
729 *   ncoord,
730 *   nelem     int      The number of coordinates, each of vector length nelem
731 *                      but containing wcs.naxis coordinate elements.  Thus
732 *                      nelem must equal or exceed the value of the NAXIS
733 *                      keyword unless ncoord == 1, in which case nelem is not
734 *                      used.
```

```
735 *
736 *   world     const double[ncoord][nelem]
737 *                     Array of world coordinates.  For celestial axes,
738 *                     world[][wcs.lng] and world[][wcs.lat] are the
739 *                     celestial longitude and latitude [deg]. For spectral
740 *                     axes, world[][wcs.spec] is the spectral coordinate, in
741 *                     SI units.
742 *
743 * Returned:
744 *   phi,theta double[ncoord]
745 *                     Longitude and latitude in the native coordinate
746 *                     system of the projection [deg].
747 *
748 *   imgcrd    double[ncoord][nelem]
749 *                     Array of intermediate world coordinates.  For
750 *                     celestial axes, imgcrd[][wcs.lng] and
751 *                     imgcrd[][wcs.lat] are the projected x-, and
752 *                     y-coordinates in pseudo "degrees".  For quadcube
753 *                     projections with a CUBEFACE axis the face number is
754 *                     also returned in imgcrd[][wcs.cubeface].  For
755 *                     spectral axes, imgcrd[][wcs.spec] is the intermediate
756 *                     spectral coordinate, in SI units.
757 *
758 *   pixcrd    double[ncoord][nelem]
759 *                     Array of pixel coordinates.
760 *
761 *   stat      int[ncoord]
762 *                     Status return value for each coordinate:
763 *                       0: Success.
764 *                      1+: A bit mask indicating invalid world coordinate
765 *                          element(s).
766 *
767 * Function return value:
768 *             int       Status return value:
769 *                         0: Success.
770 *                         1: Null wcsprm pointer passed.
771 *                         2: Memory allocation failed.
772 *                         3: Linear transformation matrix is singular.
773 *                         4: Inconsistent or unrecognized coordinate axis
774 *                            types.
775 *                         5: Invalid parameter value.
776 *                         6: Invalid coordinate transformation parameters.
777 *                         7: Ill-conditioned coordinate transformation
778 *                            parameters.
779 *                         9: One or more of the world coordinates were
780 *                            invalid, as indicated by the stat vector.
781 *
782 *                       For returns > 1, a detailed error message is set in
783 *                       wcsprm::err if enabled, see wcserr_enable().
784 *
785 *
786 * wcsmix() - Hybrid coordinate transformation
787 * -------------------------------------------
788 * wcsmix(), given either the celestial longitude or latitude plus an element
789 * of the pixel coordinate, solves for the remaining elements by iterating on
790 * the unknown celestial coordinate element using wcss2p().  Refer also to the
791 * notes below.
792 *
793 * Given and returned:
794 *   wcs       struct wcsprm*
795 *                     Indices for the celestial coordinates obtained
796 *                     by parsing the wcsprm::ctype[].
797 *
798 * Given:
799 *   mixpix    int       Which element of the pixel coordinate is given.
800 *
801 *   mixcel    int       Which element of the celestial coordinate is given:
802 *                         1: Celestial longitude is given in
803 *                            world[wcs.lng], latitude returned in
804 *                            world[wcs.lat].
805 *                         2: Celestial latitude is given in
806 *                            world[wcs.lat], longitude returned in
807 *                            world[wcs.lng].
808 *
809 *   vspan     const double[2]
810 *                     Solution interval for the celestial coordinate [deg].
811 *                     The ordering of the two limits is irrelevant.
812 *                     Longitude ranges may be specified with any convenient
813 *                     normalization, for example [-120,+120] is the same as
814 *                     [240,480], except that the solution will be returned
815 *                     with the same normalization, i.e. lie within the
816 *                     interval specified.
817 *
818 *   vstep     const double
819 *                     Step size for solution search [deg].  If zero, a
820 *                     sensible, although perhaps non-optimal default will be
821 *                     used.
```

```
822 *
823 *   viter     int       If a solution is not found then the step size will be
824 *                       halved and the search recommenced.  viter controls how
825 *                       many times the step size is halved.  The allowed range
826 *                       is 5 - 10.
827 *
828 * Given and returned:
829 *   world     double[naxis]
830 *                       World coordinate elements.  world[wcs.lng] and
831 *                       world[wcs.lat] are the celestial longitude and
832 *                       latitude [deg].  Which is given and which returned
833 *                       depends on the value of mixcel.  All other elements
834 *                       are given.
835 *
836 * Returned:
837 *   phi,theta double[naxis]
838 *                       Longitude and latitude in the native coordinate
839 *                       system of the projection [deg].
840 *
841 *   imgcrd    double[naxis]
842 *                       Image coordinate elements.  imgcrd[wcs.lng] and
843 *                       imgcrd[wcs.lat] are the projected x-, and
844 *                       y-coordinates in pseudo "degrees".
845 *
846 * Given and returned:
847 *   pixcrd    double[naxis]
848 *                       Pixel coordinate.  The element indicated by mixpix is
849 *                       given and the remaining elements are returned.
850 *
851 * Function return value:
852 *             int       Status return value:
853 *                        0: Success.
854 *                        1: Null wcsprm pointer passed.
855 *                        2: Memory allocation failed.
856 *                        3: Linear transformation matrix is singular.
857 *                        4: Inconsistent or unrecognized coordinate axis
858 *                           types.
859 *                        5: Invalid parameter value.
860 *                        6: Invalid coordinate transformation parameters.
861 *                        7: Ill-conditioned coordinate transformation
862 *                           parameters.
863 *                       10: Invalid world coordinate.
864 *                       11: No solution found in the specified interval.
865 *
866 *                       For returns > 1, a detailed error message is set in
867 *                       wcsprm::err if enabled, see wcserr_enable().
868 *
869 * Notes:
870 *   1: Initially the specified solution interval is checked to see if it's a
871 *      "crossing" interval.  If it isn't, a search is made for a crossing
872 *      solution by iterating on the unknown celestial coordinate starting at
873 *      the upper limit of the solution interval and decrementing by the
874 *      specified step size.  A crossing is indicated if the trial value of the
875 *      pixel coordinate steps through the value specified.  If a crossing
876 *      interval is found then the solution is determined by a modified form of
877 *      "regula falsi" division of the crossing interval.  If no crossing
878 *      interval was found within the specified solution interval then a search
879 *      is made for a "non-crossing" solution as may arise from a point of
880 *      tangency.  The process is complicated by having to make allowance for
881 *      the discontinuities that occur in all map projections.
882 *
883 *      Once one solution has been determined others may be found by subsequent
884 *      invokations of wcsmix() with suitably restricted solution intervals.
885 *
886 *      Note the circumstance that arises when the solution point lies at a
887 *      native pole of a projection in which the pole is represented as a
888 *      finite curve, for example the zenithals and conics.  In such cases two
889 *      or more valid solutions may exist but wcsmix() only ever returns one.
890 *
891 *      Because of its generality wcsmix() is very compute-intensive.  For
892 *      compute-limited applications more efficient special-case solvers could
893 *      be written for simple projections, for example non-oblique cylindrical
894 *      projections.
895 *
896 *
897 * wcsccs() - Change celestial coordinate system
898 * ---------------------------------------------
899 * wcsccs() changes the celestial coordinate system of a wcsprm struct.  For
900 * example, from equatorial to galactic coordinates.
901 *
902 * Parameters that define the spherical coordinate transformation, essentially
903 * being three Euler angles, must be provided.  Thereby wcsccs() does not need
904 * prior knowledge of specific celestial coordinate systems.  It also has the
905 * advantage of making it completely general.
906 *
907 * Auxiliary members of the wcsprm struct relating to equatorial celestial
908 * coordinate systems may also be changed.
```

```
909 *
910 * Only orthodox spherical coordinate systems are supported.  That is, they
911 * must be right-handed, with latitude increasing from zero at the equator to
912 * +90 degrees at the pole.  This precludes systems such as aziumuth and zenith
913 * distance, which, however, could be handled as negative azimuth and
914 * elevation.
915 *
916 * PLEASE NOTE: Information in the wcsprm struct relating to the original
917 * coordinate system will be overwritten and therefore lost.  If this is
918 * undesirable, invoke wcsccs() on a copy of the struct made with wcssub().
919 * The wcsprm struct is reset on return with an explicit call to wcsset().
920 *
921 * Given and returned:
922 *   wcs        struct wcsprm*
923 *                        Coordinate transformation parameters.  Particular
924 *                        "values to be given" elements of the wcsprm struct
925 *                        are modified.
926 *
927 * Given:
928 *   lng2p1,
929 *   lat2p1    double   Longitude and latitude in the new celestial coordinate
930 *                        system of the pole (i.e. latitude +90) of the original
931 *                        system [deg].  See notes 1 and 2 below.
932 *
933 *   lng1p2    double   Longitude in the original celestial coordinate system
934 *                        of the pole (i.e. latitude +90) of the new system
935 *                        [deg].  See note 1 below.
936 *
937 *   clng,clat const char*
938 *                        Longitude and latitude identifiers of the new CTYPEia
939 *                        celestial axis codes, without trailing dashes.  For
940 *                        example, "RA" and "DEC" or "GLON" and "GLAT".  Up to
941 *                        four characters are used, longer strings need not be
942 *                        null-terminated.
943 *
944 *   radesys   const char*
945 *                        Used when transforming to equatorial coordinates,
946 *                        identified by clng == "RA" and clat = "DEC".  May be
947 *                        set to the null pointer to preserve the current value.
948 *                        Up to 71 characters are used, longer strings need not
949 *                        be null-terminated.
950 *
951 *                        If the new coordinate system is anything other than
952 *                        equatorial, then wcsprm::radesys will be cleared.
953 *
954 *   equinox   double   Used when transforming to equatorial coordinates.  May
955 *                        be set to zero to preserve the current value.
956 *
957 *                        If the new coordinate system is not equatorial, then
958 *                        wcsprm::equinox will be marked as undefined.
959 *
960 *   alt       const char*
961 *                        Character code for alternate coordinate descriptions
962 *                        (i.e. the 'a' in keyword names such as CTYPEia).  This
963 *                        is blank for the primary coordinate description, or
964 *                        one of the 26 upper-case letters, A-Z.  May be set to
965 *                        the null pointer, or null string if no change is
966 *                        required.
967 *
968 * Function return value:
969 *             int      Status return value:
970 *                          0: Success.
971 *                          1: Null wcsprm pointer passed.
972 *                         12: Invalid subimage specification (no celestial
973 *                             axes).
974 *
975 * Notes:
976 *   1: Follows the prescription given in WCS Paper II, Sect. 2.7 for changing
977 *      celestial coordinates.
978 *
979 *      The implementation takes account of indeterminacies that arise in that
980 *      prescription in the particular cases where one of the poles of the new
981 *      system is at the fiducial point, or one of them is at the native pole.
982 *
983 *   2: If lat2p1 == +90, i.e. where the poles of the two coordinate systems
984 *      coincide, then the spherical coordinate transformation becomes a simple
985 *      change in origin of longitude given by
986 *      lng2 = lng1 + (lng2p1 - lng1p2 - 180), and lat2 = lat1, where
987 *      (lng2,lat2) are coordinates in the new system, and (lng1,lat1) are
988 *      coordinates in the original system.
989 *
990 *      Likewise, if lat2p1 == -90, then lng2 = -lng1 + (lng2p1 + lng1p2), and
991 *      lat2 = -lat1.
992 *
993 *   3: For example, if the original coordinate system is B1950 equatorial and
994 *      the desired new coordinate system is galactic, then
995 *
```

```
 996 *       - (lng2p1,lat2p1) are the galactic coordinates of the B1950 celestial
 997 *         pole, defined by the IAU to be (123.0,+27.4), and lng1p2 is the B1950
 998 *         right ascension of the galactic pole, defined as 192.25.  Clearly
 999 *         these coordinates are fixed for a particular coordinate
1000 *          transformation.
1001 *
1002 *       - (clng,clat) would be 'GLON' and 'GLAT', these being the FITS standard
1003 *         identifiers for galactic coordinates.
1004 *
1005 *       - Since the new coordinate system is not equatorial, wcsprm::radesys
1006 *         and wcsprm::equinox will be cleared.
1007 *
1008 *    4. The coordinates required for some common transformations (obtained from
1009 *       https://ned.ipac.caltech.edu/coordinate_calculator) are as follows:
1010 *
1011 =       (123.0000,+27.4000) galactic coordinates of B1950 celestial pole,
1012 =       (192.2500,+27.4000) B1950 equatorial coordinates of galactic pole.
1013 *
1014 =       (122.9319,+27.1283) galactic coordinates of J2000 celestial pole,
1015 =       (192.8595,+27.1283) J2000 equatorial coordinates of galactic pole.
1016 *
1017 =       (359.6774,+89.7217) B1950 equatorial coordinates of J2000 pole,
1018 =       (180.3162,+89.7217) J2000 equatorial coordinates of B1950 pole.
1019 *
1020 =       (270.0000,+66.5542) B1950 equatorial coordinates of B1950 ecliptic pole,
1021 =       ( 90.0000,+66.5542) B1950 ecliptic coordinates of B1950 celestial pole.
1022 *
1023 =       (270.0000,+66.5607) J2000 equatorial coordinates of J2000 ecliptic pole,
1024 =       ( 90.0000,+66.5607) J2000 ecliptic coordinates of J2000 celestial pole.
1025 *
1026 =       ( 26.7315,+15.6441) supergalactic coordinates of B1950 celestial pole,
1027 =       (283.1894,+15.6441) B1950 equatorial coordinates of supergalactic pole.
1028 *
1029 =       ( 26.4505,+15.7089) supergalactic coordinates of J2000 celestial pole,
1030 =       (283.7542,+15.7089) J2000 equatorial coordinates of supergalactic pole.
1031 *
1032 *
1033 * wcssptr() - Spectral axis translation
1034 * -------------------------------------
1035 * wcssptr() translates the spectral axis in a wcsprm struct.  For example, a
1036 * 'FREQ' axis may be translated into 'ZOPT-F2W' and vice versa.
1037 *
1038 * PLEASE NOTE: Information in the wcsprm struct relating to the original
1039 * coordinate system will be overwritten and therefore lost.  If this is
1040 * undesirable, invoke wcssptr() on a copy of the struct made with wcssub().
1041 * The wcsprm struct is reset on return with an explicit call to wcsset().
1042 *
1043 * Given and returned:
1044 *   wcs       struct wcsprm*
1045 *                       Coordinate transformation parameters.
1046 *
1047 *   i         int*      Index of the spectral axis (0-relative).  If given < 0
1048 *                       it will be set to the first spectral axis identified
1049 *                       from the ctype[] keyvalues in the wcsprm struct.
1050 *
1051 *   ctype     char[9]   Desired spectral CTYPEia.  Wildcarding may be used as
1052 *                       for the ctypeS2 argument to spctrn() as described in
1053 *                       the prologue of spc.h, i.e. if the final three
1054 *                       characters are specified as "???", or if just the
1055 *                       eighth character is specified as '?', the correct
1056 *                       algorithm code will be substituted and returned.
1057 *
1058 * Function return value:
1059 *             int       Status return value:
1060 *                         0: Success.
1061 *                         1: Null wcsprm pointer passed.
1062 *                         2: Memory allocation failed.
1063 *                         3: Linear transformation matrix is singular.
1064 *                         4: Inconsistent or unrecognized coordinate axis
1065 *                            types.
1066 *                         5: Invalid parameter value.
1067 *                         6: Invalid coordinate transformation parameters.
1068 *                         7: Ill-conditioned coordinate transformation
1069 *                            parameters.
1070 *                        12: Invalid subimage specification (no spectral
1071 *                            axis).
1072 *
1073 *                       For returns > 1, a detailed error message is set in
1074 *                       wcsprm::err if enabled, see wcserr_enable().
1075 *
1076 *
1077 * wcslib_version() - WCSLIB version number
1078 * ----------------------------------------
1079 * wcslib_version() returns the WCSLIB version number.
1080 *
1081 * The major version number changes when the ABI changes or when the license
1082 * conditions change.  ABI changes typically result from a change to the
```

```
1083 * contents of one of the structs.  The major version number is used to
1084 * distinguish between incompatible versions of the sharable library.
1085 *
1086 * The minor version number changes with new functionality or bug fixes that do
1087 * not involve a change in the ABI.
1088 *
1089 * The auxiliary version number (which is often absent) signals changes to the
1090 * documentation, test suite, build procedures, or any other change that does
1091 * not affect the compiled library.
1092 *
1093 * Returned:
1094 *   vers[3]   int[3]    The broken-down version number:
1095 *                         0: Major version number.
1096 *                         1: Minor version number.
1097 *                         2: Auxiliary version number (zero if absent).
1098 *                       May be given as a null pointer if not required.
1099 *
1100 * Function return value:
1101 *             char*      A null-terminated, statically allocated string
1102 *                        containing the version number in the usual form, i.e.
1103 *                        "<major>.<minor>.<auxiliary>".
1104 *
1105 *
1106 * wcsprm struct - Coordinate transformation parameters
1107 * ----------------------------------------------------
1108 * The wcsprm struct contains information required to transform world
1109 * coordinates.  It consists of certain members that must be set by the user
1110 * ("given") and others that are set by the WCSLIB routines ("returned").
1111 * While the addresses of the arrays themselves may be set by wcsinit() if it
1112 * (optionally) allocates memory, their contents must be set by the user.
1113 *
1114 * Some parameters that are given are not actually required for transforming
1115 * coordinates.  These are described as "auxiliary"; the struct simply provides
1116 * a place to store them, though they may be used by wcshdo() in constructing a
1117 * FITS header from a wcsprm struct.  Some of the returned values are supplied
1118 * for informational purposes and others are for internal use only as
1119 * indicated.
1120 *
1121 * In practice, it is expected that a WCS parser would scan the FITS header to
1122 * determine the number of coordinate axes.  It would then use wcsinit() to
1123 * allocate memory for arrays in the wcsprm struct and set default values.
1124 * Then as it reread the header and identified each WCS keyrecord it would load
1125 * the value into the relevant wcsprm array element.  This is essentially what
1126 * wcspih() does - refer to the prologue of wcshdr.h.  As the final step,
1127 * wcsset() is invoked, either directly or indirectly, to set the derived
1128 * members of the wcsprm struct.  wcsset() strips off trailing blanks in all
1129 * string members and null-fills the character array.
1130 *
1131 *   int flag
1132 *     (Given and returned) This flag must be set to zero whenever any of the
1133 *     following wcsprm struct members are set or changed:
1134 *
1135 *       - wcsprm::naxis (q.v., not normally set by the user),
1136 *       - wcsprm::crpix,
1137 *       - wcsprm::pc,
1138 *       - wcsprm::cdelt,
1139 *       - wcsprm::crval,
1140 *       - wcsprm::cunit,
1141 *       - wcsprm::ctype,
1142 *       - wcsprm::lonpole,
1143 *       - wcsprm::latpole,
1144 *       - wcsprm::restfrq,
1145 *       - wcsprm::restwav,
1146 *       - wcsprm::npv,
1147 *       - wcsprm::pv,
1148 *       - wcsprm::nps,
1149 *       - wcsprm::ps,
1150 *       - wcsprm::cd,
1151 *       - wcsprm::crota,
1152 *       - wcsprm::altlin,
1153 *       - wcsprm::ntab,
1154 *       - wcsprm::nwtb,
1155 *       - wcsprm::tab,
1156 *       - wcsprm::wtb.
1157 *
1158 *     This signals the initialization routine, wcsset(), to recompute the
1159 *     returned members of the linprm, celprm, spcprm, and tabprm structs.
1160 *     wcsset() will reset flag to indicate that this has been done.
1161 *
1162 *     PLEASE NOTE: flag should be set to -1 when wcsinit() is called for the
1163 *     first time for a particular wcsprm struct in order to initialize memory
1164 *     management.  It must ONLY be used on the first initialization otherwise
1165 *     memory leaks may result.
1166 *
1167 *   int naxis
1168 *     (Given or returned) Number of pixel and world coordinate elements.
1169 *
```

```
1170 *      If wcsinit() is used to initialize the linprm struct (as would normally
1171 *      be the case) then it will set naxis from the value passed to it as a
1172 *      function argument.  The user should not subsequently modify it.
1173 *
1174 *   double *crpix
1175 *      (Given) Address of the first element of an array of double containing
1176 *      the coordinate reference pixel, CRPIXja.
1177 *
1178 *   double *pc
1179 *      (Given) Address of the first element of the PCi_ja (pixel coordinate)
1180 *      transformation matrix.  The expected order is
1181 *
1182 =        struct wcsprm wcs;
1183 =        wcs.pc = {PC1_1, PC1_2, PC2_1, PC2_2};
1184 *
1185 *      This may be constructed conveniently from a 2-D array via
1186 *
1187 =        double m[2][2] = {{PC1_1, PC1_2},
1188 =                          {PC2_1, PC2_2}};
1189 *
1190 *      which is equivalent to
1191 *
1192 =        double m[2][2];
1193 =        m[0][0] = PC1_1;
1194 =        m[0][1] = PC1_2;
1195 =        m[1][0] = PC2_1;
1196 =        m[1][1] = PC2_2;
1197 *
1198 *      The storage order for this 2-D array is the same as for the 1-D array,
1199 *      whence
1200 *
1201 =        wcs.pc = *m;
1202 *
1203 *      would be legitimate.
1204 *
1205 *   double *cdelt
1206 *      (Given) Address of the first element of an array of double containing
1207 *      the coordinate increments, CDELTia.
1208 *
1209 *   double *crval
1210 *      (Given) Address of the first element of an array of double containing
1211 *      the coordinate reference values, CRVALia.
1212 *
1213 *   char (*cunit)[72]
1214 *      (Given) Address of the first element of an array of char[72] containing
1215 *      the CUNITia keyvalues which define the units of measurement of the
1216 *      CRVALia, CDELTia, and CDi_ja keywords.
1217 *
1218 *      As CUNITia is an optional header keyword, cunit[][72] may be left blank
1219 *      but otherwise is expected to contain a standard units specification as
1220 *      defined by WCS Paper I.  Utility function wcsutrn(), described in
1221 *      wcsunits.h, is available to translate commonly used non-standard units
1222 *      specifications but this must be done as a separate step before invoking
1223 *      wcsset().
1224 *
1225 *      For celestial axes, if cunit[][72] is not blank, wcsset() uses
1226 *      wcsunits() to parse it and scale cdelt[], crval[], and cd[][*] to
1227 *      degrees.  It then resets cunit[][72] to "deg".
1228 *
1229 *      For spectral axes, if cunit[][72] is not blank, wcsset() uses wcsunits()
1230 *      to parse it and scale cdelt[], crval[], and cd[][*] to SI units.  It
1231 *      then resets cunit[][72] accordingly.
1232 *
1233 *      wcsset() ignores cunit[][72] for other coordinate types; cunit[][72] may
1234 *      be used to label coordinate values.
1235 *
1236 *      These variables accomodate the longest allowed string-valued FITS
1237 *      keyword, being limited to 68 characters, plus the null-terminating
1238 *      character.
1239 *
1240 *   char (*ctype)[72]
1241 *      (Given) Address of the first element of an array of char[72] containing
1242 *      the coordinate axis types, CTYPEia.
1243 *
1244 *      The ctype[][72] keyword values must be in upper case and there must be
1245 *      zero or one pair of matched celestial axis types, and zero or one
1246 *      spectral axis.  The ctype[][72] strings should be padded with blanks on
1247 *      the right and null-terminated so that they are at least eight characters
1248 *      in length.
1249 *
1250 *      These variables accomodate the longest allowed string-valued FITS
1251 *      keyword, being limited to 68 characters, plus the null-terminating
1252 *      character.
1253 *
1254 *   double lonpole
1255 *      (Given and returned) The native longitude of the celestial pole, phi_p,
1256 *      given by LONPOLEa [deg] or by PVi_2a [deg] attached to the longitude
```

```
1257 *      axis which takes precedence if defined, and ...
1258 *   double latpole
1259 *      (Given and returned) ... the native latitude of the celestial pole,
1260 *      theta_p, given by LATPOLEa [deg] or by PVi_3a [deg] attached to the
1261 *      longitude axis which takes precedence if defined.
1262 *
1263 *      lonpole and latpole may be left to default to values set by wcsinit()
1264 *      (see celprm::ref), but in any case they will be reset by wcsset() to
1265 *      the values actually used.  Note therefore that if the wcsprm struct is
1266 *      reused without resetting them, whether directly or via wcsinit(), they
1267 *      will no longer have their default values.
1268 *
1269 *   double restfrq
1270 *      (Given) The rest frequency [Hz], and/or ...
1271 *   double restwav
1272 *      (Given) ... the rest wavelength in vacuo [m], only one of which need be
1273 *      given, the other should be set to zero.
1274 *
1275 *   int npv
1276 *      (Given) The number of entries in the wcsprm::pv[] array.
1277 *
1278 *   int npvmax
1279 *      (Given or returned) The length of the wcsprm::pv[] array.
1280 *
1281 *      npvmax will be set by wcsinit() if it allocates memory for wcsprm::pv[],
1282 *      otherwise it must be set by the user.  See also wcsnpv().
1283 *
1284 *   struct pvcard *pv
1285 *      (Given) Address of the first element of an array of length npvmax of
1286 *      pvcard structs.
1287 *
1288 *      As a FITS header parser encounters each PVi_ma keyword it should load it
1289 *      into a pvcard struct in the array and increment npv.  wcsset()
1290 *      interprets these as required.
1291 *
1292 *      Note that, if they were not given, wcsset() resets the entries for
1293 *      PVi_1a, PVi_2a, PVi_3a, and PVi_4a for longitude axis i to match
1294 *      phi_0 and theta_0 (the native longitude and latitude of the reference
1295 *      point), LONPOLEa and LATPOLEa respectively.
1296 *
1297 *   int nps
1298 *      (Given) The number of entries in the wcsprm::ps[] array.
1299 *
1300 *   int npsmax
1301 *      (Given or returned) The length of the wcsprm::ps[] array.
1302 *
1303 *      npsmax will be set by wcsinit() if it allocates memory for wcsprm::ps[],
1304 *      otherwise it must be set by the user.  See also wcsnps().
1305 *
1306 *   struct pscard *ps
1307 *      (Given) Address of the first element of an array of length npsmax of
1308 *      pscard structs.
1309 *
1310 *      As a FITS header parser encounters each PSi_ma keyword it should load it
1311 *      into a pscard struct in the array and increment nps.  wcsset()
1312 *      interprets these as required (currently no PSi_ma keyvalues are
1313 *      recognized).
1314 *
1315 *   double *cd
1316 *      (Given) For historical compatibility, the wcsprm struct supports two
1317 *      alternate specifications of the linear transformation matrix, those
1318 *      associated with the CDi_ja keywords, and ...
1319 *   double *crota
1320 *      (Given) ... those associated with the CROTAi keywords.  Although these
1321 *      may not formally co-exist with PCi_ja, the approach taken here is simply
1322 *      to ignore them if given in conjunction with PCi_ja.
1323 *
1324 *   int altlin
1325 *      (Given) altlin is a bit flag that denotes which of the PCi_ja, CDi_ja
1326 *      and CROTAi keywords are present in the header:
1327 *
1328 *      - Bit 0: PCi_ja is present.
1329 *
1330 *      - Bit 1: CDi_ja is present.
1331 *
1332 *        Matrix elements in the IRAF convention are equivalent to the product
1333 *        CDi_ja = CDELTia * PCi_ja, but the defaults differ from that of the
1334 *        PCi_ja matrix.  If one or more CDi_ja keywords are present then all
1335 *        unspecified CDi_ja default to zero.  If no CDi_ja (or CROTAi) keywords
1336 *        are present, then the header is assumed to be in PCi_ja form whether
1337 *        or not any PCi_ja keywords are present since this results in an
1338 *        interpretation of CDELTia consistent with the original FITS
1339 *        specification.
1340 *
1341 *        While CDi_ja may not formally co-exist with PCi_ja, it may co-exist
1342 *        with CDELTia and CROTAi which are to be ignored.
1343 *
```

```
1344 *     – Bit 2: CROTAi is present.
1345 *
1346 *        In the AIPS convention, CROTAi may only be associated with the
1347 *        latitude axis of a celestial axis pair.  It specifies a rotation in
1348 *        the image plane that is applied AFTER the CDELTia; any other CROTAi
1349 *        keywords are ignored.
1350 *
1351 *        CROTAi may not formally co-exist with PCi_ja.
1352 *
1353 *        CROTAi and CDELTia may formally co-exist with CDi_ja but if so are to
1354 *        be ignored.
1355 *
1356 *     – Bit 3: PCi_ja + CDELTia was derived from CDi_ja by wcspcx().
1357 *
1358 *        This bit is set by wcspcx() when it derives PCi_ja and CDELTia from
1359 *        CDi_ja via an orthonormal decomposition.  In particular, it signals
1360 *        wcsset() not to replace PCi_ja by a copy of CDi_ja with CDELTia set
1361 *        to unity.
1362 *
1363 *     CDi_ja and CROTAi keywords, if found, are to be stored in the wcsprm::cd
1364 *     and wcsprm::crota arrays which are dimensioned similarly to wcsprm::pc
1365 *     and wcsprm::cdelt.  FITS header parsers should use the following
1366 *     procedure:
1367 *
1368 *     – Whenever a PCi_ja keyword is encountered: altlin |= 1;
1369 *
1370 *     – Whenever a CDi_ja keyword is encountered: altlin |= 2;
1371 *
1372 *     – Whenever a CROTAi keyword is encountered: altlin |= 4;
1373 *
1374 *     If none of these bits are set the PCi_ja representation results, i.e.
1375 *     wcsprm::pc and wcsprm::cdelt will be used as given.
1376 *
1377 *     These alternate specifications of the linear transformation matrix are
1378 *     translated immediately to PCi_ja by wcsset() and are invisible to the
1379 *     lower-level WCSLIB routines.  In particular, unless bit 3 is also set,
1380 *     wcsset() resets wcsprm::cdelt to unity if CDi_ja is present (and no
1381 *     PCi_ja).
1382 *
1383 *     If CROTAi are present but none is associated with the latitude axis
1384 *     (and no PCi_ja or CDi_ja), then wcsset() reverts to a unity PCi_ja
1385 *     matrix.
1386 *
1387 *   int velref
1388 *     (Given) AIPS velocity code VELREF, refer to spcaips().
1389 *
1390 *     It is not necessary to reset the wcsprm struct (via wcsset()) when
1391 *     wcsprm::velref is changed.
1392 *
1393 *   char alt[4]
1394 *     (Given, auxiliary) Character code for alternate coordinate descriptions
1395 *     (i.e. the 'a' in keyword names such as CTYPEia).  This is blank for the
1396 *     primary coordinate description, or one of the 26 upper-case letters,
1397 *     A–Z.
1398 *
1399 *     An array of four characters is provided for alignment purposes, only the
1400 *     first is used.
1401 *
1402 *     It is not necessary to reset the wcsprm struct (via wcsset()) when
1403 *     wcsprm::alt is changed.
1404 *
1405 *   int colnum
1406 *     (Given, auxiliary) Where the coordinate representation is associated
1407 *     with an image-array column in a FITS binary table, this variable may be
1408 *     used to record the relevant column number.
1409 *
1410 *     It should be set to zero for an image header or pixel list.
1411 *
1412 *     It is not necessary to reset the wcsprm struct (via wcsset()) when
1413 *     wcsprm::colnum is changed.
1414 *
1415 *   int *colax
1416 *     (Given, auxiliary) Address of the first element of an array of int
1417 *     recording the column numbers for each axis in a pixel list.
1418 *
1419 *     The array elements should be set to zero for an image header or image
1420 *     array in a binary table.
1421 *
1422 *     It is not necessary to reset the wcsprm struct (via wcsset()) when
1423 *     wcsprm::colax is changed.
1424 *
1425 *   char (*cname)[72]
1426 *     (Given, auxiliary) The address of the first element of an array of
1427 *     char[72] containing the coordinate axis names, CNAMEia.
1428 *
1429 *     These variables accomodate the longest allowed string-valued FITS
1430 *     keyword, being limited to 68 characters, plus the null-terminating
```

```
1431 *      character.
1432 *
1433 *      It is not necessary to reset the wcsprm struct (via wcsset()) when
1434 *      wcsprm::cname is changed.
1435 *
1436 *   double *crder
1437 *      (Given, auxiliary) Address of the first element of an array of double
1438 *      recording the random error in the coordinate value, CRDERia.
1439 *
1440 *      It is not necessary to reset the wcsprm struct (via wcsset()) when
1441 *      wcsprm::crder is changed.
1442 *
1443 *   double *csyer
1444 *      (Given, auxiliary) Address of the first element of an array of double
1445 *      recording the systematic error in the coordinate value, CSYERia.
1446 *
1447 *      It is not necessary to reset the wcsprm struct (via wcsset()) when
1448 *      wcsprm::csyer is changed.
1449 *
1450 *   double *czphs
1451 *      (Given, auxiliary) Address of the first element of an array of double
1452 *      recording the time at the zero point of a phase axis, CZPHSia.
1453 *
1454 *      It is not necessary to reset the wcsprm struct (via wcsset()) when
1455 *      wcsprm::czphs is changed.
1456 *
1457 *   double *cperi
1458 *      (Given, auxiliary) Address of the first element of an array of double
1459 *      recording the period of a phase axis, CPERIia.
1460 *
1461 *      It is not necessary to reset the wcsprm struct (via wcsset()) when
1462 *      wcsprm::cperi is changed.
1463 *
1464 *   char wcsname[72]
1465 *      (Given, auxiliary) The name given to the coordinate representation,
1466 *      WCSNAMEa.  This variable accomodates the longest allowed string-valued
1467 *      FITS keyword, being limited to 68 characters, plus the null-terminating
1468 *      character.
1469 *
1470 *      It is not necessary to reset the wcsprm struct (via wcsset()) when
1471 *      wcsprm::wcsname is changed.
1472 *
1473 *   char timesys[72]
1474 *      (Given, auxiliary) TIMESYS keyvalue, being the time scale (UTC, TAI,
1475 *      etc.) in which all other time-related auxiliary header values are
1476 *      recorded.  Also defines the time scale for an image axis with CTYPEia
1477 *      set to 'TIME'.
1478 *
1479 *      It is not necessary to reset the wcsprm struct (via wcsset()) when
1480 *      wcsprm::timesys is changed.
1481 *
1482 *   char trefpos[72]
1483 *      (Given, auxiliary) TREFPOS keyvalue, being the location in space where
1484 *      the recorded time is valid.
1485 *
1486 *      It is not necessary to reset the wcsprm struct (via wcsset()) when
1487 *      wcsprm::trefpos is changed.
1488 *
1489 *   char trefdir[72]
1490 *      (Given, auxiliary) TREFDIR keyvalue, being the reference direction used
1491 *      in calculating a pathlength delay.
1492 *
1493 *      It is not necessary to reset the wcsprm struct (via wcsset()) when
1494 *      wcsprm::trefdir is changed.
1495 *
1496 *   char plephem[72]
1497 *      (Given, auxiliary) PLEPHEM keyvalue, being the Solar System ephemeris
1498 *      used for calculating a pathlength delay.
1499 *
1500 *      It is not necessary to reset the wcsprm struct (via wcsset()) when
1501 *      wcsprm::plephem is changed.
1502 *
1503 *   char timeunit[72]
1504 *      (Given, auxiliary) TIMEUNIT keyvalue, being the time units in which
1505 *      the following header values are expressed: TSTART, TSTOP, TIMEOFFS,
1506 *      TIMSYER, TIMRDER, TIMEDEL.  It also provides the default value for
1507 *      CUNITia for time axes.
1508 *
1509 *      It is not necessary to reset the wcsprm struct (via wcsset()) when
1510 *      wcsprm::timeunit is changed.
1511 *
1512 *   char dateref[72]
1513 *      (Given, auxiliary) DATEREF keyvalue, being the date of a reference epoch
1514 *      relative to which other time measurements refer.
1515 *
1516 *      It is not necessary to reset the wcsprm struct (via wcsset()) when
1517 *      wcsprm::dateref is changed.
```

```
1518 *
1519 *   double mjdref[2]
1520 *      (Given, auxiliary) MJDREF keyvalue, equivalent to DATEREF expressed as
1521 *      a Modified Julian Date (MJD = JD - 2400000.5).  The value is given as
1522 *      the sum of the two-element vector, allowing increased precision.
1523 *
1524 *      It is not necessary to reset the wcsprm struct (via wcsset()) when
1525 *      wcsprm::mjdref is changed.
1526 *
1527 *   double timeoffs
1528 *      (Given, auxiliary) TIMEOFFS keyvalue, being a time offset, which may be
1529 *      used, for example, to provide a uniform clock correction for times
1530 *      referenced to DATEREF.
1531 *
1532 *      It is not necessary to reset the wcsprm struct (via wcsset()) when
1533 *      wcsprm::timeoffs is changed.
1534 *
1535 *   char dateobs[72]
1536 *      (Given, auxiliary) DATE-OBS keyvalue, being the date at the start of the
1537 *      observation unless otherwise explained in the DATE-OBS keycomment, in
1538 *      ISO format, yyyy-mm-ddThh:mm:ss.
1539 *
1540 *      It is not necessary to reset the wcsprm struct (via wcsset()) when
1541 *      wcsprm::dateobs is changed.
1542 *
1543 *   char datebeg[72]
1544 *      (Given, auxiliary) DATE-BEG keyvalue, being the date at the start of the
1545 *      observation in ISO format, yyyy-mm-ddThh:mm:ss.
1546 *
1547 *      It is not necessary to reset the wcsprm struct (via wcsset()) when
1548 *      wcsprm::datebeg is changed.
1549 *
1550 *   char dateavg[72]
1551 *      (Given, auxiliary) DATE-AVG keyvalue, being the date at a representative
1552 *      mid-point of the observation in ISO format, yyyy-mm-ddThh:mm:ss.
1553 *
1554 *      It is not necessary to reset the wcsprm struct (via wcsset()) when
1555 *      wcsprm::dateavg is changed.
1556 *
1557 *   char dateend[72]
1558 *      (Given, auxiliary) DATE-END keyvalue, baing the date at the end of the
1559 *      observation in ISO format, yyyy-mm-ddThh:mm:ss.
1560 *
1561 *      It is not necessary to reset the wcsprm struct (via wcsset()) when
1562 *      wcsprm::dateend is changed.
1563 *
1564 *   double mjdobs
1565 *      (Given, auxiliary) MJD-OBS keyvalue, equivalent to DATE-OBS expressed
1566 *      as a Modified Julian Date (MJD = JD - 2400000.5).
1567 *
1568 *      It is not necessary to reset the wcsprm struct (via wcsset()) when
1569 *      wcsprm::mjdobs is changed.
1570 *
1571 *   double mjdbeg
1572 *      (Given, auxiliary) MJD-BEG keyvalue, equivalent to DATE-BEG expressed
1573 *      as a Modified Julian Date (MJD = JD - 2400000.5).
1574 *
1575 *      It is not necessary to reset the wcsprm struct (via wcsset()) when
1576 *      wcsprm::mjdbeg is changed.
1577 *
1578 *   double mjdavg
1579 *      (Given, auxiliary) MJD-AVG keyvalue, equivalent to DATE-AVG expressed
1580 *      as a Modified Julian Date (MJD = JD - 2400000.5).
1581 *
1582 *      It is not necessary to reset the wcsprm struct (via wcsset()) when
1583 *      wcsprm::mjdavg is changed.
1584 *
1585 *   double mjdend
1586 *      (Given, auxiliary) MJD-END keyvalue, equivalent to DATE-END expressed
1587 *      as a Modified Julian Date (MJD = JD - 2400000.5).
1588 *
1589 *      It is not necessary to reset the wcsprm struct (via wcsset()) when
1590 *      wcsprm::mjdend is changed.
1591 *
1592 *   double jepoch
1593 *      (Given, auxiliary) JEPOCH keyvalue, equivalent to DATE-OBS expressed
1594 *      as a Julian epoch.
1595 *
1596 *      It is not necessary to reset the wcsprm struct (via wcsset()) when
1597 *      wcsprm::jepoch is changed.
1598 *
1599 *   double bepoch
1600 *      (Given, auxiliary) BEPOCH keyvalue, equivalent to DATE-OBS expressed
1601 *      as a Besselian epoch
1602 *
1603 *      It is not necessary to reset the wcsprm struct (via wcsset()) when
1604 *      wcsprm::bepoch is changed.
```

```
1605 *
1606 *   double tstart
1607 *      (Given, auxiliary) TSTART keyvalue, equivalent to DATE-BEG expressed
1608 *      as a time in units of TIMEUNIT relative to DATEREF+TIMEOFFS.
1609 *
1610 *      It is not necessary to reset the wcsprm struct (via wcsset()) when
1611 *      wcsprm::tstart is changed.
1612 *
1613 *   double tstop
1614 *      (Given, auxiliary) TSTOP keyvalue, equivalent to DATE-END expressed
1615 *      as a time in units of TIMEUNIT relative to DATEREF+TIMEOFFS.
1616 *
1617 *      It is not necessary to reset the wcsprm struct (via wcsset()) when
1618 *      wcsprm::tstop is changed.
1619 *
1620 *   double xposure
1621 *      (Given, auxiliary) XPOSURE keyvalue, being the effective exposure time
1622 *      in units of TIMEUNIT.
1623 *
1624 *      It is not necessary to reset the wcsprm struct (via wcsset()) when
1625 *      wcsprm::xposure is changed.
1626 *
1627 *   double telapse
1628 *      (Given, auxiliary) TELAPSE keyvalue, equivalent to the elapsed time
1629 *      between DATE-BEG and DATE-END, in units of TIMEUNIT.
1630 *
1631 *      It is not necessary to reset the wcsprm struct (via wcsset()) when
1632 *      wcsprm::telapse is changed.
1633 *
1634 *   double timsyer
1635 *      (Given, auxiliary) TIMSYER keyvalue, being the absolute error of the
1636 *      time values, in units of TIMEUNIT.
1637 *
1638 *      It is not necessary to reset the wcsprm struct (via wcsset()) when
1639 *      wcsprm::timsyer is changed.
1640 *
1641 *   double timrder
1642 *      (Given, auxiliary) TIMRDER keyvalue, being the accuracy of time stamps
1643 *      relative to each other, in units of TIMEUNIT.
1644 *
1645 *      It is not necessary to reset the wcsprm struct (via wcsset()) when
1646 *      wcsprm::timrder is changed.
1647 *
1648 *   double timedel
1649 *      (Given, auxiliary) TIMEDEL keyvalue, being the resolution of the time
1650 *      stamps.
1651 *
1652 *      It is not necessary to reset the wcsprm struct (via wcsset()) when
1653 *      wcsprm::timedel is changed.
1654 *
1655 *   double timepixr
1656 *      (Given, auxiliary) TIMEPIXR keyvalue, being the relative position of the
1657 *      time stamps in binned time intervals, a value between 0.0 and 1.0.
1658 *
1659 *      It is not necessary to reset the wcsprm struct (via wcsset()) when
1660 *      wcsprm::timepixr is changed.
1661 *
1662 *   double obsgeo[6]
1663 *      (Given, auxiliary) Location of the observer in a standard terrestrial
1664 *      reference frame.  The first three give ITRS Cartesian coordinates
1665 *      OBSGEO-X [m],   OBSGEO-Y [m],   OBSGEO-Z [m], and the second three give
1666 *      OBSGEO-L [deg], OBSGEO-B [deg], OBSGEO-H [m], which are related through
1667 *      a standard transformation.
1668 *
1669 *      It is not necessary to reset the wcsprm struct (via wcsset()) when
1670 *      wcsprm::obsgeo is changed.
1671 *
1672 *   char obsorbit[72]
1673 *      (Given, auxiliary) OBSORBIT keyvalue, being the URI, URL, or name of an
1674 *      orbit ephemeris file giving spacecraft coordinates relating to TREFPOS.
1675 *
1676 *      It is not necessary to reset the wcsprm struct (via wcsset()) when
1677 *      wcsprm::obsorbit is changed.
1678 *
1679 *   char radesys[72]
1680 *      (Given, auxiliary) The equatorial or ecliptic coordinate system type,
1681 *      RADESYSa.
1682 *
1683 *      It is not necessary to reset the wcsprm struct (via wcsset()) when
1684 *      wcsprm::radesys is changed.
1685 *
1686 *   double equinox
1687 *      (Given, auxiliary) The equinox associated with dynamical equatorial or
1688 *      ecliptic coordinate systems, EQUINOXa (or EPOCH in older headers).  Not
1689 *      applicable to ICRS equatorial or ecliptic coordinates.
1690 *
1691 *      It is not necessary to reset the wcsprm struct (via wcsset()) when
```

```
1692 *     wcsprm::equinox is changed.
1693 *
1694 *   char specsys[72]
1695 *     (Given, auxiliary) Spectral reference frame (standard of rest),
1696 *     SPECSYSa.
1697 *
1698 *     It is not necessary to reset the wcsprm struct (via wcsset()) when
1699 *     wcsprm::specsys is changed.
1700 *
1701 *   char ssysobs[72]
1702 *     (Given, auxiliary) The spectral reference frame in which there is no
1703 *     differential variation in the spectral coordinate across the
1704 *     field-of-view, SSYSOBSa.
1705 *
1706 *     It is not necessary to reset the wcsprm struct (via wcsset()) when
1707 *     wcsprm::ssysobs is changed.
1708 *
1709 *   double velosys
1710 *     (Given, auxiliary) The relative radial velocity [m/s] between the
1711 *     observer and the selected standard of rest in the direction of the
1712 *     celestial reference coordinate, VELOSYSa.
1713 *
1714 *     It is not necessary to reset the wcsprm struct (via wcsset()) when
1715 *     wcsprm::velosys is changed.
1716 *
1717 *   double zsource
1718 *     (Given, auxiliary) The redshift, ZSOURCEa, of the source.
1719 *
1720 *     It is not necessary to reset the wcsprm struct (via wcsset()) when
1721 *     wcsprm::zsource is changed.
1722 *
1723 *   char ssyssrc[72]
1724 *     (Given, auxiliary) The spectral reference frame (standard of rest),
1725 *     SSYSSRCa, in which wcsprm::zsource was measured.
1726 *
1727 *     It is not necessary to reset the wcsprm struct (via wcsset()) when
1728 *     wcsprm::ssyssrc is changed.
1729 *
1730 *   double velangl
1731 *     (Given, auxiliary) The angle [deg] that should be used to decompose an
1732 *     observed velocity into radial and transverse components.
1733 *
1734 *     It is not necessary to reset the wcsprm struct (via wcsset()) when
1735 *     wcsprm::velangl is changed.
1736 *
1737 *   struct auxprm *aux
1738 *     (Given, auxiliary) This struct holds auxiliary coordinate system
1739 *     information of a specialist nature.  While these parameters may be
1740 *     widely recognized within particular fields of astronomy, they differ
1741 *     from the above auxiliary parameters in not being defined by any of the
1742 *     FITS WCS standards.  Collecting them together in a separate struct that
1743 *     is allocated only when required helps to control bloat in the size of
1744 *     the wcsprm struct.
1745 *
1746 *   int ntab
1747 *     (Given) See wcsprm::tab.
1748 *
1749 *   int nwtb
1750 *     (Given) See wcsprm::wtb.
1751 *
1752 *   struct tabprm *tab
1753 *     (Given) Address of the first element of an array of ntab tabprm structs
1754 *     for which memory has been allocated.  These are used to store tabular
1755 *     transformation parameters.
1756 *
1757 *     Although technically wcsprm::ntab and tab are "given", they will
1758 *     normally be set by invoking wcstab(), whether directly or indirectly.
1759 *
1760 *     The tabprm structs contain some members that must be supplied and others
1761 *     that are derived.  The information to be supplied comes primarily from
1762 *     arrays stored in one or more FITS binary table extensions.  These
1763 *     arrays, referred to here as "wcstab arrays", are themselves located by
1764 *     parameters stored in the FITS image header.
1765 *
1766 *   struct wtbarr *wtb
1767 *     (Given) Address of the first element of an array of nwtb wtbarr structs
1768 *     for which memory has been allocated.  These are used in extracting
1769 *     wcstab arrays from a FITS binary table.
1770 *
1771 *     Although technically wcsprm::nwtb and wtb are "given", they will
1772 *     normally be set by invoking wcstab(), whether directly or indirectly.
1773 *
1774 *   char lngtyp[8]
1775 *     (Returned) Four-character WCS celestial longitude and ...
1776 *   char lattyp[8]
1777 *     (Returned) ... latitude axis types. e.g. "RA", "DEC", "GLON", "GLAT",
1778 *     etc. extracted from 'RA--', 'DEC-', 'GLON', 'GLAT', etc. in the first
```

```
1779 *      four characters of CTYPEia but with trailing dashes removed.  (Declared
1780 *      as char[8] for alignment reasons.)
1781 *
1782 *   int lng
1783 *      (Returned) Index for the longitude coordinate, and ...
1784 *   int lat
1785 *      (Returned) ... index for the latitude coordinate, and ...
1786 *   int spec
1787 *      (Returned) ... index for the spectral coordinate in the imgcrd[][] and
1788 *      world[][] arrays in the API of wcsp2s(), wcss2p() and wcsmix().
1789 *
1790 *      These may also serve as indices into the pixcrd[][] array provided that
1791 *      the PCi_ja matrix does not transpose axes.
1792 *
1793 *   int cubeface
1794 *      (Returned) Index into the pixcrd[][] array for the CUBEFACE axis.  This
1795 *      is used for quadcube projections where the cube faces are stored on a
1796 *      separate axis (see wcs.h).
1797 *
1798 *   int *types
1799 *      (Returned) Address of the first element of an array of int containing a
1800 *      four-digit type code for each axis.
1801 *
1802 *      - First digit (i.e. 1000s):
1803 *        - 0: Non-specific coordinate type.
1804 *        - 1: Stokes coordinate.
1805 *        - 2: Celestial coordinate (including CUBEFACE).
1806 *        - 3: Spectral coordinate.
1807 *        - 4: Time coordinate.
1808 *
1809 *      - Second digit (i.e. 100s):
1810 *        - 0: Linear axis.
1811 *        - 1: Quantized axis (STOKES, CUBEFACE).
1812 *        - 2: Non-linear celestial axis.
1813 *        - 3: Non-linear spectral axis.
1814 *        - 4: Logarithmic axis.
1815 *        - 5: Tabular axis.
1816 *
1817 *      - Third digit (i.e. 10s):
1818 *        - 0: Group number, e.g. lookup table number, being an index into the
1819 *             tabprm array (see above).
1820 *
1821 *      - The fourth digit is used as a qualifier depending on the axis type.
1822 *
1823 *        - For celestial axes:
1824 *          - 0: Longitude coordinate.
1825 *          - 1: Latitude coordinate.
1826 *          - 2: CUBEFACE number.
1827 *
1828 *        - For lookup tables: the axis number in a multidimensional table.
1829 *
1830 *      CTYPEia in "4-3" form with unrecognized algorithm code will have its
1831 *      type set to -1 and generate an error.
1832 *
1833 *   struct linprm lin
1834 *      (Returned) Linear transformation parameters (usage is described in the
1835 *      prologue to lin.h).
1836 *
1837 *   struct celprm cel
1838 *      (Returned) Celestial transformation parameters (usage is described in
1839 *      the prologue to cel.h).
1840 *
1841 *   struct spcprm spc
1842 *      (Returned) Spectral transformation parameters (usage is described in the
1843 *      prologue to spc.h).
1844 *
1845 *   struct wcserr *err
1846 *      (Returned) If enabled, when an error status is returned, this struct
1847 *      contains detailed information about the error, see wcserr_enable().
1848 *
1849 *   int m_flag
1850 *      (For internal use only.)
1851 *   int m_naxis
1852 *      (For internal use only.)
1853 *   double *m_crpix
1854 *      (For internal use only.)
1855 *   double *m_pc
1856 *      (For internal use only.)
1857 *   double *m_cdelt
1858 *      (For internal use only.)
1859 *   double *m_crval
1860 *      (For internal use only.)
1861 *   char (*m_cunit)[72]
1862 *      (For internal use only.)
1863 *   char (*m_ctype)[72]
1864 *      (For internal use only.)
1865 *   struct pvcard *m_pv
```

```
1866 *      (For internal use only.)
1867 *    struct pscard *m_ps
1868 *      (For internal use only.)
1869 *    double *m_cd
1870 *      (For internal use only.)
1871 *    double *m_crota
1872 *      (For internal use only.)
1873 *    int *m_colax
1874 *      (For internal use only.)
1875 *    char (*m_cname)[72]
1876 *      (For internal use only.)
1877 *    double *m_crder
1878 *      (For internal use only.)
1879 *    double *m_csyer
1880 *      (For internal use only.)
1881 *    double *m_czphs
1882 *      (For internal use only.)
1883 *    double *m_cperi
1884 *      (For internal use only.)
1885 *    struct tabprm *m_tab
1886 *      (For internal use only.)
1887 *    struct wtbarr *m_wtb
1888 *      (For internal use only.)
1889 *
1890 *
1891 * pvcard struct - Store for PVi_ma keyrecords
1892 * -----------------------------------------
1893 * The pvcard struct is used to pass the parsed contents of PVi_ma keyrecords
1894 * to wcsset() via the wcsprm struct.
1895 *
1896 * All members of this struct are to be set by the user.
1897 *
1898 *    int i
1899 *      (Given) Axis number (1-relative), as in the FITS PVi_ma keyword.  If
1900 *      i == 0, wcsset() will replace it with the latitude axis number.
1901 *
1902 *    int m
1903 *      (Given) Parameter number (non-negative), as in the FITS PVi_ma keyword.
1904 *
1905 *    double value
1906 *      (Given) Parameter value.
1907 *
1908 *
1909 * pscard struct - Store for PSi_ma keyrecords
1910 * -----------------------------------------
1911 * The pscard struct is used to pass the parsed contents of PSi_ma keyrecords
1912 * to wcsset() via the wcsprm struct.
1913 *
1914 * All members of this struct are to be set by the user.
1915 *
1916 *    int i
1917 *      (Given) Axis number (1-relative), as in the FITS PSi_ma keyword.
1918 *
1919 *    int m
1920 *      (Given) Parameter number (non-negative), as in the FITS PSi_ma keyword.
1921 *
1922 *    char value[72]
1923 *      (Given) Parameter value.
1924 *
1925 *
1926 * auxprm struct - Additional auxiliary parameters
1927 * -------------------------------------------------
1928 * The auxprm struct holds auxiliary coordinate system information of a
1929 * specialist nature.  It is anticipated that this struct will expand in future
1930 * to accomodate additional parameters.
1931 *
1932 * All members of this struct are to be set by the user.
1933 *
1934 *    double rsun_ref
1935 *      (Given, auxiliary) Reference radius of the Sun used in coordinate
1936 *      calculations (m).
1937 *
1938 *    double dsun_obs
1939 *      (Given, auxiliary) Distance between the centre of the Sun and the
1940 *      observer (m).
1941 *
1942 *    double crln_obs
1943 *      (Given, auxiliary) Carrington heliographic longitude of the observer
1944 *      (deg).
1945 *
1946 *    double hgln_obs
1947 *      (Given, auxiliary) Stonyhurst heliographic longitude of the observer
1948 *      (deg).
1949 *
1950 *    double hglt_obs
1951 *      (Given, auxiliary) Heliographic latitude (Carrington or Stonyhurst) of
1952 *      the observer (deg).
```

```
1953 *
1954 *
1955 * Global variable: const char *wcs_errmsg[] - Status return messages
1956 * -------------------------------------------------------------
1957 * Error messages to match the status value returned from each function.
1958 *
1959 *==========================================================================*/
1960
1961 #ifndef WCSLIB_WCS
1962 #define WCSLIB_WCS
1963
1964 #include "lin.h"
1965 #include "cel.h"
1966 #include "spc.h"
1967
1968 #ifdef __cplusplus
1969 extern "C" {
1970 #define wtbarr wtbarr_s          // See prologue of wtbarr.h.
1971 #endif
1972
1973 #define WCSSUB_LONGITUDE 0x1001
1974 #define WCSSUB_LATITUDE  0x1002
1975 #define WCSSUB_CUBEFACE  0x1004
1976 #define WCSSUB_CELESTIAL 0x1007
1977 #define WCSSUB_SPECTRAL  0x1008
1978 #define WCSSUB_STOKES    0x1010
1979 #define WCSSUB_TIME      0x1020
1980
1981
1982 #define WCSCOMPARE_ANCILLARY 0x0001
1983 #define WCSCOMPARE_TILING    0x0002
1984 #define WCSCOMPARE_CRPIX     0x0004
1985
1986
1987 extern const char *wcs_errmsg[];
1988
1989 enum wcs_errmsg_enum {
1990   WCSERR_SUCCESS        =  0,    // Success.
1991   WCSERR_NULL_POINTER   =  1,       // Null wcsprm pointer passed.
1992   WCSERR_MEMORY         =  2,    // Memory allocation failed.
1993   WCSERR_SINGULAR_MTX   =  3,       // Linear transformation matrix is singular.
1994   WCSERR_BAD_CTYPE      =  4,  // Inconsistent or unrecognized coordinate
1995                                  // axis type.
1996   WCSERR_BAD_PARAM      =  5,  // Invalid parameter value.
1997   WCSERR_BAD_COORD_TRANS =  6,    // Unrecognized coordinate transformation
1998                                  // parameter.
1999   WCSERR_ILL_COORD_TRANS =  7,    // Ill-conditioned coordinate transformation
2000                                  // parameter.
2001   WCSERR_BAD_PIX        =  8,    // One or more of the pixel coordinates were
2002                                  // invalid.
2003   WCSERR_BAD_WORLD      =  9,  // One or more of the world coordinates were
2004                                  // invalid.
2005   WCSERR_BAD_WORLD_COORD = 10,    // Invalid world coordinate.
2006   WCSERR_NO_SOLUTION    = 11,       // No solution found in the specified
2007                                  // interval.
2008   WCSERR_BAD_SUBIMAGE   = 12,       // Invalid subimage specification.
2009   WCSERR_NON_SEPARABLE  = 13,       // Non-separable subimage coordinate system.
2010   WCSERR_UNSET          = 14       // wcsprm struct is unset.
2011 };
2012
2013
2014 // Struct used for storing PVi_ma keywords.
2015 struct pvcard {
2016   int i;                        // Axis number, as in PVi_ma (1-relative).
2017   int m;                        // Parameter number, ditto  (0-relative).
2018   double value;                     // Parameter value.
2019 };
2020
2021 // Size of the pvcard struct in int units, used by the Fortran wrappers.
2022 #define PVLEN (sizeof(struct pvcard)/sizeof(int))
2023
2024 // Struct used for storing PSi_ma keywords.
2025 struct pscard {
2026   int i;                        // Axis number, as in PSi_ma (1-relative).
2027   int m;                        // Parameter number, ditto  (0-relative).
2028   char value[72];           // Parameter value.
2029 };
2030
2031 // Size of the pscard struct in int units, used by the Fortran wrappers.
2032 #define PSLEN (sizeof(struct pscard)/sizeof(int))
2033
2034 // Struct used to hold additional auxiliary parameters.
2035 struct auxprm {
2036   double rsun_ref;             // Solar radius.
2037   double dsun_obs;             // Distance from Sun centre to observer.
2038   double crln_obs;             // Carrington heliographic lng of observer.
2039   double hgln_obs;             // Stonyhurst heliographic lng of observer.
```

```
2040    double hglt_obs;                   // Heliographic latitude of observer.
2041  };
2042
2043  // Size of the auxprm struct in int units, used by the Fortran wrappers.
2044  #define AUXLEN (sizeof(struct auxprm)/sizeof(int))
2045
2046
2047  struct wcsprm {
2048    // Initialization flag (see the prologue above).
2049    //------------------------------------------------------------------------
2050    int    flag;                       // Set to zero to force initialization.
2051
2052    // FITS header keyvalues to be provided (see the prologue above).
2053    //------------------------------------------------------------------------
2054    int    naxis;                      // Number of axes (pixel and coordinate).
2055    double *crpix;            // CRPIXja keyvalues for each pixel axis.
2056    double *pc;               // PCi_ja  linear transformation matrix.
2057    double *cdelt;            // CDELTia keyvalues for each coord axis.
2058    double *crval;            // CRVALia keyvalues for each coord axis.
2059
2060    char   (*cunit)[72];             // CUNITia keyvalues for each coord axis.
2061    char   (*ctype)[72];             // CTYPEia keyvalues for each coord axis.
2062
2063    double lonpole;           // LONPOLEa keyvalue.
2064    double latpole;           // LATPOLEa keyvalue.
2065
2066    double restfrq;           // RESTFRQa keyvalue.
2067    double restwav;           // RESTWAVa keyvalue.
2068
2069    int    npv;                        // Number of PVi_ma keywords, and the
2070    int    npvmax;            // number for which space was allocated.
2071    struct pvcard *pv;        // PVi_ma keywords for each i and m.
2072
2073    int    nps;                        // Number of PSi_ma keywords, and the
2074    int    npsmax;            // number for which space was allocated.
2075    struct pscard *ps;        // PSi_ma keywords for each i and m.
2076
2077    // Alternative header keyvalues (see the prologue above).
2078    //------------------------------------------------------------------------
2079    double *cd;               // CDi_ja linear transformation matrix.
2080    double *crota;            // CROTAi keyvalues for each coord axis.
2081    int    altlin;            // Alternative representations
2082                                       //   Bit 0: PCi_ja is present,
2083                                       //   Bit 1: CDi_ja is present,
2084                                       //   Bit 2: CROTAi is present.
2085    int    velref;            // AIPS velocity code, VELREF.
2086
2087    // Auxiliary coordinate system information of a general nature.  Not
2088    // used by WCSLIB.  Refer to the prologue comments above for a brief
2089    // explanation of these values.
2090    char   alt[4];
2091    int    colnum;
2092    int    *colax;
2093                                       // Auxiliary coordinate axis information.
2094    char   (*cname)[72];
2095    double *crder;
2096    double *csyer;
2097    double *czphs;
2098    double *cperi;
2099
2100    char   wcsname[72];
2101                                       // Time reference system and measurement.
2102    char   timesys[72], trefpos[72], trefdir[72], plephem[72];
2103    char   timeunit[72];
2104    char   dateref[72];
2105    double mjdref[2];
2106    double timeoffs;
2107                                       // Data timestamps and durations.
2108    char   dateobs[72], datebeg[72], dateavg[72], dateend[72];
2109    double mjdobs, mjdbeg, mjdavg, mjdend;
2110    double jepoch, bepoch;
2111    double tstart, tstop;
2112    double xposure, telapse;
2113                                       // Timing accuracy.
2114    double timsyer, timrder;
2115    double timedel, timepixr;
2116                                       // Spatial & celestial reference frame.
2117    double obsgeo[6];
2118    char   obsorbit[72];
2119    char   radesys[72];
2120    double equinox;
2121    char   specsys[72];
2122    char   ssysobs[72];
2123    double velosys;
2124    double zsource;
2125    char   ssyssrc[72];
2126    double velangl;
```

```
2127
2128    // Additional auxiliary coordinate system information of a specialist
2129    // nature.  Not used by WCSLIB.  Refer to the prologue comments above.
2130    struct auxprm *aux;
2131
2132    // Coordinate lookup tables (see the prologue above).
2133    //------------------------------------------------------------------------
2134    int     ntab;                       // Number of separate tables.
2135    int     nwtb;                       // Number of wtbarr structs.
2136    struct tabprm *tab;          // Tabular transformation parameters.
2137    struct wtbarr *wtb;          // Array of wtbarr structs.
2138
2139    //------------------------------------------------------------------------
2140    // Information derived from the FITS header keyvalues by wcsset().
2141    //------------------------------------------------------------------------
2142    char    lngtyp[8], lattyp[8];       // Celestial axis types, e.g. RA, DEC.
2143    int     lng, lat, spec;      // Longitude, latitude and spectral axis
2144                                       // indices (0-relative).
2145    int     cubeface;            // True if there is a CUBEFACE axis.
2146    int     *types;              // Coordinate type codes for each axis.
2147
2148    struct linprm lin;           //    Linear transformation parameters.
2149    struct celprm cel;           // Celestial transformation parameters.
2150    struct spcprm spc;           //  Spectral transformation parameters.
2151
2152    //------------------------------------------------------------------------
2153    //            THE REMAINDER OF THE WCSPRM STRUCT IS PRIVATE.
2154    //------------------------------------------------------------------------
2155
2156    // Error handling, if enabled.
2157    //------------------------------------------------------------------------
2158    struct wcserr *err;
2159
2160    // Memory management.
2161    //------------------------------------------------------------------------
2162    int     m_flag, m_naxis;
2163    double *m_crpix, *m_pc, *m_cdelt, *m_crval;
2164    char    (*m_cunit)[72], (*m_ctype)[72];
2165    struct pvcard *m_pv;
2166    struct pscard *m_ps;
2167    double *m_cd, *m_crota;
2168    int     *m_colax;
2169    char    (*m_cname)[72];
2170    double *m_crder, *m_csyer, *m_czphs, *m_cperi;
2171    struct auxprm *m_aux;
2172    struct tabprm *m_tab;
2173    struct wtbarr *m_wtb;
2174 };
2175
2176 // Size of the wcsprm struct in int units, used by the Fortran wrappers.
2177 #define WCSLEN (sizeof(struct wcsprm)/sizeof(int))
2178
2179
2180 int wcsnpv(int n);
2181
2182 int wcsnps(int n);
2183
2184 int wcsini(int alloc, int naxis, struct wcsprm *wcs);
2185
2186 int wcsinit(int alloc, int naxis, struct wcsprm *wcs, int npvmax, int npsmax,
2187             int ndpmax);
2188
2189 int wcsauxi(int alloc, struct wcsprm *wcs);
2190
2191 int wcssub(int alloc, const struct wcsprm *wcssrc, int *nsub, int axes[],
2192            struct wcsprm *wcsdst);
2193
2194 int wcscompare(int cmp, double tol, const struct wcsprm *wcs1,
2195                const struct wcsprm *wcs2, int *equal);
2196
2197 int wcsfree(struct wcsprm *wcs);
2198
2199 int wcstrim(struct wcsprm *wcs);
2200
2201 int wcssize(const struct wcsprm *wcs, int sizes[2]);
2202
2203 int auxsize(const struct auxprm *aux, int sizes[2]);
2204
2205 int wcsprt(const struct wcsprm *wcs);
2206
2207 int wcsperr(const struct wcsprm *wcs, const char *prefix);
2208
2209 int wcsbchk(struct wcsprm *wcs, int bounds);
2210
2211 int wcsset(struct wcsprm *wcs);
2212
2213 int wcsp2s(struct wcsprm *wcs, int ncoord, int nelem, const double pixcrd[],
```

```
2214              double imgcrd[], double phi[], double theta[], double world[],
2215              int stat[]);
2216
2217 int wcss2p(struct wcsprm *wcs, int ncoord, int nelem, const double world[],
2218              double phi[], double theta[], double imgcrd[], double pixcrd[],
2219              int stat[]);
2220
2221 int wcsmix(struct wcsprm *wcs, int mixpix, int mixcel, const double vspan[2],
2222              double vstep, int viter, double world[], double phi[],
2223              double theta[], double imgcrd[], double pixcrd[]);
2224
2225 int wcsccs(struct wcsprm *wcs, double lng2p1, double lat2p1, double lng1p2,
2226              const char *clng, const char *clat, const char *radesys,
2227              double equinox, const char *alt);
2228
2229 int wcssptr(struct wcsprm *wcs, int *i, char ctype[9]);
2230
2231 const char* wcslib_version(int vers[3]);
2232
2233 // Defined mainly for backwards compatibility, use wcssub() instead.
2234 #define wcscopy(alloc, wcssrc, wcsdst) wcssub(alloc, wcssrc, 0x0, 0x0, wcsdst)
2235
2236
2237 // Deprecated.
2238 #define wcsini_errmsg wcs_errmsg
2239 #define wcssub_errmsg wcs_errmsg
2240 #define wcscopy_errmsg wcs_errmsg
2241 #define wcsfree_errmsg wcs_errmsg
2242 #define wcsprt_errmsg wcs_errmsg
2243 #define wcsset_errmsg wcs_errmsg
2244 #define wcsp2s_errmsg wcs_errmsg
2245 #define wcss2p_errmsg wcs_errmsg
2246 #define wcsmix_errmsg wcs_errmsg
2247
2248 #ifdef __cplusplus
2249 #undef wtbarr
2250 }
2251 #endif
2252
2253 #endif // WCSLIB_WCS
```

19.25   wcserr.h File Reference

Data Structures

- struct wcserr

  Error message handling.

Macros

- #define ERRLEN (sizeof(struct wcserr)/sizeof(int))
- #define WCSERR_SET(status) err, status, function, __FILE__, __LINE__

  Fill in the contents of an error object.

Functions

- int wcserr_enable (int enable)

  Enable/disable error messaging.
- int wcserr_size (const struct wcserr *err, int sizes[2])

  Compute the size of a wcserr struct.
- int wcserr_prt (const struct wcserr *err, const char *prefix)

  Print a wcserr struct.
- int wcserr_clear (struct wcserr **err)

  Clear a wcserr struct.
- int wcserr_set (struct wcserr **err, int status, const char *function, const char *file, int line_no, const char *format,...)

  Fill in the contents of an error object.
- int wcserr_copy (const struct wcserr *src, struct wcserr *dst)

  Copy an error object.

19.25.1   Detailed Description

Most of the structs in WCSLIB contain a pointer to a wcserr struct as a member. Functions in WCSLIB that return an error status code can also allocate and set a detailed error message in this struct, which also identifies the function, source file, and line number where the error occurred.

For example:

```
struct prjprm prj;
wcserr_enable(1);
if (prjini(&prj)) {
  // Print the error message to stderr.
  wcsprintf_set(stderr);
  wcserr_prt(prj.err, 0x0);
}
```

A number of utility functions used in managing the wcserr struct are for **internal use only**. They are documented here solely as an aid to understanding the code. They are not intended for external use - the API may change without notice!

19.25.2   Macro Definition Documentation

19.25.2.1   ERRLEN   #define ERRLEN (sizeof(struct wcserr)/sizeof(int))

19.25.2.2   WCSERR_SET   #define WCSERR_SET(
              status) err, status, function, __FILE__, __LINE__

INTERNAL USE ONLY.

WCSERR_SET() is a preprocessor macro that helps to fill in the argument list of wcserr_set(). It takes status as an argument of its own and provides the name of the source file and the line number at the point where invoked. It assumes that the err and function arguments of wcserr_set() will be provided by variables of the same names.

19.25.3   Function Documentation

19.25.3.1   wcserr_enable()   int wcserr_enable (
              int *enable*)

wcserr_enable() enables or disables wcserr error messaging. By default it is disabled.

PLEASE NOTE: This function is not thread-safe.

Parameters

in	*enable*	If true (non-zero), enable error messaging, else disable it.

Returns

Status return value:

- 0: Error messaging is disabled.
- 1: Error messaging is enabled.

19.25.3.2 wcserr_size() `int wcserr_size (`
            `const struct wcserr * err,`
            `int sizes[2] )`

wcserr_size() computes the full size of a wcserr struct, including allocated memory.

Parameters

in	*err*	The error object.
		If NULL, the base size of the struct and the allocated size are both set to zero.
out	*sizes*	The first element is the base size of the struct as returned by sizeof(struct wcserr). The second element is the total allocated size of the message buffer, in bytes.

Returns

Status return value:

- 0: Success.

19.25.3.3 wcserr_prt() `int wcserr_prt (`
            `const struct wcserr * err,`
            `const char * prefix )`

wcserr_prt() prints the error message (if any) contained in a wcserr struct. It uses the wcsprintf() functions.

Parameters

in	*err*	The error object. If NULL, nothing is printed.
in	*prefix*	If non-NULL, each output line will be prefixed with this string.

Returns

Status return value:

- 0: Success.
- 2: Error messaging is not enabled.

19.25.3.4 wcserr_clear() `int wcserr_clear (`
     `struct` `wcserr` `** err )`

wcserr_clear() clears (deletes) a wcserr struct.

Parameters

in,out	*err*	The error object. If NULL, nothing is done. Set to NULL on return.

Returns

Status return value:

- 0: Success.

19.25.3.5 wcserr_set() `int wcserr_set (`
`        struct wcserr ** err,`
`        int status,`
`        const char * function,`
`        const char * file,`
`        int line_no,`
`        const char * format,`
`         ...  )`

INTERNAL USE ONLY.

wcserr_set() fills a wcserr struct with information about an error.

A convenience macro, WCSERR_SET, provides the source file and line number information automatically.

Parameters

in,out	*err*	Error object. If err is NULL, returns the status code given without setting an error message. If ∗err is NULL, allocates memory for a wcserr struct (provided that status is non-zero).
in	*status*	Numeric status code to set. If 0, then ∗err will be deleted and ∗err will be returned as NULL.
in	*function*	Name of the function generating the error. This must point to a constant string, i.e. in the initialized read-only data section ("data") of the executable.
in	*file*	Name of the source file generating the error. This must point to a constant string, i.e. in the initialized read-only data section ("data") of the executable such as given by the __FILE__ preprocessor macro.
in	*line_no*	Line number in the source file generating the error such as given by the __LINE__ preprocessor macro.
in	*format*	Format string of the error message. May contain printf-style %-formatting codes.
in	*...*	The remaining variable arguments are applied (like printf) to the format string to generate the error message.

Returns

The status return code passed in.

19.25.3.6 wcserr_copy() `int wcserr_copy (`
              `const struct wcserr * src,`
              `struct wcserr * dst )`

INTERNAL USE ONLY.

wcserr_copy() copies one error object to another. Use of this function should be avoided in general since the function, source file, and line number information copied to the destination may lose its context.

Parameters

`in`	*src*	Source error object. If src is NULL, dst is cleared.
`out`	*dst*	Destination error object. If NULL, no copy is made.

Returns

Numeric status code of the source error object.

19.26 wcserr.h

[Go to the documentation of this file.](#)
```
1 /*============================================================================
2   WCSLIB 7.11 - an implementation of the FITS WCS standard.
3   Copyright (C) 1995-2022, Mark Calabretta
4
5   This file is part of WCSLIB.
6
7   WCSLIB is free software: you can redistribute it and/or modify it under the
8   terms of the GNU Lesser General Public License as published by the Free
9   Software Foundation, either version 3 of the License, or (at your option)
10   any later version.
11
12   WCSLIB is distributed in the hope that it will be useful, but WITHOUT ANY
13   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
14   FOR A PARTICULAR PURPOSE.  See the GNU Lesser General Public License for
15   more details.
16
17   You should have received a copy of the GNU Lesser General Public License
18   along with WCSLIB.  If not, see http://www.gnu.org/licenses.
19
20   Author: Mark Calabretta, Australia Telescope National Facility, CSIRO.
21   Module author: Michael Droettboom
22   http://www.atnf.csiro.au/people/Mark.Calabretta
23   $Id: wcserr.h,v 7.11 2022/04/26 06:13:52 mcalabre Exp $
24 *============================================================================
25 *
26 * WCSLIB 7.11 - C routines that implement the FITS World Coordinate System
27 * (WCS) standard.  Refer to the README file provided with WCSLIB for an
28 * overview of the library.
29 *
30 * Summary of the wcserr routines
31 * ------------------------------
32 * Most of the structs in WCSLIB contain a pointer to a wcserr struct as a
33 * member.  Functions in WCSLIB that return an error status code can also
34 * allocate and set a detailed error message in this struct, which also
35 * identifies the function, source file, and line number where the error
36 * occurred.
37 *
38 * For example:
39 *
40 =     struct prjprm prj;
41 =     wcserr_enable(1);
42 =     if (prjini(&prj)) {
43 =       // Print the error message to stderr.
44 =       wcsprintf_set(stderr);
45 =       wcserr_prt(prj.err, 0x0);
46 =     }
47 *
48 * A number of utility functions used in managing the wcserr struct are for
49 * internal use only.  They are documented here solely as an aid to
50 * understanding the code.  They are not intended for external use - the API
51 * may change without notice!
```

```
 52 *
 53 *
 54 * wcserr struct - Error message handling
 55 * -------------------------------------
 56 * The wcserr struct contains the numeric error code, a textual description of
 57 * the error, and information about the function, source file, and line number
 58 * where the error was generated.
 59 *
 60 *   int status
 61 *     Numeric status code associated with the error, the meaning of which
 62 *     depends on the function that generated it.  See the documentation for
 63 *     the particular function.
 64 *
 65 *   int line_no
 66 *     Line number where the error occurred as given by the __LINE__
 67 *     preprocessor macro.
 68 *
 69 *   const char *function
 70 *     Name of the function where the error occurred.
 71 *
 72 *   const char *file
 73 *     Name of the source file where the error occurred as given by the
 74 *     __FILE__ preprocessor macro.
 75 *
 76 *   char *msg
 77 *     Informative error message.
 78 *
 79 *
 80 * wcserr_enable() - Enable/disable error messaging
 81 * ------------------------------------------------
 82 * wcserr_enable() enables or disables wcserr error messaging.  By default it
 83 * is disabled.
 84 *
 85 * PLEASE NOTE: This function is not thread-safe.
 86 *
 87 * Given:
 88 *   enable    int       If true (non-zero), enable error messaging, else
 89 *                       disable it.
 90 *
 91 * Function return value:
 92 *             int       Status return value:
 93 *                         0: Error messaging is disabled.
 94 *                         1: Error messaging is enabled.
 95 *
 96 *
 97 * wcserr_size() - Compute the size of a wcserr struct
 98 * ---------------------------------------------------
 99 * wcserr_size() computes the full size of a wcserr struct, including allocated
100 * memory.
101 *
102 * Given:
103 *   err       const struct wcserr*
104 *                       The error object.
105 *
106 *                       If NULL, the base size of the struct and the allocated
107 *                       size are both set to zero.
108 *
109 * Returned:
110 *   sizes     int[2]    The first element is the base size of the struct as
111 *                       returned by sizeof(struct wcserr).  The second element
112 *                       is the total allocated size of the message buffer, in
113 *                       bytes.
114 *
115 * Function return value:
116 *             int       Status return value:
117 *                         0: Success.
118 *
119 *
120 * wcserr_prt() - Print a wcserr struct
121 * ------------------------------------
122 * wcserr_prt() prints the error message (if any) contained in a wcserr struct.
123 * It uses the wcsprintf() functions.
124 *
125 * Given:
126 *   err       const struct wcserr*
127 *                       The error object.  If NULL, nothing is printed.
128 *
129 *   prefix    const char *
130 *                       If non-NULL, each output line will be prefixed with
131 *                       this string.
132 *
133 * Function return value:
134 *             int       Status return value:
135 *                         0: Success.
136 *                         2: Error messaging is not enabled.
137 *
138 *
```

```
139 * wcserr_clear() - Clear a wcserr struct
140 * ------------------------------------
141 * wcserr_clear() clears (deletes) a wcserr struct.
142 *
143 * Given and returned:
144 *   err      struct wcserr**
145 *                     The error object.  If NULL, nothing is done.  Set to
146 *                     NULL on return.
147 *
148 * Function return value:
149 *            int       Status return value:
150 *                      0: Success.
151 *
152 *
153 * wcserr_set() - Fill in the contents of an error object
154 * ------------------------------------------------------
155 * INTERNAL USE ONLY.
156 *
157 * wcserr_set() fills a wcserr struct with information about an error.
158 *
159 * A convenience macro, WCSERR_SET, provides the source file and line number
160 * information automatically.
161 *
162 * Given and returned:
163 *   err      struct wcserr**
164 *                     Error object.
165 *
166 *                     If err is NULL, returns the status code given without
167 *                     setting an error message.
168 *
169 *                     If *err is NULL, allocates memory for a wcserr struct
170 *                     (provided that status is non-zero).
171 *
172 * Given:
173 *   status   int       Numeric status code to set.  If 0, then *err will be
174 *                     deleted and *err will be returned as NULL.
175 *
176 *   function  const char *
177 *                     Name of the function generating the error.  This
178 *                     must point to a constant string, i.e. in the
179 *                     initialized read-only data section ("data") of the
180 *                     executable.
181 *
182 *   file     const char *
183 *                     Name of the source file generating the error.  This
184 *                     must point to a constant string, i.e. in the
185 *                     initialized read-only data section ("data") of the
186 *                     executable such as given by the __FILE__ preprocessor
187 *                     macro.
188 *
189 *   line_no   int       Line number in the source file generating the error
190 *                     such as given by the __LINE__ preprocessor macro.
191 *
192 *   format    const char *
193 *                     Format string of the error message.  May contain
194 *                     printf-style %-formatting codes.
195 *
196 *   ...      mixed    The remaining variable arguments are applied (like
197 *                     printf) to the format string to generate the error
198 *                     message.
199 *
200 * Function return value:
201 *            int       The status return code passed in.
202 *
203 *
204 * wcserr_copy() - Copy an error object
205 * ------------------------------------
206 * INTERNAL USE ONLY.
207 *
208 * wcserr_copy() copies one error object to another.  Use of this function
209 * should be avoided in general since the function, source file, and line
210 * number information copied to the destination may lose its context.
211 *
212 * Given:
213 *   src      const struct wcserr*
214 *                     Source error object.  If src is NULL, dst is cleared.
215 *
216 * Returned:
217 *   dst      struct wcserr*
218 *                     Destination error object.  If NULL, no copy is made.
219 *
220 * Function return value:
221 *            int       Numeric status code of the source error object.
222 *
223 *
224 * WCSERR_SET() macro - Fill in the contents of an error object
225 * ------------------------------------------------------------
```

```
226  * INTERNAL USE ONLY.
227  *
228  * WCSERR_SET() is a preprocessor macro that helps to fill in the argument list
229  * of wcserr_set().  It takes status as an argument of its own and provides the
230  * name of the source file and the line number at the point where invoked.  It
231  * assumes that the err and function arguments of wcserr_set() will be provided
232  * by variables of the same names.
233  *
234  *===========================================================================*/
235
236  #ifndef WCSLIB_WCSERR
237  #define WCSLIB_WCSERR
238
239  #ifdef __cplusplus
240  extern "C" {
241  #endif
242
243  struct wcserr {
244    int   status;                 // Status code for the error.
245    int   line_no;                // Line number where the error occurred.
246    const char *function;         // Function name.
247    const char *file;             // Source file name.
248    char *msg;                    // Informative error message.
249  };
250
251  // Size of the wcserr struct in int units, used by the Fortran wrappers.
252  #define ERRLEN (sizeof(struct wcserr)/sizeof(int))
253
254  int wcserr_enable(int enable);
255
256  int wcserr_size(const struct wcserr *err, int sizes[2]);
257
258  int wcserr_prt(const struct wcserr *err, const char *prefix);
259
260  int wcserr_clear(struct wcserr **err);
261
262
263  // INTERNAL USE ONLY -------------------------------------------------------
264
265  int wcserr_set(struct wcserr **err, int status, const char *function,
266    const char *file, int line_no, const char *format, ...);
267
268  int wcserr_copy(const struct wcserr *src, struct wcserr *dst);
269
270  // Convenience macro for invoking wcserr_set().
271  #define WCSERR_SET(status) err, status, function, __FILE__, __LINE__
272
273  #ifdef __cplusplus
274  }
275  #endif
276
277  #endif // WSCLIB_WCSERR
```

19.27 wcsfix.h File Reference

```
#include "wcs.h"
#include "wcserr.h"
```

Macros

- #define CDFIX 0

  Index of cdfix() status value in vector returned by wcsfix().
- #define DATFIX 1

  Index of datfix() status value in vector returned by wcsfix().
- #define OBSFIX 2
- #define UNITFIX 3

  Index of unitfix() status value in vector returned by wcsfix().
- #define SPCFIX 4

  Index of spcfix() status value in vector returned by wcsfix().
- #define CELFIX 5

Index of celfix() status value in vector returned by wcsfix().
- #define CYLFIX 6

    Index of cylfix() status value in vector returned by wcsfix().
- #define NWCSFIX 7

    Number of elements in the status vector returned by wcsfix().
- #define cylfix_errmsg wcsfix_errmsg

    Deprecated.

Enumerations

- enum wcsfix_errmsg_enum {
  FIXERR_OBSGEO_FIX = -5 , FIXERR_DATE_FIX = -4 , FIXERR_SPC_UPDATE = -3 , FIXERR_UNITS_ALIAS
  = -2 ,
  FIXERR_NO_CHANGE = -1 , FIXERR_SUCCESS = 0 , FIXERR_NULL_POINTER = 1 , FIXERR_MEMORY
  = 2 ,
  FIXERR_SINGULAR_MTX = 3 , FIXERR_BAD_CTYPE = 4 , FIXERR_BAD_PARAM = 5 , FIXERR_BAD_COORD_TRANS
  = 6 ,
  FIXERR_ILL_COORD_TRANS = 7 , FIXERR_BAD_CORNER_PIX = 8 , FIXERR_NO_REF_PIX_COORD =
  9 , FIXERR_NO_REF_PIX_VAL = 10 }

Functions

- int wcsfix (int ctrl, const int naxis[ ], struct wcsprm ∗wcs, int stat[ ])

    Translate a non-standard WCS struct.
- int wcsfixi (int ctrl, const int naxis[ ], struct wcsprm ∗wcs, int stat[ ], struct wcserr info[ ])

    Translate a non-standard WCS struct.
- int cdfix (struct wcsprm ∗wcs)

    Fix erroneously omitted CDi_ja keywords.
- int datfix (struct wcsprm ∗wcs)

    Translate DATE−OBS and derive MJD−OBS or vice versa.
- int obsfix (int ctrl, struct wcsprm ∗wcs)

    complete the OBSGEO-[XYZLBH] vector of observatory coordinates.
- int unitfix (int ctrl, struct wcsprm ∗wcs)

    Correct aberrant CUNITia keyvalues.
- int spcfix (struct wcsprm ∗wcs)

    Translate AIPS-convention spectral types.
- int celfix (struct wcsprm ∗wcs)

    Translate AIPS-convention celestial projection types.
- int cylfix (const int naxis[ ], struct wcsprm ∗wcs)

    Fix malformed cylindrical projections.
- int wcspcx (struct wcsprm ∗wcs, int dopc, int permute, double rotn[2])

    regularize PCi_j.

Variables

- const char ∗ wcsfix_errmsg [ ]

    Status return messages.

19.27.1 Detailed Description

Routines in this suite identify and translate various forms of construct known to occur in FITS headers that violate the FITS World Coordinate System (WCS) standard described in

```
"Representations of world coordinates in FITS",
Greisen, E.W., & Calabretta, M.R. 2002, A&A, 395, 1061 (WCS Paper I)
"Representations of celestial coordinates in FITS",
Calabretta, M.R., & Greisen, E.W. 2002, A&A, 395, 1077 (WCS Paper II)
"Representations of spectral coordinates in FITS",
Greisen, E.W., Calabretta, M.R., Valdes, F.G., & Allen, S.L.
2006, A&A, 446, 747 (WCS Paper III)
"Representations of time coordinates in FITS -
 Time and relative dimension in space",
Rots, A.H., Bunclark, P.S., Calabretta, M.R., Allen, S.L.,
Manchester, R.N., & Thompson, W.T. 2015, A&A, 574, A36 (WCS Paper VII)
```

Repairs effected by these routines range from the translation of non-standard values for standard WCS keywords, to the repair of malformed coordinate representations. Some routines are also provided to check the consistency of pairs of keyvalues that define the same measure in two different ways, for example, as a date and an MJD.

A separate routine, wcspcx(), "regularizes" the linear transformation matrix component (PCi_j) of the coordinate transformation to make it more human- readable. Where a coordinate description was constructed from CDi_j, it decomposes it into PCi_j + CDELTi in a meaningful way. Optionally, it can also diagonalize the PCi_j matrix (as far as possible), i.e. undo a transposition of axes in the intermediate pixel coordinate system.

Non-standard keyvalues:
AIPS-convention celestial projection types, **NCP** and **GLS**, and spectral types, **'FREQ−LSR'**, **'FELO−HEL'**, etc., set in **CTYPE**ia are translated on-the-fly by wcsset() but without modifying the relevant ctype[], pv[] or specsys members of the wcsprm struct. That is, only the information extracted from ctype[] is translated when wcsset() fills in wcsprm::cel (celprm struct) or wcsprm::spc (spcprm struct).

On the other hand, these routines do change the values of wcsprm::ctype[], wcsprm::pv[], wcsprm::specsys and other wcsprm struct members as appropriate to produce the same result as if the FITS header itself had been translated.

Auxiliary WCS header information not used directly by WCSLIB may also be translated. For example, the older **DATE−OBS** date format (wcsprm::dateobs) is recast to year-2000 standard form, and **MJD−OBS** (wcsprm::mjdobs) will be deduced from it if not already set.

Certain combinations of keyvalues that result in malformed coordinate systems, as described in Sect. 7.3.4 of Paper I, may also be repaired. These are handled by cylfix().

Non-standard keywords:
The AIPS-convention CROTAn keywords are recognized as quasi-standard and as such are accomodated by wcsprm::crota[] and translated to wcsprm::pc[][] by wcsset(). These are not dealt with here, nor are any other non-standard keywords since these routines work only on the contents of a wcsprm struct and do not deal with FITS headers per se. In particular, they do not identify or translate **CD00i00j**, **PC00i00j**, **PROJPn**, **EPOCH**, **VELREF** or **VSOURCEa** keywords; this may be done by the FITS WCS header parser supplied with WCSLIB, refer to wcshdr.h.

wcsfix() and wcsfixi() apply all of the corrections handled by the following specific functions, which may also be invoked separately:

- cdfix(): Sets the diagonal element of the **CD**i_ja matrix to 1.0 if all **CD**i_ja keywords associated with a particular axis are omitted.

- datfix(): recast an older **DATE−OBS** date format in dateobs to year-2000 standard form. Derive dateref from mjdref if not already set. Alternatively, if dateref is set and mjdref isn't, then derive mjdref from it. If both are set, then check consistency. Likewise for dateobs and mjdobs; datebeg and mjdbeg; dateavg and mjdavg; and dateend and mjdend.

  - obsfix(): if only one half of obsgeo[] is set, then derive the other half from it. If both halves are set, then check consistency.

  - unitfix(): translate some commonly used but non-standard unit strings in the **CUNIT**ia keyvalues, e.g. **'DEG'** -> **'deg'**.

  - spcfix(): translate AIPS-convention spectral types, **'FREQ−LSR'**, **'FELO−HEL'**, etc., in ctype[] as set from **CTYPE**ia.

  - celfix(): translate AIPS-convention celestial projection types, **NCP** and **GLS**, in ctype[] as set from **CTYPE**ia.

  - cylfix(): fixes WCS keyvalues for malformed cylindrical projections that suffer from the problem described in Sect. 7.3.4 of Paper I.

19.27.2   Macro Definition Documentation

19.27.2.1   CDFIX #define CDFIX 0

Index of the status value returned by cdfix() in the status vector returned by wcsfix().

19.27.2.2   DATFIX #define DATFIX 1

Index of the status value returned by datfix() in the status vector returned by wcsfix().

19.27.2.3   OBSFIX #define OBSFIX 2

19.27.2.4   UNITFIX #define UNITFIX 3

Index of the status value returned by unitfix() in the status vector returned by wcsfix().

19.27.2.5   SPCFIX #define SPCFIX 4

Index of the status value returned by spcfix() in the status vector returned by wcsfix().

19.27.2.6   CELFIX #define CELFIX 5

Index of the status value returned by celfix() in the status vector returned by wcsfix().

19.27.2.7   CYLFIX #define CYLFIX 6

Index of the status value returned by cylfix() in the status vector returned by wcsfix().

19.27.2.8   NWCSFIX #define NWCSFIX 7

Number of elements in the status vector returned by wcsfix().

19.27.2.9 cylfix_errmsg `#define cylfix_errmsg` `wcsfix_errmsg`

Deprecated Added for backwards compatibility, use wcsfix_errmsg directly now instead.

19.27.3 Enumeration Type Documentation

19.27.3.1 wcsfix_errmsg_enum `enum` `wcsfix_errmsg_enum`

Enumerator

FIXERR_OBSGEO_FIX	
FIXERR_DATE_FIX	
FIXERR_SPC_UPDATE	
FIXERR_UNITS_ALIAS	
FIXERR_NO_CHANGE	
FIXERR_SUCCESS	
FIXERR_NULL_POINTER	
FIXERR_MEMORY	
FIXERR_SINGULAR_MTX	
FIXERR_BAD_CTYPE	
FIXERR_BAD_PARAM	
FIXERR_BAD_COORD_TRANS	
FIXERR_ILL_COORD_TRANS	
FIXERR_BAD_CORNER_PIX	
FIXERR_NO_REF_PIX_COORD	
FIXERR_NO_REF_PIX_VAL	

19.27.4   Function Documentation

19.27.4.1   wcsfix()   int wcsfix (
            int *ctrl,*
            const int *naxis[],*
            struct wcsprm * *wcs,*
            int *stat[]*)

wcsfix() is identical to **wcsfixi**(), but lacks the info argument.

19.27.4.2   wcsfixi()   int wcsfixi (
            int *ctrl,*
            const int *naxis[],*
            struct wcsprm * *wcs,*
            int *stat[],*
            struct wcserr *info[]*)

wcsfixi() applies all of the corrections handled separately by cdfix(), datfix(), obsfix(), unitfix(), spcfix(), celfix(), and cylfix().

Parameters

in	*ctrl*	Do potentially unsafe translations of non-standard unit strings as described in the usage notes to wcsutrn().
in	*naxis*	Image axis lengths. If this array pointer is set to zero then cylfix() will not be invoked.
in,out	*wcs*	Coordinate transformation parameters.
out	*stat*	Status returns from each of the functions. Use the preprocessor macros NWCSFIX to dimension this vector and CDFIX, DATFIX, **OBS**FIX, UNITFIX, SPCFIX, CELFIX, and CYLFIX to access its elements. A status value of -2 is set for functions that were not invoked.

Parameters

out	*info*	Status messages from each of the functions. Use the preprocessor macros NWCSFIX to dimension this vector and CDFIX, DATFIX, **OBS**FIX, UNITFIX, SPCFIX, CELFIX, and CYLFIX to access its elements.
		Note that the memory allocated by **wcsfixi**() for the message in each wcserr struct (wcserr::msg, if non-zero) must be freed by the user. See wcsdealloc().

Returns

Status return value:

- 0: Success.

- 1: One or more of the translation functions returned an error.

19.27.4.3 cdfix() int cdfix (
        struct wcsprm * *wcs*)

cdfix() sets the diagonal element of the **CD**i_ja matrix to unity if all **CD**i_ja keywords associated with a given axis were omitted. According to WCS Paper I, if any **CD**i_ja keywords at all are given in a FITS header then those not given default to zero. This results in a singular matrix with an intersecting row and column of zeros.

cdfix() is expected to be invoked before wcsset(), which will fail if these errors have not been corrected.

Parameters

in,out	*wcs*	Coordinate transformation parameters.

Returns

Status return value:

- -1: No change required (not an error).
- 0: Success.
- 1: Null wcsprm pointer passed.

19.27.4.4 datfix() int datfix (
        struct wcsprm * *wcs*)

datfix() translates the old **DATE−OBS** date format set in wcsprm::dateobs to year-2000 standard form (*yyyy-mm-dd**T**hh:mm:ss*). It derives wcsprm::dateref from wcsprm::mjdref if not already set. Alternatively, if dateref is set and mjdref isn't, then it derives mjdref from it. If both are set but disagree by more than 0.001 day (86.4 seconds) then an error status is returned. Likewise for wcsprm::dateobs and wcsprm::mjdobs; wcsprm::datebeg and wcsprm::mjdbeg; wcsprm::dateavg and wcsprm::mjdavg; and wcsprm::dateend and wcsprm::mjdend.

If neither dateobs nor mjdobs are set, but wcsprm::jepoch (primarily) or wcsprm::bepoch is, then both are derived from it. If jepoch and/or bepoch are set but disagree with dateobs or mjdobs by more than 0.000002 year (63.2 seconds), an informative message is produced.

The translations done by **datfix**() do not affect and are not affected by wcsset().

Parameters

in,out	*wcs*	Coordinate transformation parameters. wcsprm::dateref and/or wcsprm::mjdref may be changed. wcsprm::dateobs and/or wcsprm::mjdobs may be changed. wcsprm::datebeg and/or wcsprm::mjdbeg may be changed. wcsprm::dateavg and/or wcsprm::mjdavg may be changed. wcsprm::dateend and/or wcsprm::mjdend may be changed.

Returns

Status return value:

- -1: No change required (not an error).
- 0: Success.
- 1: Null wcsprm pointer passed.
- 5: Invalid parameter value.

For returns >= 0, a detailed message, whether informative or an error message, may be set in wcsprm::err if enabled, see wcserr_enable(), with wcsprm::err.status set to FIXERR_DATE_FIX.

Notes:

1. The MJD algorithms used by **datfix**() are from D.A. Hatcher, 1984, QJRAS, 25, 53-55, as modified by P.T. Wallace for use in SLALIB subroutines *CLDJ* and *DJCL*.

19.27.4.5   obsfix()   int obsfix (
        int *ctrl,*
        struct wcsprm * *wcs*)

obsfix() completes the wcsprm::obsgeo vector of observatory coordinates. That is, if only the (x,y,z) Cartesian coordinate triplet or the (l,b,h) geodetic coordinate triplet are set, then it derives the other triplet from it. If both triplets are set, then it checks for consistency at the level of 1 metre.

The operations done by **obsfix**() do not affect and are not affected by wcsset().

Parameters

in	*ctrl*	Flag that controls behaviour if one triplet is defined and the other is only partially defined: - 0: Reset only the undefined elements of an incomplete coordinate triplet. - 1: Reset all elements of an incomplete triplet. - 2: Don't make any changes, check for consistency only. Returns an error if either of the two triplets is incomplete.
in,out	*wcs*	Coordinate transformation parameters. wcsprm::obsgeo may be changed.

Returns

Status return value:

- -1: No change required (not an error).
- 0: Success.
- 1: Null wcsprm pointer passed.
- 5: Invalid parameter value.

For returns >= 0, a detailed message, whether informative or an error message, may be set in wcsprm::err if enabled, see wcserr_enable(), with wcsprm::err.status set to FIXERR_**OBS**_FIX.

Notes:

1. While the International Terrestrial Reference System (ITRS) is based solely on Cartesian coordinates, it recommends the use of the GRS80 ellipsoid in converting to geodetic coordinates. However, while WCS Paper III recommends ITRS Cartesian coordinates, Paper VII prescribes the use of the IAU(1976) ellipsoid for geodetic coordinates, and consequently that is what is used here.

2. For reference, parameters of commonly used global reference ellipsoids:
   ```
      a (m)        1/f                Standard
   ---------  -------------  ------------------------------
   6378140    298.2577        IAU(1976)
   6378137    298.257222101   GRS80
   6378137    298.257223563   WGS84
   6378136    298.257         IERS(1989)
   6378136.6  298.25642       IERS(2003,2010), IAU(2009/2012)
   ```

   where f = (a - b) / a is the flattening, and a and b are the semi-major and semi-minor radii in metres.

3. The transformation from geodetic (lng,lat,hgt) to Cartesian (x,y,z) is
   ```
   x = (n + hgt)*coslng*coslat,
   y = (n + hgt)*sinlng*coslat,
   z = (n*(1.0 - e^2) + hgt)*sinlat,
   ```

   where the "prime vertical radius", n, is a function of latitude
   ```
   n = a / sqrt(1 - (e*sinlat)^2),
   ```

   and a, the equatorial radius, and e^2 = (2 - f)∗f, the (first) eccentricity of the ellipsoid, are constants. **obsfix**() inverts these iteratively by writing
   ```
      x = rho*coslng*coslat,
      y = rho*sinlng*coslat,
   zeta = rho*sinlat,
   ```

   where
   ```
    rho = n + hgt,
        = sqrt(x^2 + y^2 + zeta^2),
   zeta = z / (1 - n*e^2/rho),
   ```

   and iterating over the value of zeta. Since e is small, a good first approximation is given by zeta = z.

19.27.4.6 unitfix() `int unitfix (`
            `int ctrl,`
            `struct wcsprm * wcs )`

unitfix() applies wcsutrn() to translate non-standard **CUNIT**ia keyvalues, e.g. '**DEG**' -> '**deg**', also stripping off unnecessary whitespace.

unitfix() is expected to be invoked before wcsset(), which will fail if non-standard **CUNIT**ia keyvalues have not been translated.

Parameters

in	*ctrl*	Do potentially unsafe translations described in the usage notes to wcsutrn().
in,out	*wcs*	Coordinate transformation parameters.

Returns

Status return value:

- -1: No change required (not an error).
- 0: Success (an alias was applied).
- 1: Null wcsprm pointer passed.

When units are translated (i.e. 0 is returned), an informative message is set in wcsprm::err if enabled, see wcserr_enable(), with wcsprm::err.status set to FIXERR_UNITS_ALIAS.

19.27.4.7   spcfix() `int spcfix (`
                `struct wcsprm * wcs )`

spcfix() translates AIPS-convention spectral coordinate types, '{**FREQ**,**FELO**,**VELO**}-{**LSR**,**HEL**,**OBS**}' (e.g. 'FREQ-OBS', **'FELO−HEL'**, 'VELO-LSR') set in wcsprm::ctype[], subject to **VELREF** set in wcsprm::velref.

Note that if wcs::specsys is already set then it will not be overridden.

AIPS-convention spectral types set in **CTYPE**ia are translated on-the-fly by wcsset() but without modifying wcsprm::ctype[] or wcsprm::specsys. That is, only the information extracted from wcsprm::ctype[] is translated when wcsset() fills in wcsprm::spc (spcprm struct). **spcfix**() modifies wcsprm::ctype[] so that if the header is subsequently written out, e.g. by wcshdo(), then it will contain translated **CTYPE**ia keyvalues.

The operations done by **spcfix**() do not affect and are not affected by wcsset().

Parameters

in,out	*wcs*	Coordinate transformation parameters. wcsprm::ctype[] and/or wcsprm::specsys may be changed.

Returns

Status return value:

- -1: No change required (not an error).
- 0: Success.
- 1: Null wcsprm pointer passed.
- 2: Memory allocation failed.
- 3: Linear transformation matrix is singular.
- 4: Inconsistent or unrecognized coordinate axis types.
- 5: Invalid parameter value.
- 6: Invalid coordinate transformation parameters.
- 7: Ill-conditioned coordinate transformation parameters.

For returns $>= 0$, a detailed message, whether informative or an error message, may be set in wcsprm::err if enabled, see wcserr_enable(), with wcsprm::err.status set to FIXERR_SPC_UPDTE.

19.27.4.8 celfix() `int celfix (`
    `struct wcsprm * wcs )`

celfix() translates AIPS-convention celestial projection types, **NCP** and **GLS**, set in the ctype[] member of the wc-sprm struct.

Two additional pv[] keyvalues are created when translating **NCP**, and three are created when translating **GLS** with non-zero reference point. If the pv[] array was initially allocated by wcsini() then the array will be expanded if necessary. Otherwise, error 2 will be returned if sufficient empty slots are not already available for use.

AIPS-convention celestial projection types set in **CTYPE**ia are translated on-the-fly by wcsset() but without modifying wcsprm::ctype[], wcsprm::pv[], or wcsprm::npv. That is, only the information extracted from wcsprm::ctype[] is translated when wcsset() fills in wcsprm::cel (celprm struct). **celfix**() modifies wcsprm::ctype[], wcsprm::pv[], and wcsprm::npv so that if the header is subsequently written out, e.g. by wcshdo(), then it will contain translated **CTYPE**ia keyvalues and the relevant **PV**i_ma.

The operations done by **celfix**() do not affect and are not affected by wcsset(). However, it uses information in the wcsprm struct provided by wcsset(), and will invoke it if necessary.

Parameters

`in,out`	*wcs*	Coordinate transformation parameters. wcsprm::ctype[] and/or wcsprm::pv[] may be changed.

Returns

Status return value:

- -1: No change required (not an error).
- 0: Success.
- 1: Null wcsprm pointer passed.
- 2: Memory allocation failed.
- 3: Linear transformation matrix is singular.
- 4: Inconsistent or unrecognized coordinate axis types.
- 5: Invalid parameter value.
- 6: Invalid coordinate transformation parameters.
- 7: Ill-conditioned coordinate transformation parameters.

For returns > 1, a detailed error message is set in wcsprm::err if enabled, see wcserr_enable().

19.27.4.9 cylfix() `int cylfix (`
    `const int naxis[],`
    `struct wcsprm * wcs )`

cylfix() fixes WCS keyvalues for malformed cylindrical projections that suffer from the problem described in Sect. 7.3.4 of Paper I.

cylfix() requires the wcsprm struct to have been set up by wcsset(), and will invoke it if necessary. After modification, the struct is reset on return with an explicit call to wcsset().

Parameters

in	*naxis*	Image axis lengths.
in,out	*wcs*	Coordinate transformation parameters.

Returns

Status return value:

- -1: No change required (not an error).
- 0: Success.
- 1: Null wcsprm pointer passed.
- 2: Memory allocation failed.
- 3: Linear transformation matrix is singular.
- 4: Inconsistent or unrecognized coordinate axis types.
- 5: Invalid parameter value.
- 6: Invalid coordinate transformation parameters.
- 7: Ill-conditioned coordinate transformation parameters.
- 8: All of the corner pixel coordinates are invalid.
- 9: Could not determine reference pixel coordinate.
- 10: Could not determine reference pixel value.

For returns > 1, a detailed error message is set in wcsprm::err if enabled, see wcserr_enable().

19.27.4.10   wcspcx() `int wcspcx (`
    `struct wcsprm * wcs,`
    `int dopc,`
    `int permute,`
    `double rotn[2] )`

wcspcx() "regularizes" the linear transformation matrix component of the coordinate transformation (`PC`i_ja) to make it more human-readable.

Normally, upon encountering a FITS header containing a `CD`i_ja matrix, wcsset() simply treats it as `PC`i_ja and sets `CDELT`ia to unity. However, **wcspcx**() decomposes `CD`i_ja into `PC`i_ja and `CDELT`ia in such a way that `CDELT`ia form meaningful scaling parameters. In practice, the residual `PC`i_ja matrix will often then be orthogonal, i.e. unity, or describing a pure rotation, axis permutation, or reflection, or a combination thereof.

The decomposition is based on normalizing the length in the transformed system (i.e. intermediate pixel coordinates) of the orthonormal basis vectors of the pixel coordinate system. This deviates slightly from the prescription given by Eq. (4) of WCS Paper I, namely $\text{Sum}(j=1,N)(\mathbf{PC}i\_ja)^2 = 1$, in replacing the sum over j with the sum over i. Consequently, the columns of `PC`i_ja will consist of unit vectors. In practice, especially in cubes and higher dimensional images, at least some pairs of these unit vectors, if not all, will often be orthogonal or close to orthogonal.

The sign of `CDELT`ia is chosen to make the `PC`i_ja matrix as close to the, possibly permuted, unit matrix as possible, except that where the coordinate description contains a pair of celestial axes, the sign of `CDELT`ia is set negative for the longitude axis and positive for the latitude axis.

Optionally, rows of the `PC`i_ja matrix may also be permuted to diagonalize it as far as possible, thus undoing any transposition of axes in the intermediate pixel coordinate system.

If the coordinate description contains a celestial plane, then the angle of rotation of each of the basis vectors associated with the celestial axes is returned. For a pure rotation the two angles should be identical. Any difference between them is a measure of axis skewness.

The decomposition is not performed for axes involving a sequent distortion function that is defined in terms of **CD**i_ja, such as TPV, TNX, or ZPX, which always are. The independent variables of the polynomial are therefore intermediate world coordinates rather than intermediate pixel coordinates. Because sequent distortions are always applied before **CDELT**ia, if **CD**i_ja was translated to **PC**i_ja plus **CDELT**ia, then the distortion would be altered unless the polynomial coefficients were also adjusted to account for the change of scale.

wcspcx() requires the wcsprm struct to have been set up by wcsset(), and will invoke it if necessary. The wcsprm struct is reset on return with an explicit call to wcsset().

Parameters

in,out	*wcs*	Coordinate transformation parameters.
in	*dopc*	If 1, then **PC**i_ja and **CDELT**ia, as given, will be recomposed according to the above prescription. If 0, the operation is restricted to decomposing **CD**i_ja.
in	*permute*	If 1, then after decomposition (or recomposition), permute rows of **PC**i_ja to make the axes of the intermediate pixel coordinate system match as closely as possible those of the pixel coordinates. That is, make it as close to a diagonal matrix as possible. However, celestial axes are special in always being paired, with the longitude axis preceding the latitude axis. All WCS entities indexed by i, such as **CTYPE**ia, **CRVAL**ia, **CDELT**ia, etc., including coordinate lookup tables, will also be permuted as necessary to account for the change to **PC**i_ja. This does not apply to **CRPIX**ja, nor prior distortion functions. These operate on pixel coordinates, which are not affected by the permutation.
out	*rotn*	with the celestial axes. For a pure rotation the two angles should be identical. Any difference between them is a measure of axis skewness. May be set to the NULL pointer if this information is not required.

Returns

Status return value:

- 0: Success.
- 1: Null wcsprm pointer passed.
- 2: Memory allocation failed.
- 5: CDi_j matrix not used.
- 6: Sequent distortion function present.

19.27.5 Variable Documentation

19.27.5.1 wcsfix_errmsg const char * wcsfix_errmsg[] [extern]

Error messages to match the status value returned from each function.

19.28 wcsfix.h

```
1  /*============================================================================
2    WCSLIB 7.11 - an implementation of the FITS WCS standard.
3    Copyright (C) 1995-2022, Mark Calabretta
4
5    This file is part of WCSLIB.
6
7    WCSLIB is free software: you can redistribute it and/or modify it under the
8    terms of the GNU Lesser General Public License as published by the Free
9    Software Foundation, either version 3 of the License, or (at your option)
10   any later version.
11
12   WCSLIB is distributed in the hope that it will be useful, but WITHOUT ANY
13   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
14   FOR A PARTICULAR PURPOSE.  See the GNU Lesser General Public License for
15   more details.
16
17   You should have received a copy of the GNU Lesser General Public License
18   along with WCSLIB.  If not, see http://www.gnu.org/licenses.
19
20   Author: Mark Calabretta, Australia Telescope National Facility, CSIRO.
21   http://www.atnf.csiro.au/people/Mark.Calabretta
22   $Id: wcsfix.h,v 7.11 2022/04/26 06:13:52 mcalabre Exp $
23  *============================================================================
24  *
25  * WCSLIB 7.11 - C routines that implement the FITS World Coordinate System
26  * (WCS) standard.  Refer to the README file provided with WCSLIB for an
27  * overview of the library.
28  *
29  *
30  * Summary of the wcsfix routines
31  * ------------------------------
32  * Routines in this suite identify and translate various forms of construct
33  * known to occur in FITS headers that violate the FITS World Coordinate System
34  * (WCS) standard described in
35  *
36  =   "Representations of world coordinates in FITS",
37  =   Greisen, E.W., & Calabretta, M.R. 2002, A&A, 395, 1061 (WCS Paper I)
38  =
39  =   "Representations of celestial coordinates in FITS",
40  =   Calabretta, M.R., & Greisen, E.W. 2002, A&A, 395, 1077 (WCS Paper II)
41  =
42  =   "Representations of spectral coordinates in FITS",
43  =   Greisen, E.W., Calabretta, M.R., Valdes, F.G., & Allen, S.L.
44  =   2006, A&A, 446, 747 (WCS Paper III)
45  =
46  =   "Representations of time coordinates in FITS -
47  =    Time and relative dimension in space",
48  =   Rots, A.H., Bunclark, P.S., Calabretta, M.R., Allen, S.L.,
49  =   Manchester, R.N., & Thompson, W.T. 2015, A&A, 574, A36 (WCS Paper VII)
50  *
51  * Repairs effected by these routines range from the translation of
52  * non-standard values for standard WCS keywords, to the repair of malformed
53  * coordinate representations.  Some routines are also provided to check the
54  * consistency of pairs of keyvalues that define the same measure in two
55  * different ways, for example, as a date and an MJD.
56  *
57  * A separate routine, wcspcx(), "regularizes" the linear transformation matrix
58  * component (PCi_j) of the coordinate transformation to make it more human-
59  * readable.  Where a coordinate description was constructed from CDi_j, it
60  * decomposes it into PCi_j + CDELTi in a meaningful way.  Optionally, it can
61  * also diagonalize the PCi_j matrix (as far as possible), i.e. undo a
62  * transposition of axes in the intermediate pixel coordinate system.
63  *
64  * Non-standard keyvalues:
65  * -----------------------
66  *   AIPS-convention celestial projection types, NCP and GLS, and spectral
67  *   types, 'FREQ-LSR', 'FELO-HEL', etc., set in CTYPEia are translated
68  *   on-the-fly by wcsset() but without modifying the relevant ctype[], pv[] or
69  *   specsys members of the wcsprm struct.  That is, only the information
70  *   extracted from ctype[] is translated when wcsset() fills in wcsprm::cel
71  *   (celprm struct) or wcsprm::spc (spcprm struct).
72  *
73  *   On the other hand, these routines do change the values of wcsprm::ctype[],
74  *   wcsprm::pv[], wcsprm::specsys and other wcsprm struct members as
75  *   appropriate to produce the same result as if the FITS header itself had
76  *   been translated.
77  *
78  *   Auxiliary WCS header information not used directly by WCSLIB may also be
79  *   translated.  For example, the older DATE-OBS date format (wcsprm::dateobs)
80  *   is recast to year-2000 standard form, and MJD-OBS (wcsprm::mjdobs) will be
81  *   deduced from it if not already set.
82  *
83  *   Certain combinations of keyvalues that result in malformed coordinate
```

```
 84 *   systems, as described in Sect. 7.3.4 of Paper I, may also be repaired.
 85 *   These are handled by cylfix().
 86 *
 87 * Non-standard keywords:
 88 * ---------------------
 89 *   The AIPS-convention CROTAn keywords are recognized as quasi-standard
 90 *   and as such are accomodated by wcsprm::crota[] and translated to
 91 *   wcsprm::pc[][] by wcsset().  These are not dealt with here, nor are any
 92 *   other non-standard keywords since these routines work only on the contents
 93 *   of a wcsprm struct and do not deal with FITS headers per se.  In
 94 *   particular, they do not identify or translate CD00i00j, PC00i00j, PROJPn,
 95 *   EPOCH, VELREF or VSOURCEa keywords; this may be done by the FITS WCS
 96 *   header parser supplied with WCSLIB, refer to wcshdr.h.
 97 *
 98 * wcsfix() and wcsfixi() apply all of the corrections handled by the following
 99 * specific functions, which may also be invoked separately:
100 *
101 *   - cdfix(): Sets the diagonal element of the CDi_ja matrix to 1.0 if all
102 *     CDi_ja keywords associated with a particular axis are omitted.
103 *
104 *   - datfix(): recast an older DATE-OBS date format in dateobs to year-2000
105 *     standard form.  Derive dateref from mjdref if not already set.
106 *     Alternatively, if dateref is set and mjdref isn't, then derive mjdref
107 *     from it.  If both are set, then check consistency.  Likewise for dateobs
108 *     and mjdobs; datebeg and mjdbeg; dateavg and mjdavg; and dateend and
109 *     mjdend.
110 *
111 *   - obsfix(): if only one half of obsgeo[] is set, then derive the other
112 *     half from it.  If both halves are set, then check consistency.
113 *
114 *   - unitfix(): translate some commonly used but non-standard unit strings in
115 *     the CUNITia keyvalues, e.g. 'DEG' -> 'deg'.
116 *
117 *   - spcfix(): translate AIPS-convention spectral types, 'FREQ-LSR',
118 *     'FELO-HEL', etc., in ctype[] as set from CTYPEia.
119 *
120 *   - celfix(): translate AIPS-convention celestial projection types, NCP and
121 *     GLS, in ctype[] as set from CTYPEia.
122 *
123 *   - cylfix(): fixes WCS keyvalues for malformed cylindrical projections that
124 *     suffer from the problem described in Sect. 7.3.4 of Paper I.
125 *
126 *
127 * wcsfix() - Translate a non-standard WCS struct
128 * ----------------------------------------------
129 * wcsfix() is identical to wcsfixi(), but lacks the info argument.
130 *
131 *
132 * wcsfixi() - Translate a non-standard WCS struct
133 * -----------------------------------------------
134 * wcsfixi() applies all of the corrections handled separately by cdfix(),
135 * datfix(), obsfix(), unitfix(), spcfix(), celfix(), and cylfix().
136 *
137 * Given:
138 *   ctrl      int        Do potentially unsafe translations of non-standard
139 *                        unit strings as described in the usage notes to
140 *                        wcsutrn().
141 *
142 *   naxis     const int []
143 *                        Image axis lengths.  If this array pointer is set to
144 *                        zero then cylfix() will not be invoked.
145 *
146 * Given and returned:
147 *   wcs       struct wcsprm*
148 *                        Coordinate transformation parameters.
149 *
150 * Returned:
151 *   stat      int [NWCSFIX]
152 *                        Status returns from each of the functions.  Use the
153 *                        preprocessor macros NWCSFIX to dimension this vector
154 *                        and CDFIX, DATFIX, OBSFIX, UNITFIX, SPCFIX, CELFIX,
155 *                        and CYLFIX to access its elements.  A status value
156 *                        of -2 is set for functions that were not invoked.
157 *
158 *   info      struct wcserr [NWCSFIX]
159 *                        Status messages from each of the functions.  Use the
160 *                        preprocessor macros NWCSFIX to dimension this vector
161 *                        and CDFIX, DATFIX, OBSFIX, UNITFIX, SPCFIX, CELFIX,
162 *                        and CYLFIX to access its elements.
163 *
164 *                        Note that the memory allocated by wcsfixi() for the
165 *                        message in each wcserr struct (wcserr::msg, if
166 *                        non-zero) must be freed by the user.  See
167 *                        wcsdealloc().
168 *
169 * Function return value:
170 *              int        Status return value:
```

```
171 *                              0: Success.
172 *                              1: One or more of the translation functions
173 *                                 returned an error.
174 *
175 *
176 * cdfix() - Fix erroneously omitted CDi_ja keywords
177 * -------------------------------------------------
178 * cdfix() sets the diagonal element of the CDi_ja matrix to unity if all
179 * CDi_ja keywords associated with a given axis were omitted.  According to WCS
180 * Paper I, if any CDi_ja keywords at all are given in a FITS header then those
181 * not given default to zero.  This results in a singular matrix with an
182 * intersecting row and column of zeros.
183 *
184 * cdfix() is expected to be invoked before wcsset(), which will fail if these
185 * errors have not been corrected.
186 *
187 * Given and returned:
188 *    wcs       struct wcsprm*
189 *                      Coordinate transformation parameters.
190 *
191 * Function return value:
192 *              int      Status return value:
193 *                       -1: No change required (not an error).
194 *                        0: Success.
195 *                        1: Null wcsprm pointer passed.
196 *
197 *
198 * datfix() - Translate DATE-OBS and derive MJD-OBS or vice versa
199 * -------------------------------------------------------------
200 * datfix() translates the old DATE-OBS date format set in wcsprm::dateobs to
201 * year-2000 standard form (yyyy-mm-ddThh:mm:ss).  It derives wcsprm::dateref
202 * from wcsprm::mjdref if not already set.  Alternatively, if dateref is set
203 * and mjdref isn't, then it derives mjdref from it.  If both are set but
204 * disagree by more than 0.001 day (86.4 seconds) then an error status is
205 * returned.  Likewise for wcsprm::dateobs and wcsprm::mjdobs; wcsprm::datebeg
206 * and wcsprm::mjdbeg; wcsprm::dateavg and wcsprm::mjdavg; and wcsprm::dateend
207 * and wcsprm::mjdend.
208 *
209 * If neither dateobs nor mjdobs are set, but wcsprm::jepoch (primarily) or
210 * wcsprm::bepoch is, then both are derived from it.  If jepoch and/or bepoch
211 * are set but disagree with dateobs or mjdobs by more than 0.000002 year
212 * (63.2 seconds), an informative message is produced.
213 *
214 * The translations done by datfix() do not affect and are not affected by
215 * wcsset().
216 *
217 * Given and returned:
218 *    wcs       struct wcsprm*
219 *                      Coordinate transformation parameters.
220 *                      wcsprm::dateref and/or wcsprm::mjdref may be changed.
221 *                      wcsprm::dateobs and/or wcsprm::mjdobs may be changed.
222 *                      wcsprm::datebeg and/or wcsprm::mjdbeg may be changed.
223 *                      wcsprm::dateavg and/or wcsprm::mjdavg may be changed.
224 *                      wcsprm::dateend and/or wcsprm::mjdend may be changed.
225 *
226 * Function return value:
227 *              int      Status return value:
228 *                       -1: No change required (not an error).
229 *                        0: Success.
230 *                        1: Null wcsprm pointer passed.
231 *                        5: Invalid parameter value.
232 *
233 *                      For returns >= 0, a detailed message, whether
234 *                      informative or an error message, may be set in
235 *                      wcsprm::err if enabled, see wcserr_enable(), with
236 *                      wcsprm::err.status set to FIXERR_DATE_FIX.
237 *
238 * Notes:
239 *    1: The MJD algorithms used by datfix() are from D.A. Hatcher, 1984, QJRAS,
240 *       25, 53-55, as modified by P.T. Wallace for use in SLALIB subroutines
241 *       CLDJ and DJCL.
242 *
243 *
244 * obsfix() - complete the OBSGEO-[XYZLBH] vector of observatory coordinates
245 * ------------------------------------------------------------------------
246 * obsfix() completes the wcsprm::obsgeo vector of observatory coordinates.
247 * That is, if only the (x,y,z) Cartesian coordinate triplet or the (l,b,h)
248 * geodetic coordinate triplet are set, then it derives the other triplet from
249 * it.  If both triplets are set, then it checks for consistency at the level
250 * of 1 metre.
251 *
252 * The operations done by obsfix() do not affect and are not affected by
253 * wcsset().
254 *
255 * Given:
256 *    ctrl      int      Flag that controls behaviour if one triplet is
257 *                       defined and the other is only partially defined:
```

```
258 *                          0: Reset only the undefined elements of an
259 *                             incomplete coordinate triplet.
260 *                          1: Reset all elements of an incomplete triplet.
261 *                          2: Don't make any changes, check for consistency
262 *                             only.  Returns an error if either of the two
263 *                             triplets is incomplete.
264 *
265 * Given and returned:
266 *   wcs       struct wcsprm*
267 *                       Coordinate transformation parameters.
268 *                       wcsprm::obsgeo may be changed.
269 *
270 * Function return value:
271 *             int       Status return value:
272 *                          -1: No change required (not an error).
273 *                           0: Success.
274 *                           1: Null wcsprm pointer passed.
275 *                           5: Invalid parameter value.
276 *
277 *                       For returns >= 0, a detailed message, whether
278 *                       informative or an error message, may be set in
279 *                       wcsprm::err if enabled, see wcserr_enable(), with
280 *                       wcsprm::err.status set to FIXERR_OBS_FIX.
281 *
282 * Notes:
283 *   1: While the International Terrestrial Reference System (ITRS) is based
284 *      solely on Cartesian coordinates, it recommends the use of the GRS80
285 *      ellipsoid in converting to geodetic coordinates.  However, while WCS
286 *      Paper III recommends ITRS Cartesian coordinates, Paper VII prescribes
287 *      the use of the IAU(1976) ellipsoid for geodetic coordinates, and
288 *      consequently that is what is used here.
289 *
290 *   2: For reference, parameters of commonly used global reference ellipsoids:
291 *
292 =           a (m)          1/f               Standard
293 =         ---------    -------------    -------------------------------
294 =         6378140    298.2577         IAU(1976)
295 =         6378137    298.257222101    GRS80
296 =         6378137    298.257223563    WGS84
297 =         6378136    298.257          IERS(1989)
298 =         6378136.6  298.25642        IERS(2003,2010), IAU(2009/2012)
299 *
300 *      where f = (a - b) / a is the flattening, and a and b are the semi-major
301 *      and semi-minor radii in metres.
302 *
303 *   3: The transformation from geodetic (lng,lat,hgt) to Cartesian (x,y,z) is
304 *
305 =         x = (n + hgt)*coslng*coslat,
306 =         y = (n + hgt)*sinlng*coslat,
307 =         z = (n*(1.0 - e^2) + hgt)*sinlat,
308 *
309 *      where the "prime vertical radius", n, is a function of latitude
310 *
311 =         n = a / sqrt(1 - (e*sinlat)^2),
312 *
313 *      and a, the equatorial radius, and e^2 = (2 - f)*f, the (first)
314 *      eccentricity of the ellipsoid, are constants.  obsfix() inverts these
315 *      iteratively by writing
316 *
317 =           x = rho*coslng*coslat,
318 =           y = rho*sinlng*coslat,
319 =        zeta = rho*sinlat,
320 *
321 *      where
322 *
323 =         rho = n + hgt,
324 =             = sqrt(x^2 + y^2 + zeta^2),
325 =        zeta = z / (1 - n*e^2/rho),
326 *
327 *      and iterating over the value of zeta.  Since e is small, a good first
328 *      approximation is given by zeta = z.
329 *
330 *
331 * unitfix() - Correct aberrant CUNITia keyvalues
332 * ----------------------------------------------
333 * unitfix() applies wcsutrn() to translate non-standard CUNITia keyvalues,
334 * e.g. 'DEG' -> 'deg', also stripping off unnecessary whitespace.
335 *
336 * unitfix() is expected to be invoked before wcsset(), which will fail if
337 * non-standard CUNITia keyvalues have not been translated.
338 *
339 * Given:
340 *   ctrl      int       Do potentially unsafe translations described in the
341 *                       usage notes to wcsutrn().
342 *
343 * Given and returned:
344 *   wcs       struct wcsprm*
```

```
345 *                          Coordinate transformation parameters.
346 *
347 * Function return value:
348 *              int       Status return value:
349 *                         -1: No change required (not an error).
350 *                          0: Success (an alias was applied).
351 *                          1: Null wcsprm pointer passed.
352 *
353 *                        When units are translated (i.e. 0 is returned), an
354 *                        informative message is set in wcsprm::err if enabled,
355 *                        see wcserr_enable(), with wcsprm::err.status set to
356 *                        FIXERR_UNITS_ALIAS.
357 *
358 *
359 * spcfix() - Translate AIPS-convention spectral types
360 * ---------------------------------------------------
361 * spcfix() translates AIPS-convention spectral coordinate types,
362 * '{FREQ,FELO,VELO}-{LSR,HEL,OBS}' (e.g. 'FREQ-OBS', 'FELO-HEL', 'VELO-LSR')
363 * set in wcsprm::ctype[], subject to VELREF set in wcsprm::velref.
364 *
365 * Note that if wcs::specsys is already set then it will not be overridden.
366 *
367 * AIPS-convention spectral types set in CTYPEia are translated on-the-fly by
368 * wcsset() but without modifying wcsprm::ctype[] or wcsprm::specsys.  That is,
369 * only the information extracted from wcsprm::ctype[] is translated when
370 * wcsset() fills in wcsprm::spc (spcprm struct).  spcfix() modifies
371 * wcsprm::ctype[] so that if the header is subsequently written out, e.g. by
372 * wcshdo(), then it will contain translated CTYPEia keyvalues.
373 *
374 * The operations done by spcfix() do not affect and are not affected by
375 * wcsset().
376 *
377 * Given and returned:
378 *    wcs       struct wcsprm*
379 *                        Coordinate transformation parameters.  wcsprm::ctype[]
380 *                        and/or wcsprm::specsys may be changed.
381 *
382 * Function return value:
383 *              int       Status return value:
384 *                         -1: No change required (not an error).
385 *                          0: Success.
386 *                          1: Null wcsprm pointer passed.
387 *                          2: Memory allocation failed.
388 *                          3: Linear transformation matrix is singular.
389 *                          4: Inconsistent or unrecognized coordinate axis
390 *                             types.
391 *                          5: Invalid parameter value.
392 *                          6: Invalid coordinate transformation parameters.
393 *                          7: Ill-conditioned coordinate transformation
394 *                             parameters.
395 *
396 *                        For returns >= 0, a detailed message, whether
397 *                        informative or an error message, may be set in
398 *                        wcsprm::err if enabled, see wcserr_enable(), with
399 *                        wcsprm::err.status set to FIXERR_SPC_UPDTE.
400 *
401 *
402 * celfix() - Translate AIPS-convention celestial projection types
403 * --------------------------------------------------------------
404 * celfix() translates AIPS-convention celestial projection types, NCP and
405 * GLS, set in the ctype[] member of the wcsprm struct.
406 *
407 * Two additional pv[] keyvalues are created when translating NCP, and three
408 * are created when translating GLS with non-zero reference point.  If the pv[]
409 * array was initially allocated by wcsini() then the array will be expanded if
410 * necessary.  Otherwise, error 2 will be returned if sufficient empty slots
411 * are not already available for use.
412 *
413 * AIPS-convention celestial projection types set in CTYPEia are translated
414 * on-the-fly by wcsset() but without modifying wcsprm::ctype[], wcsprm::pv[],
415 * or wcsprm::npv.  That is, only the information extracted from
416 * wcsprm::ctype[] is translated when wcsset() fills in wcsprm::cel (celprm
417 * struct).  celfix() modifies wcsprm::ctype[], wcsprm::pv[], and wcsprm::npv
418 * so that if the header is subsequently written out, e.g. by wcshdo(), then it
419 * will contain translated CTYPEia keyvalues and the relevant PVi_ma.
420 *
421 * The operations done by celfix() do not affect and are not affected by
422 * wcsset().  However, it uses information in the wcsprm struct provided by
423 * wcsset(), and will invoke it if necessary.
424 *
425 * Given and returned:
426 *    wcs       struct wcsprm*
427 *                        Coordinate transformation parameters.  wcsprm::ctype[]
428 *                        and/or wcsprm::pv[] may be changed.
429 *
430 * Function return value:
431 *              int       Status return value:
```

```
432 *                             -1: No change required (not an error).
433 *                              0: Success.
434 *                              1: Null wcsprm pointer passed.
435 *                              2: Memory allocation failed.
436 *                              3: Linear transformation matrix is singular.
437 *                              4: Inconsistent or unrecognized coordinate axis
438 *                                 types.
439 *                              5: Invalid parameter value.
440 *                              6: Invalid coordinate transformation parameters.
441 *                              7: Ill-conditioned coordinate transformation
442 *                                 parameters.
443 *
444 *                        For returns > 1, a detailed error message is set in
445 *                        wcsprm::err if enabled, see wcserr_enable().
446 *
447 *
448 * cylfix() - Fix malformed cylindrical projections
449 * ------------------------------------------------
450 * cylfix() fixes WCS keyvalues for malformed cylindrical projections that
451 * suffer from the problem described in Sect. 7.3.4 of Paper I.
452 *
453 * cylfix() requires the wcsprm struct to have been set up by wcsset(), and
454 * will invoke it if necessary.  After modification, the struct is reset on
455 * return with an explicit call to wcsset().
456 *
457 * Given:
458 *   naxis     const int []
459 *                        Image axis lengths.
460 *
461 * Given and returned:
462 *   wcs       struct wcsprm*
463 *                        Coordinate transformation parameters.
464 *
465 * Function return value:
466 *             int        Status return value:
467 *                         -1: No change required (not an error).
468 *                          0: Success.
469 *                          1: Null wcsprm pointer passed.
470 *                          2: Memory allocation failed.
471 *                          3: Linear transformation matrix is singular.
472 *                          4: Inconsistent or unrecognized coordinate axis
473 *                             types.
474 *                          5: Invalid parameter value.
475 *                          6: Invalid coordinate transformation parameters.
476 *                          7: Ill-conditioned coordinate transformation
477 *                             parameters.
478 *                          8: All of the corner pixel coordinates are invalid.
479 *                          9: Could not determine reference pixel coordinate.
480 *                         10: Could not determine reference pixel value.
481 *
482 *                        For returns > 1, a detailed error message is set in
483 *                        wcsprm::err if enabled, see wcserr_enable().
484 *
485 *
486 * wcspcx() - regularize PCi_j
487 * --------------------------
488 * wcspcx() "regularizes" the linear transformation matrix component of the
489 * coordinate transformation (PCi_ja) to make it more human-readable.
490 *
491 * Normally, upon encountering a FITS header containing a CDi_ja matrix,
492 * wcsset() simply treats it as PCi_ja and sets CDELTia to unity.  However,
493 * wcspcx() decomposes CDi_ja into PCi_ja and CDELTia in such a way that
494 * CDELTia form meaningful scaling parameters.  In practice, the residual
495 * PCi_ja matrix will often then be orthogonal, i.e. unity, or describing a
496 * pure rotation, axis permutation, or reflection, or a combination thereof.
497 *
498 * The decomposition is based on normalizing the length in the transformed
499 * system (i.e. intermediate pixel coordinates) of the orthonormal basis
500 * vectors of the pixel coordinate system.  This deviates slightly from the
501 * prescription given by Eq. (4) of WCS Paper I, namely Sum(j=1,N)(PCi_ja)² = 1,
502 * in replacing the sum over j with the sum over i.  Consequently, the columns
503 * of PCi_ja will consist of unit vectors.  In practice, especially in cubes
504 * and higher dimensional images, at least some pairs of these unit vectors, if
505 * not all, will often be orthogonal or close to orthogonal.
506 *
507 * The sign of CDELTia is chosen to make the PCi_ja matrix as close to the,
508 * possibly permuted, unit matrix as possible, except that where the coordinate
509 * description contains a pair of celestial axes, the sign of CDELTia is set
510 * negative for the longitude axis and positive for the latitude axis.
511 *
512 * Optionally, rows of the PCi_ja matrix may also be permuted to diagonalize
513 * it as far as possible, thus undoing any transposition of axes in the
514 * intermediate pixel coordinate system.
515 *
516 * If the coordinate description contains a celestial plane, then the angle of
517 * rotation of each of the basis vectors associated with the celestial axes is
518 * returned.  For a pure rotation the two angles should be identical.  Any
```

```
519 * difference between them is a measure of axis skewness.
520 *
521 * The decomposition is not performed for axes involving a sequent distortion
522 * function that is defined in terms of CDi_ja, such as TPV, TNX, or ZPX, which
523 * always are.  The independent variables of the polynomial are therefore
524 * intermediate world coordinates rather than intermediate pixel coordinates.
525 * Because sequent distortions are always applied before CDELTia, if CDi_ja was
526 * translated to PCi_ja plus CDELTia, then the distortion would be altered
527 * unless the polynomial coefficients were also adjusted to account for the
528 * change of scale.
529 *
530 * wcspcx() requires the wcsprm struct to have been set up by wcsset(), and
531 * will invoke it if necessary.  The wcsprm struct is reset on return with an
532 * explicit call to wcsset().
533 *
534 * Given and returned:
535 *    wcs       struct wcsprm*
536 *                      Coordinate transformation parameters.
537 *
538 * Given:
539 *    dopc      int       If 1, then PCi_ja and CDELTia, as given, will be
540 *                        recomposed according to the above prescription.  If 0,
541 *                        the operation is restricted to decomposing CDi_ja.
542 *
543 *    permute   int       If 1, then after decomposition (or recomposition),
544 *                        permute rows of PCi_ja to make the axes of the
545 *                        intermediate pixel coordinate system match as closely
546 *                        as possible those of the pixel coordinates.  That is,
547 *                        make it as close to a diagonal matrix as possible.
548 *                        However, celestial axes are special in always being
549 *                        paired, with the longitude axis preceding the latitude
550 *                        axis.
551 *
552 *                        All WCS entities indexed by i, such as CTYPEia,
553 *                        CRVALia, CDELTia, etc., including coordinate lookup
554 *                        tables, will also be permuted as necessary to account
555 *                        for the change to PCi_ja.  This does not apply to
556 *                        CRPIXja, nor prior distortion functions.  These
557 *                        operate on pixel coordinates, which are not affected
558 *                        by the permutation.
559 *
560 * Returned:
561 *    rotn      double[2] Rotation angle [deg] of each basis vector associated
562 *                        with the celestial axes.  For a pure rotation the two
563 *                        angles should be identical.  Any difference between
564 *                        them is a measure of axis skewness.
565 *
566 *                        May be set to the NULL pointer if this information is
567 *                        not required.
568 *
569 * Function return value:
570 *              int       Status return value:
571 *                          0: Success.
572 *                          1: Null wcsprm pointer passed.
573 *                          2: Memory allocation failed.
574 *                          5: CDi_j matrix not used.
575 *                          6: Sequent distortion function present.
576 *
577 *
578 * Global variable: const char *wcsfix_errmsg[] - Status return messages
579 * -----------------------------------------------------------------
580 * Error messages to match the status value returned from each function.
581 *
582 *=========================================================================*/
583
584 #ifndef WCSLIB_WCSFIX
585 #define WCSLIB_WCSFIX
586
587 #include "wcs.h"
588 #include "wcserr.h"
589
590 #ifdef __cplusplus
591 extern "C" {
592 #endif
593
594 #define CDFIX    0
595 #define DATFIX   1
596 #define OBSFIX   2
597 #define UNITFIX  3
598 #define SPCFIX   4
599 #define CELFIX   5
600 #define CYLFIX   6
601 #define NWCSFIX  7
602
603 extern const char *wcsfix_errmsg[];
604 #define cylfix_errmsg wcsfix_errmsg
605
```

```
606 enum wcsfix_errmsg_enum {
607   FIXERR_OBSGEO_FIX       = -5, // Observatory coordinates amended.
608   FIXERR_DATE_FIX         = -4, // Date string reformatted.
609   FIXERR_SPC_UPDATE       = -3, // Spectral axis type modified.
610   FIXERR_UNITS_ALIAS      = -2,        // Units alias translation.
611   FIXERR_NO_CHANGE        = -1, // No change.
612   FIXERR_SUCCESS          =  0,    // Success.
613   FIXERR_NULL_POINTER     =  1,      // Null wcsprm pointer passed.
614   FIXERR_MEMORY           =  2,    // Memory allocation failed.
615   FIXERR_SINGULAR_MTX     =  3,      // Linear transformation matrix is singular.
616   FIXERR_BAD_CTYPE        =  4, // Inconsistent or unrecognized coordinate
617                                 // axis types.
618   FIXERR_BAD_PARAM        =  5, // Invalid parameter value.
619   FIXERR_BAD_COORD_TRANS  =  6,   // Invalid coordinate transformation
620                                 // parameters.
621   FIXERR_ILL_COORD_TRANS  =  7,   // Ill-conditioned coordinate transformation
622                                 // parameters.
623   FIXERR_BAD_CORNER_PIX   =  8,    // All of the corner pixel coordinates are
624                                 // invalid.
625   FIXERR_NO_REF_PIX_COORD =  9,  // Could not determine reference pixel
626                                 // coordinate.
627   FIXERR_NO_REF_PIX_VAL   = 10     // Could not determine reference pixel value.
628 };
629
630 int wcsfix(int ctrl, const int naxis[], struct wcsprm *wcs, int stat[]);
631
632 int wcsfixi(int ctrl, const int naxis[], struct wcsprm *wcs, int stat[],
633            struct wcserr info[]);
634
635 int cdfix(struct wcsprm *wcs);
636
637 int datfix(struct wcsprm *wcs);
638
639 int obsfix(int ctrl, struct wcsprm *wcs);
640
641 int unitfix(int ctrl, struct wcsprm *wcs);
642
643 int spcfix(struct wcsprm *wcs);
644
645 int celfix(struct wcsprm *wcs);
646
647 int cylfix(const int naxis[], struct wcsprm *wcs);
648
649 int wcspcx(struct wcsprm *wcs, int dopc, int permute, double rotn[2]);
650
651
652 #ifdef __cplusplus
653 }
654 #endif
655
656 #endif // WCSLIB_WCSFIX
```

19.29   wcshdr.h File Reference

```
#include "wcs.h"
```

Macros

- #define WCSHDR_none 0x00000000

  Bit mask for wcspih() and wcsbth() - reject all extensions.
- #define WCSHDR_all 0x000FFFFF

  Bit mask for wcspih() and wcsbth() - accept all extensions.
- #define WCSHDR_reject 0x10000000

  Bit mask for wcspih() and wcsbth() - reject non-standard keywords.
- #define WCSHDR_strict 0x20000000
- #define WCSHDR_CROTAia 0x00000001

  *Bit mask for wcspih() and wcsbth() - accept **CROTA**ia, i**CROT**na, **TCROT**na.*
- #define WCSHDR_VELREFa 0x00000002

  *Bit mask for wcspih() and wcsbth() - accept **VELREF**a.*

- #define [WCSHDR_CD00i00j](#) 0x00000004

  Bit mask for [wcspih()](#) and [wcsbth()](#) - accept $CD00i00j$.

- #define [WCSHDR_PC00i00j](#) 0x00000008

  Bit mask for [wcspih()](#) and [wcsbth()](#) - accept $PC00i00j$.

- #define [WCSHDR_PROJPn](#) 0x00000010

  Bit mask for [wcspih()](#) and [wcsbth()](#) - accept $PROJPn$.

- #define [WCSHDR_CD0i_0ja](#) 0x00000020
- #define [WCSHDR_PC0i_0ja](#) 0x00000040
- #define [WCSHDR_PV0i_0ma](#) 0x00000080
- #define [WCSHDR_PS0i_0ma](#) 0x00000100
- #define [WCSHDR_DOBSn](#) 0x00000200

  Bit mask for [wcspih()](#) and [wcsbth()](#) - accept $DOBSn$.

- #define [WCSHDR_OBSGLBHn](#) 0x00000400
- #define [WCSHDR_RADECSYS](#) 0x00000800

  Bit mask for [wcspih()](#) and [wcsbth()](#) - accept $RADECSYS$.

- #define [WCSHDR_EPOCHa](#) 0x00001000

  Bit mask for [wcspih()](#) and [wcsbth()](#) - accept $EPOCHa$.

- #define [WCSHDR_VSOURCE](#) 0x00002000

  Bit mask for [wcspih()](#) and [wcsbth()](#) - accept $VSOURCEa$.

- #define [WCSHDR_DATEREF](#) 0x00004000
- #define [WCSHDR_LONGKEY](#) 0x00008000

  Bit mask for [wcspih()](#) and [wcsbth()](#) - accept long forms of the alternate binary table and pixel list WCS keywords.

- #define [WCSHDR_CNAMn](#) 0x00010000

  Bit mask for [wcspih()](#) and [wcsbth()](#) - accept $iCNAMn$, $TCNAMn$, $iCRDEn$, $TCRDEn$, $iCSYEn$, $TCSYEn$.

- #define [WCSHDR_AUXIMG](#) 0x00020000

  Bit mask for [wcspih()](#) and [wcsbth()](#) - allow the image-header form of an auxiliary WCS keyword to provide a default value for all images.

- #define [WCSHDR_ALLIMG](#) 0x00040000

  Bit mask for [wcspih()](#) and [wcsbth()](#) - allow the image-header form of all image header WCS keywords to provide a default value for all images.

- #define [WCSHDR_IMGHEAD](#) 0x00100000

  Bit mask for [wcsbth()](#) - restrict to image header keywords only.

- #define [WCSHDR_BIMGARR](#) 0x00200000

  Bit mask for [wcsbth()](#) - restrict to binary table image array keywords only.

- #define [WCSHDR_PIXLIST](#) 0x00400000

  Bit mask for [wcsbth()](#) - restrict to pixel list keywords only.

- #define [WCSHDO_none](#) 0x00000

  Bit mask for [wcshdo()](#) - don't write any extensions.

- #define [WCSHDO_all](#) 0x000FF

  Bit mask for [wcshdo()](#) - write all extensions.

- #define [WCSHDO_safe](#) 0x0000F

  Bit mask for [wcshdo()](#) - write safe extensions only.

- #define [WCSHDO_DOBSn](#) 0x00001

  Bit mask for [wcshdo()](#) - write $DOBSn$.

- #define [WCSHDO_TPCn_ka](#) 0x00002

  Bit mask for [wcshdo()](#) - write $TPCn\_ka$.

- #define [WCSHDO_PVn_ma](#) 0x00004

  Bit mask for [wcshdo()](#) - write $iPVn\_ma$, $TPVn\_ma$, $iPSn\_ma$, $TPSn\_ma$.

- #define [WCSHDO_CRPXna](#) 0x00008

  Bit mask for [wcshdo()](#) - write $jCRPXna$, $TCRPXna$, $iCDLTna$, $TCDLTna$, $iCUNIna$, $TCUNIna$, $iCTYPna$, $TCTYPna$, $iCRVLna$, $TCRVLna$.

- #define [WCSHDO_CNAMna](#) 0x00010

*Bit mask for [wcshdo()](#) - write i***CNAM***na, ***TCNAM***na, i***CRDE***na, ***TCRDE***na, i***CSYE***na, ***TCSYE***na.*

- #define [WCSHDO_WCSNna](#) 0x00020

    *Bit mask for [wcshdo()](#) - write ***WCSN***na instead of ***TWCS***na*

- #define [WCSHDO_P12](#) 0x01000
- #define [WCSHDO_P13](#) 0x02000
- #define [WCSHDO_P14](#) 0x04000
- #define [WCSHDO_P15](#) 0x08000
- #define [WCSHDO_P16](#) 0x10000
- #define [WCSHDO_P17](#) 0x20000
- #define [WCSHDO_EFMT](#) 0x40000

Enumerations

- enum [wcshdr_errmsg_enum](#) {
  [WCSHDRERR_SUCCESS](#) = 0 , [WCSHDRERR_NULL_POINTER](#) = 1 , [WCSHDRERR_MEMORY](#) = 2 ,
  [WCSHDRERR_BAD_COLUMN](#) = 3 ,
  [WCSHDRERR_PARSER](#) = 4 , [WCSHDRERR_BAD_TABULAR_PARAMS](#) = 5 }

Functions

- int [wcspih](#) (char ∗header, int nkeyrec, int relax, int ctrl, int ∗nreject, int ∗nwcs, struct [wcsprm](#) ∗∗wcs)

    FITS WCS parser routine for image headers.

- int [wcsbth](#) (char ∗header, int nkeyrec, int relax, int ctrl, int keysel, int ∗colsel, int ∗nreject, int ∗nwcs, struct [wcsprm](#) ∗∗wcs)

    FITS WCS parser routine for binary table and image headers.

- int [wcstab](#) (struct [wcsprm](#) ∗wcs)

    Tabular construction routine.

- int [wcsidx](#) (int nwcs, struct [wcsprm](#) ∗∗wcs, int alts[27])

    Index alternate coordinate representations.

- int [wcsbdx](#) (int nwcs, struct [wcsprm](#) ∗∗wcs, int type, short alts[1000][28])

    Index alternate coordinate representions.

- int [wcsvfree](#) (int ∗nwcs, struct [wcsprm](#) ∗∗wcs)

    Free the array of [wcsprm](#) structs.

- int [wcshdo](#) (int ctrl, struct [wcsprm](#) ∗wcs, int ∗nkeyrec, char ∗∗header)

    Write out a [wcsprm](#) struct as a FITS header.

Variables

- const char ∗ [wcshdr_errmsg](#) [ ]

    Status return messages.

19.29.1   Detailed Description

Routines in this suite are aimed at extracting WCS information from a FITS file. The information is encoded via keywords defined in

```
"Representations of world coordinates in FITS",
Greisen, E.W., & Calabretta, M.R. 2002, A&A, 395, 1061 (WCS Paper I)
"Representations of celestial coordinates in FITS",
Calabretta, M.R., & Greisen, E.W. 2002, A&A, 395, 1077 (WCS Paper II)
"Representations of spectral coordinates in FITS",
Greisen, E.W., Calabretta, M.R., Valdes, F.G., & Allen, S.L.
2006, A&A, 446, 747 (WCS Paper III)
"Representations of distortions in FITS world coordinate systems",
Calabretta, M.R. et al. (WCS Paper IV, draft dated 2004/04/22),
available from http://www.atnf.csiro.au/people/Mark.Calabretta
"Representations of time coordinates in FITS -
 Time and relative dimension in space",
Rots, A.H., Bunclark, P.S., Calabretta, M.R., Allen, S.L.,
Manchester, R.N., & Thompson, W.T. 2015, A&A, 574, A36 (WCS Paper VII)
```

These routines provide the high-level interface between the FITS file and the WCS coordinate transformation routines.

Additionally, function wcshdo() is provided to write out the contents of a wcsprm struct as a FITS header.

Briefly, the anticipated sequence of operations is as follows:

- 1: Open the FITS file and read the image or binary table header, e.g. using CFITSIO routine *fits_hdr2str()*.

- 2: Parse the header using wcspih() or wcsbth(); they will automatically interpret **'TAB'** header keywords using wcstab().

- 3: Allocate memory for, and read **'TAB'** arrays from the binary table extension, e.g. using CFITSIO routine fits_read_wcstab() - refer to the prologue of getwcstab.h. wcsset() will automatically take control of this allocated memory, in particular causing it to be freed by wcsfree().

- 4: Translate non-standard WCS usage using wcsfix(), see wcsfix.h.

- 5: Initialize wcsprm struct(s) using wcsset() and calculate coordinates using wcsp2s() and/or wcss2p(). Refer to the prologue of wcs.h for a description of these and other high-level WCS coordinate transformation routines.

- 6: Clean up by freeing memory with wcsvfree().

In detail:

- wcspih() is a high-level FITS WCS routine that parses an image header. It returns an array of up to 27 wcsprm structs on each of which it invokes wcstab().

- wcsbth() is the analogue of wcspih() for use with binary tables; it handles image array and pixel list keywords. As an extension of the FITS WCS standard, it also recognizes image header keywords which may be used to provide default values via an inheritance mechanism.

- wcstab() assists in filling in members of the wcsprm struct associated with coordinate lookup tables (**'TAB'**). These are based on arrays stored in a FITS binary table extension (BINTABLE) that are located by **PV**i_ma keywords in the image header.

- wcsidx() and wcsbdx() are utility routines that return the index for a specified alternate coordinate descriptor in the array of wcsprm structs returned by wcspih() or wcsbth().

- wcsvfree() deallocates memory for an array of wcsprm structs, such as returned by wcspih() or wcsbth().

- wcshdo() writes out a wcsprm struct as a FITS header.

19.29.2 Macro Definition Documentation

19.29.2.1 WCSHDR_none `#define WCSHDR_none 0x00000000`

Bit mask for the *relax* argument of wcspih() and wcsbth() - reject all extensions.

Refer to wcsbth() note 5.

19.29.2.2 WCSHDR_all `#define WCSHDR_all 0x000FFFFF`

Bit mask for the *relax* argument of wcspih() and wcsbth() - accept all extensions.

Refer to wcsbth() note 5.

19.29.2.3 WCSHDR_reject `#define WCSHDR_reject 0x10000000`

Bit mask for the *relax* argument of wcspih() and wcsbth() - reject non-standard keywords.

Refer to wcsbth() note 5.

19.29.2.4 WCSHDR_strict `#define WCSHDR_strict 0x20000000`

19.29.2.5 WCSHDR_CROTAia `#define WCSHDR_CROTAia 0x00000001`

Bit mask for the *relax* argument of wcspih() and wcsbth() - accept **CROTA**ia, i**CROT**na, **TCROT**na.

Refer to wcsbth() note 5.

19.29.2.6 WCSHDR_VELREFa `#define WCSHDR_VELREFa 0x00000002`

Bit mask for the *relax* argument of wcspih() and wcsbth() - accept **VELREF**a.

Refer to wcsbth() note 5.

19.29.2.7 WCSHDR_CD00i00j `#define WCSHDR_CD00i00j 0x00000004`

Bit mask for the *relax* argument of wcspih() and wcsbth() - accept **CD00**i**00**j.

Refer to wcsbth() note 5.

19.29.2.8 WCSHDR_PC00i00j `#define WCSHDR_PC00i00j 0x00000008`

Bit mask for the *relax* argument of wcspih() and wcsbth() - accept **PC00**i**00**j.

Refer to wcsbth() note 5.

19.29.2.9   WCSHDR_PROJPn   #define WCSHDR_PROJPn 0x00000010

Bit mask for the *relax* argument of wcspih() and wcsbth() - accept **PROJP**n.

Refer to wcsbth() note 5.

19.29.2.10   WCSHDR_CD0i_0ja   #define WCSHDR_CD0i_0ja 0x00000020

19.29.2.11   WCSHDR_PC0i_0ja   #define WCSHDR_PC0i_0ja 0x00000040

19.29.2.12   WCSHDR_PV0i_0ma   #define WCSHDR_PV0i_0ma 0x00000080

19.29.2.13   WCSHDR_PS0i_0ma   #define WCSHDR_PS0i_0ma 0x00000100

19.29.2.14   WCSHDR_DOBSn   #define WCSHDR_DOBSn 0x00000200

Bit mask for the *relax* argument of wcspih() and wcsbth() - accept **DOBS**n.

Refer to wcsbth() note 5.

19.29.2.15   WCSHDR_OBSGLBHn   #define WCSHDR_OBSGLBHn 0x00000400

19.29.2.16   WCSHDR_RADECSYS   #define WCSHDR_RADECSYS 0x00000800

Bit mask for the *relax* argument of wcspih() and wcsbth() - accept **RADECSYS**.

Refer to wcsbth() note 5.

19.29.2.17   WCSHDR_EPOCHa   #define WCSHDR_EPOCHa 0x00001000

Bit mask for the *relax* argument of wcspih() and wcsbth() - accept **EPOCH**a.

Refer to wcsbth() note 5.

19.29.2.18   WCSHDR_VSOURCE   #define WCSHDR_VSOURCE 0x00002000

Bit mask for the *relax* argument of wcspih() and wcsbth() - accept **VSOURCE**a.

Refer to wcsbth() note 5.

19.29.2.19 WCSHDR_DATEREF #define WCSHDR_DATEREF 0x00004000

19.29.2.20 WCSHDR_LONGKEY #define WCSHDR_LONGKEY 0x00008000

Bit mask for the *relax* argument of wcspih() and wcsbth() - accept long forms of the alternate binary table and pixel list WCS keywords.

Refer to wcsbth() note 5.

19.29.2.21 WCSHDR_CNAMn #define WCSHDR_CNAMn 0x00010000

Bit mask for the *relax* argument of wcspih() and wcsbth() - accept i**CNAM**n, **TCNAM**n, i**CRDE**n, **TCRDE**n, i**CSYE**n, **TCSYE**n.

Refer to wcsbth() note 5.

19.29.2.22 WCSHDR_AUXIMG #define WCSHDR_AUXIMG 0x00020000

Bit mask for the *relax* argument of wcspih() and wcsbth() - allow the image-header form of an auxiliary WCS keyword with representation-wide scope to provide a default value for all images.

Refer to wcsbth() note 5.

19.29.2.23 WCSHDR_ALLIMG #define WCSHDR_ALLIMG 0x00040000

Bit mask for the *relax* argument of wcspih() and wcsbth() - allow the image-header form of *all* image header WCS keywords to provide a default value for all image arrays in a binary table (n.b. not pixel list).

Refer to wcsbth() note 5.

19.29.2.24 WCSHDR_IMGHEAD #define WCSHDR_IMGHEAD 0x00100000

Bit mask for the *keysel* argument of wcsbth() - restrict keyword types considered to image header keywords only.

19.29.2.25 WCSHDR_BIMGARR #define WCSHDR_BIMGARR 0x00200000

Bit mask for the *keysel* argument of wcsbth() - restrict keyword types considered to binary table image array keywords only.

19.29.2.26 WCSHDR_PIXLIST #define WCSHDR_PIXLIST 0x00400000

Bit mask for the *keysel* argument of wcsbth() - restrict keyword types considered to pixel list keywords only.

19.29.2.27 WCSHDO_none #define WCSHDO_none 0x00000

Bit mask for the *relax* argument of wcshdo() - don't write any extensions.

Refer to the notes for wcshdo().

19.29.2.28 WCSHDO_all `#define WCSHDO_all 0x000FF`

Bit mask for the *relax* argument of wcshdo() - write all extensions.

Refer to the notes for wcshdo().

19.29.2.29 WCSHDO_safe `#define WCSHDO_safe 0x0000F`

Bit mask for the *relax* argument of wcshdo() - write only extensions that are considered safe.

Refer to the notes for wcshdo().

19.29.2.30 WCSHDO_DOBSn `#define WCSHDO_DOBSn 0x00001`

Bit mask for the *relax* argument of wcshdo() - write **DOBS**n, the column-specific analogue of DATE-OBS for use in binary tables and pixel lists.

Refer to the notes for wcshdo().

19.29.2.31 WCSHDO_TPCn_ka `#define WCSHDO_TPCn_ka 0x00002`

Bit mask for the *relax* argument of wcshdo() - write **TPC**n_ka if less than eight characters instead of **TP**n_ka.

Refer to the notes for wcshdo().

19.29.2.32 WCSHDO_PVn_ma `#define WCSHDO_PVn_ma 0x00004`

Bit mask for the *relax* argument of wcshdo() - write i**PV**n_ma, **TPV**n_ma, i**PS**n_ma, **TPS**n_ma, if less than eight characters instead of i**V**n_ma, **TV**n_ma, i**S**n_ma, **TS**n_ma.

Refer to the notes for wcshdo().

19.29.2.33 WCSHDO_CRPXna `#define WCSHDO_CRPXna 0x00008`

Bit mask for the *relax* argument of wcshdo() - write j**CRPX**na, **TCRPX**na, i**CDLT**na, **TCDLT**na, i**CUNI**na, **TCUNI**na, i**CTYP**na, **TCTYP**na, i**CRVL**na, **TCRVL**na, if less than eight characters instead of j**CRP**na, **TCRP**na, i**CDE**na, **TCDE**na, i**CUN**na, **TCUN**na, i**CTY**na, **TCTY**na, i**CRV**na, **TCRV**na.

Refer to the notes for wcshdo().

19.29.2.34 WCSHDO_CNAMna `#define WCSHDO_CNAMna 0x00010`

Bit mask for the *relax* argument of wcshdo() - write i**CNAM**na, **TCNAM**na, i**CRDE**na, **TCRDE**na, i**CSYE**na, **TCSYE**na, if less than eight characters instead of i**CNA**na, **TCNA**na, i**CRD**na, **TCRD**na, i**CSY**na, **TCSY**na.

Refer to the notes for wcshdo().

19.29.2.35 WCSHDO_WCSNna `#define WCSHDO_WCSNna 0x00020`

Bit mask for the *relax* argument of wcshdo() - write **WCSN**na instead of **TWCS**na.

Refer to the notes for wcshdo().

19.29.2.36 WCSHDO_P12 `#define WCSHDO_P12 0x01000`

19.29.2.37 WCSHDO_P13 `#define WCSHDO_P13 0x02000`

19.29.2.38 WCSHDO_P14 `#define WCSHDO_P14 0x04000`

19.29.2.39 WCSHDO_P15 `#define WCSHDO_P15 0x08000`

19.29.2.40 WCSHDO_P16 `#define WCSHDO_P16 0x10000`

19.29.2.41 WCSHDO_P17 `#define WCSHDO_P17 0x20000`

19.29.2.42 WCSHDO_EFMT `#define WCSHDO_EFMT 0x40000`

19.29.3 Enumeration Type Documentation

19.29.3.1 wcshdr_errmsg_enum enum wcshdr_errmsg_enum

Enumerator

WCSHDRERR_SUCCESS	
WCSHDRERR_NULL_POINTER	
WCSHDRERR_MEMORY	
WCSHDRERR_BAD_COLUMN	
WCSHDRERR_PARSER	
WCSHDRERR_BAD_TABULAR_PARAMS	

19.29.4 Function Documentation

19.29.4.1 wcspih()   int wcspih (
          char * *header,*
          int *nkeyrec,*
          int *relax,*
          int *ctrl,*
          int * *nreject,*
          int * *nwcs,*
          struct wcsprm ** *wcs*)

wcspih() is a high-level FITS WCS routine that parses an image header, either that of a primary HDU or of an image extension. All WCS keywords defined in Papers I, II, III, IV, and VII are recognized, and also those used by the AIPS convention and certain other keywords that existed in early drafts of the WCS papers as explained in wcsbth() note 5. **wcspih**() also handles keywords associated with non-standard distortion functions described in the prologue of dis.h.

Given a character array containing a FITS image header, **wcspih**() identifies and reads all WCS keywords for the primary coordinate representation and up to 26 alternate representations. It returns this information as an array of wcsprm structs.

wcspih() invokes wcstab() on each of the wcsprm structs that it returns.

Use wcsbth() in preference to **wcspih**() for FITS headers of unknown type; wcsbth() can parse image headers as well as binary table and pixel list headers, although it cannot handle keywords relating to distortion functions, which may only exist in an image header (primary or extension).

Parameters

in,out	*header*	Character array containing the (entire) FITS image header from which to identify and construct the coordinate representations, for example, as might be obtained conveniently via the CFITSIO routine *fits_hdr2str()*. Each header "keyrecord" (formerly "card image") consists of exactly 80 7-bit ASCII printing characters in the range 0x20 to 0x7e (which excludes NUL, BS, TAB, LF, FF and CR) especially noting that the keyrecords are NOT null-terminated. For negative values of ctrl (see below), header[] is modified so that WCS keyrecords processed by **wcspih**() are removed from it.
in	*nkeyrec*	Number of keyrecords in header[].

in	*relax*	Degree of permissiveness:
		
		    • 0: Recognize only FITS keywords defined by the published WCS standard.
		
		    • WCSHDR_all: Admit all recognized informal extensions of the WCS standard.
		
		Fine-grained control of the degree of permissiveness is also possible as explained in wcsbth() note 5.

Parameters

in	*ctrl*	Error reporting and other control options for invalid WCS and other header keyrecords:
		• 0: Do not report any rejected header keyrecords.
		• 1: Produce a one-line message stating the number of WCS keyrecords rejected (nreject).
		• 2: Report each rejected keyrecord and the reason why it was rejected.
		• 3: As above, but also report all non-WCS keyrecords that were discarded, and the number of coordinate representations (nwcs) found.
		• 4: As above, but also report the accepted WCS keyrecords, with a summary of the number accepted as well as rejected.
		The report is written to stderr by default, or the stream set by wcsprintf_set().
		For ctrl $<$ 0, WCS keyrecords processed by **wcspih**() are removed from header[]:
		• -1: Remove only valid WCS keyrecords whose values were successfully extracted, nothing is reported.
		• -2: As above, but also remove WCS keyrecords that were rejected, reporting each one and the reason that it was rejected.
		• -3: As above, and also report the number of coordinate representations (nwcs) found.
		• -11: Same as -1 but preserving global WCS-related keywords such as '{**DATE**,**MJD**}−{**OBS**,**BEG**,**AVG**,**END**}' and the other basic time-related keywords, and '**OBSGEO**−{**X**,**Y**,**Z**,**L**,**B**,**H**}'.
		If any keyrecords are removed from header[] it will be null-terminated (NUL not being a legal FITS header character), otherwise it will contain its original complement of nkeyrec keyrecords and possibly not be null-terminated.
out	*nreject*	Number of WCS keywords rejected for syntax errors, illegal values, etc. Keywords not recognized as WCS keywords are simply ignored. Refer also to wcsbth() note 5.
out	*nwcs*	Number of coordinate representations found.
out	*wcs*	Pointer to an array of wcsprm structs containing up to 27 coordinate representations. Memory for the array is allocated by **wcspih**() which also invokes wcsini() for each struct to allocate memory for internal arrays and initialize their members to default values. Refer also to wcsbth() note 8. Note that wcsset() is not invoked on these structs.
		This allocated memory must be freed by the user, first by invoking wcsfree() for each struct, and then by freeing the array itself. A routine, wcsvfree(), is provided to do this (see below).

Returns

Status return value:

- 0: Success.
- 1: Null wcsprm pointer passed.
- 2: Memory allocation failed.
- 4: Fatal error returned by Flex parser.

Notes:

1. Refer to wcsbth() notes 1, 2, 3, 5, 7, and 8.

19.29.4.2   wcsbth() `int wcsbth (`
          `char * header,`
          `int nkeyrec,`
          `int relax,`
          `int ctrl,`
          `int keysel,`
          `int * colsel,`
          `int * nreject,`
          `int * nwcs,`
          `struct wcsprm ** wcs )`

wcsbth() is a high-level FITS WCS routine that parses a binary table header. It handles image array and pixel list WCS keywords which may be present together in one header.

As an extension of the FITS WCS standard, **wcsbth**() also recognizes image header keywords in a binary table header. These may be used to provide default values via an inheritance mechanism discussed in note 5 (c.↩ f. WCSHDR_AUXIMG and WCSHDR_ALLIMG), or may instead result in wcsprm structs that are not associated with any particular column. Thus **wcsbth**() can handle primary image and image extension headers in addition to binary table headers (it ignores **NAXIS** and does not rely on the presence of the **TFIELDS** keyword).

All WCS keywords defined in Papers I, II, III, and VII are recognized, and also those used by the AIPS convention and certain other keywords that existed in early drafts of the WCS papers as explained in note 5 below.

wcsbth() sets the colnum or colax[] members of the wcsprm structs that it returns with the column number of an image array or the column numbers associated with each pixel coordinate element in a pixel list. wcsprm structs that are not associated with any particular column, as may be derived from image header keywords, have colnum == 0.

Note 6 below discusses the number of wcsprm structs returned by **wcsbth**(), and the circumstances in which image header keywords cause a struct to be created. See also note 9 concerning the number of separate images that may be stored in a pixel list.

The API to **wcsbth**() is similar to that of wcspih() except for the addition of extra arguments that may be used to restrict its operation. Like wcspih(), **wcsbth**() invokes wcstab() on each of the wcsprm structs that it returns.

Parameters

in,out	*header*	Character array containing the (entire) FITS binary table, primary image, or image extension header from which to identify and construct the coordinate representations, for example, as might be obtained conveniently via the CFITSIO routine *fits_hdr2str()*. Each header "keyrecord" (formerly "card image") consists of exactly 80 7-bit ASCII printing characters in the range 0x20 to 0x7e (which excludes NUL, BS, TAB, LF, FF and CR) especially noting that the keyrecords are NOT null-terminated. For negative values of ctrl (see below), header[] is modified so that WCS keyrecords processed by **wcsbth**() are removed from it.
in	*nkeyrec*	Number of keyrecords in header[].
in	*relax*	Degree of permissiveness: • 0: Recognize only FITS keywords defined by the published WCS standard. • WCSHDR_all: Admit all recognized informal extensions of the WCS standard. Fine-grained control of the degree of permissiveness is also possible, as explained in note 5 below.

Parameters

in	*ctrl*	Error reporting and other control options for invalid WCS and other header keyrecords:
		• 0: Do not report any rejected header keyrecords.
		• 1: Produce a one-line message stating the number of WCS keyrecords rejected (nreject).
		• 2: Report each rejected keyrecord and the reason why it was rejected.
		• 3: As above, but also report all non-WCS keyrecords that were discarded, and the number of coordinate representations (nwcs) found.
		• 4: As above, but also report the accepted WCS keyrecords, with a summary of the number accepted as well as rejected.
		The report is written to stderr by default, or the stream set by wcsprintf_set().
		For ctrl < 0, WCS keyrecords processed by **wcsbth**() are removed from header[]:
		• -1: Remove only valid WCS keyrecords whose values were successfully extracted, nothing is reported.
		• -2: Also remove WCS keyrecords that were rejected, reporting each one and the reason that it was rejected.
		• -3: As above, and also report the number of coordinate representations (nwcs) found.
		• -11: Same as -1 but preserving global WCS-related keywords such as '{**DATE**,**MJD**}−{**OBS**,**BEG**,**AVG**,**END**}' and the other basic time-related keywords, and '**OBSGEO**−{**X**,**Y**,**Z**,**L**,**B**,**H**}'.
		If any keyrecords are removed from header[] it will be null-terminated (NUL not being a legal FITS header character), otherwise it will contain its original complement of nkeyrec keyrecords and possibly not be null-terminated.
in	*keysel*	Vector of flag bits that may be used to restrict the keyword types considered:
		• WCSHDR_IMGHEAD: Image header keywords.
		• WCSHDR_BIMGARR: Binary table image array.
		• WCSHDR_PIXLIST: Pixel list keywords.
		If zero, there is no restriction.
		Keywords such as **EQUI**na or **RFRQ**na that are common to binary table image arrays and pixel lists (including **WCSN**na and **TWCS**na, as explained in note 4 below) are selected by both WCSHDR_BIMGARR and WCSHDR_PIXLIST. Thus if inheritance via WCSHDR_ALLIMG is enabled as discussed in note 5 and one of these shared keywords is present, then WCSHDR_IMGHEAD and WCSHDR_PIXLIST alone may be sufficient to cause the construction of coordinate descriptions for binary table image arrays.

Parameters

in	*colsel*	Pointer to an array of table column numbers used to restrict the keywords considered by **wcsbth**().
		A null pointer may be specified to indicate that there is no restriction. Otherwise, the magnitude of cols[0] specifies the length of the array:
		• cols[0] > 0: the columns are included,
		• cols[0] < 0: the columns are excluded.
		For the pixel list keywords **TP**n_ka and **TC**n_ka (and **TPC**n_ka and **TCD**n_ka if WCSHDR_LONGKEY is enabled), it is an error for one column to be selected but not the other. This is unlike the situation with invalid keyrecords, which are simply rejected, because the error is not intrinsic to the header itself but arises in the way that it is processed.
out	*nreject*	Number of WCS keywords rejected for syntax errors, illegal values, etc. Keywords not recognized as WCS keywords are simply ignored, refer also to note 5 below.
out	*nwcs*	Number of coordinate representations found.
out	*wcs*	Pointer to an array of wcsprm structs containing up to 27027 coordinate representations, refer to note 6 below.
		Memory for the array is allocated by **wcsbth**() which also invokes wcsini() for each struct to allocate memory for internal arrays and initialize their members to default values. Refer also to note 8 below. Note that wcsset() is not invoked on these structs. This allocated memory must be freed by the user, first by invoking wcsfree() for each struct, and then by freeing the array itself. A routine, wcsvfree(), is provided to do this (see below).

Returns

Status return value:

- 0: Success.
- 1: Null wcsprm pointer passed.
- 2: Memory allocation failed.
- 3: Invalid column selection.
- 4: Fatal error returned by Flex parser.

Notes:

1. wcspih() determines the number of coordinate axes independently for each alternate coordinate representation (denoted by the "a" value in keywords like **CTYPE**ia) from the higher of

   a **NAXIS**,

   b **WCSAXES**a,

   c The highest axis number in any parameterized WCS keyword. The keyvalue, as well as the keyword, must be syntactically valid otherwise it will not be considered.

   If none of these keyword types is present, i.e. if the header only contains auxiliary WCS keywords for a particular coordinate representation, then no coordinate description is constructed for it.

   wcsbth() is similar except that it ignores the **NAXIS** keyword if given an image header to process.

   The number of axes, which is returned as a member of the wcsprm struct, may differ for different coordinate representations of the same image.

2. wcspih() and **wcsbth**() enforce correct FITS "keyword = value" syntax with regard to "= " occurring in columns 9 and 10.

   However, they do recognize free-format character (NOST 100-2.0, Sect. 5.2.1), integer (Sect. 5.2.3), and floating-point values (Sect. 5.2.4) for all keywords.

3. Where **CROTA**n, **CD**i_ja, and **PC**i_ja occur together in one header wcspih() and **wcsbth**() treat them as described in the prologue to wcs.h.

4. WCS Paper I mistakenly defined the pixel list form of **WCSNAME**a as **TWCS**na instead of **WCSN**na; the **'T'** is meant to substitute for the axis number in the binary table form of the keyword - note that keywords defined in WCS Papers II, III, and VII that are not parameterized by axis number have identical forms for binary tables and pixel lists. Consequently **wcsbth**() always treats **WCSN**na and **TWCS**na as equivalent.

5. wcspih() and **wcsbth**() interpret the *relax* argument as a vector of flag bits to provide fine-grained control over what non-standard WCS keywords to accept. The flag bits are subject to change in future and should be set by using the preprocessor macros (see below) for the purpose.

   - WCSHDR_none: Don't accept any extensions (not even those in the errata). Treat non-conformant keywords in the same way as non-WCS keywords in the header, i.e. simply ignore them.
   - WCSHDR_all: Accept all extensions recognized by the parser.
   - WCSHDR_reject: Reject non-standard keyrecords (that are not otherwise explicitly accepted by one of the flags below). A message will optionally be printed on stderr by default, or the stream set by wcsprintf_set(), as determined by the ctrl argument, and nreject will be incremented.

     This flag may be used to signal the presence of non-standard keywords, otherwise they are simply passed over as though they did not exist in the header. It is mainly intended for testing conformance of a FITS header to the WCS standard.

     Keyrecords may be non-standard in several ways:

     - The keyword may be syntactically valid but with keyvalue of incorrect type or invalid syntax, or the keycomment may be malformed.
     - The keyword may strongly resemble a WCS keyword but not, in fact, be one because it does not conform to the standard. For example, "CRPIX01" looks like a **CRPIX**ja keyword, but in fact the leading zero on the axis number violates the basic FITS standard. Likewise, "LONPOLE2" is not a valid **LONPOLE**a keyword in the WCS standard, and indeed there is nothing the parser can sensibly do with it.
     - Use of the keyword may be deprecated by the standard. Such will be rejected if not explicitly accepted via one of the flags below.

   - WCSHDR_strict: As for WCSHDR_reject, but also reject AIPS-convention keywords and all other deprecated usage that is not explicitly accepted.
   - WCSHDR_CROTAia: Accept **CROTA**ia (wcspih()), i**CROT**na (**wcsbth**()), **TCROT**na (**wcsbth**()).
   - WCSHDR_VELREFa: Accept **VELREF**a. wcspih() always recognizes the AIPS-convention keywords, **CROTA**n, **EPOCH**, and **VELREF** for the primary representation (a = ' ') but alternates are non-standard. **wcsbth**() accepts **EPOCH**a and **VELREF**a only if WCSHDR_AUXIMG is also enabled.
   - WCSHDR_CD00i00j: Accept **CD00**i**00**j (wcspih()).
   - WCSHDR_PC00i00j: Accept **PC00**i**00**j (wcspih()).
   - WCSHDR_PROJPn: Accept **PROJP**n (wcspih()). These appeared in early drafts of WCS Paper I+II (before they were split) and are equivalent to **CD**i_ja, **PC**i_ja, and **PV**i_ma for the primary representation (a = ' '). **PROJP**n is equivalent to **PV**i_ma with m = n ≤ 9, and is associated exclusively with the latitude axis.
   - WCSHDR_CD0i_0ja: Accept CD0i_0ja (wcspih()).
   - WCSHDR_PC0i_0ja: Accept PC0i_0ja (wcspih()).
   - WCSHDR_PV0i_0ma: Accept PV0i_0ja (wcspih()).
   - WCSHDR_PS0i_0ma: Accept PS0i_0ja (wcspih()). Allow the numerical index to have a leading zero in doubly- parameterized keywords, for example, PC01_01. WCS Paper I (Sects 2.1.2 & 2.1.4) explicitly disallows leading zeroes. The FITS 3.0 standard document (Sect. 4.1.2.1) states that the index in singly-parameterized keywords (e.g. **CTYPE**ia) "shall not have leading zeroes", and later in Sect. 8.1 that "leading zeroes must not be used" on **PV**i_ma and **PS**i_ma. However, by an oversight, it is silent on **PC**i_ja and **CD**i_ja.

- WCSHDR_DOBSn (**wcsbth**() only): Allow **DOBS**n, the column-specific analogue of **DATE−OBS**. By an oversight this was never formally defined in the standard.

- WCSHDR_OBSGLBHn (**wcsbth**() only): Allow **OBSGL**n, **OBSGB**n, and **OBSGH**n, the column-specific analogues of **OBSGEO−L**, **OBSGEO−B**, and **OBSGEO−H**. By an oversight these were never formally defined in the standard.

- WCSHDR_RADECSYS: Accept **RADECSYS**. This appeared in early drafts of WCS Paper I+II and was subsequently replaced by **RADESYS**a.

  wcsbth() accepts **RADECSYS** only if WCSHDR_AUXIMG is also enabled.

- WCSHDR_EPOCHa: Accept **EPOCH**a.

- WCSHDR_VSOURCE: Accept **VSOURCE**a or **VSOU**na (**wcsbth**()). This appeared in early drafts of WCS Paper III and was subsequently dropped in favour of **ZSOURCE**a and **ZSOU**na.

  wcsbth() accepts **VSOURCE**a only if WCSHDR_AUXIMG is also enabled.

- #WCSHDR_<TT>**DATEREF**: Accept DATE-REF, MJD-REF, MJD-REFI, MJD-REFF, **JDREF**, JD-↩
  REFI, and JD-REFF as synonyms for the standard keywords, **DATEREF**, **MJDREF**, **MJDREFI**,
  MJDREFF, **JDREF**, **JDREFI**, and **JDREFF**. The latter buck the pattern set by the other date key-
  words   ({DATE,**MJD**}−{**OBS,BEG,AVG,**END}), thereby increasing the potential
  for confusion and error.

- WCSHDR_LONGKEY (**wcsbth**() only): Accept long forms of the alternate
  binary table and pixel list WCS keywords, i.e. with "a" non- blank.
  Specifically

j**CRPX**na	**TCRPX**na	↩:	j**CRPX**n	j**CRP**na	**TCRPX**n	**TCRP**na	**CRPIX**ja
	TPCn_ka	↩:		ij**PC**na		**TP**n_ka	**PC**i_ja
	TCDn_ka	↩:		ij**CD**na		**TC**n_ka	**CD**i_ja
i**CDLT**na	**TCDLT**na	↩:	i**CDLT**n	i**CDE**na	**TCDLT**n	**TCDE**na	**CDELT**ia
i**CUNI**na	**TCUNI**na	↩:	i**CUNI**n	i**CUN**na	**TCUNI**n	**TCUN**na	**CUNIT**ia
i**CTYP**na	**TCTYP**na	↩:	i**CTYP**n	i**CTY**na	**TCTYP**n	**TCTY**na	**CTYPE**ia
i**CRVL**na	**TCRVL**na	↩:	i**CRVL**n	i**CRV**na	**TCRVL**n	**TCRV**na	**CRVAL**ia
i**PV**n_ma	**TPV**n_ma	↩:		i**V**n_ma		**TV**n_ma	**PV**i_ma
i**PS**n_ma	**TPS**n_ma	↩:		i**S**n_ma		**TS**n_ma	**PS**i_ma

where the primary and standard alternate forms together with the
image−header equivalent are shown rightwards of the colon.

The long form of these keywords could be described as quasi− standard.
TPCn_ka, i**PV**n_ma, and **TPV**n_ma appeared by mistake in the examples
in WCS Paper II and subsequently these and also **TCD**n_ka, i**PS**n_ma and
TPSn_ma were legitimized by the errata to the WCS papers.

Strictly speaking, the other long forms are non−standard and in fact
have never appeared in any draft of the WCS papers nor in the errata.
However, as natural extensions of the primary form they are unlikely
to be written with any other intention. Thus it should be safe to
accept them provided, of course, that the resulting keyword does not
exceed the 8−character limit.

If WCSHDR_CNAMn is enabled then also accept

i**CNAM**na	**TCNAM**na	↩:	−−	i**CNA**na	−−	**TCNA**na	**CNAME**ia

iCRDEna	TCRDEna	↩ :	――	iCRDna	――	TCRDna	CRDERia
iCSYEna	TCSYEna	↩ :	――	iCSYna	――	TCSYna	CSYERia
TCZPHna	TCZPHna	↩ :	――	TCZPna	――	TCZPna	CZPHSia
iCPERna	TCPERna	↩ :	――	iCPRna	――	TCPRna	CPERIia

Note that **CNAME**ia, **CRDER**ia, **CSYER**ia, CZPHSia, CPERIia, and their variants are not used by WCSLIB but are stored in the wcsprm struct as auxiliary information.

- WCSHDR_CNAMn (**wcsbth**() only): Accept i**CNAM**n, i**CRDE**n, i**CSYE**n, **TCZPH**n, i**CPER**n, **TCNAM**n, **TCRDE**n, **TCSYE**n, **TCZPH**n, and **TCPER**n, i.e. with "a" blank. While non-standard, these are the obvious analogues of i**CTYP**n, **TCTYP**n, etc.

- WCSHDR_AUXIMG (**wcsbth**() only): Allow the image-header form of an auxiliary WCS keyword with representation-wide scope to provide a default value for all images. This default may be overridden by the column-specific form of the keyword.

  For example, a keyword like **EQUINOX**a would apply to all image arrays in a binary table, or all pixel list columns with alternate representation "a" unless overridden by **EQUI**na.

  Specifically the keywords are:

LONPOLEa	for **LONP**na	
LATPOLEa	for **LATP**na	
VELREF		... (No column-specific form.)
VELREFa		... Only if WCSHDR_VELREFa is set.

whose keyvalues are actually used by WCSLIB, and also keywords providing auxiliary information that is simply stored in the wcsprm struct:

WCSNAMEa	for **WCSN**na	... Or **TWCS**na (see below).
DATE-OBS	for **DOBS**n	
MJD-OBS	for **MJDOB**n	
RADESYSa	for **RADE**na	
RADECSYS	for **RADE**na	... Only if WCSHDR_RADECSYS is set.
EPOCH		... (No column-specific form.)
EPOCHa		... Only if WCSHDR_EPOCHa is set.
EQUINOXa	for **EQUI**na	

where the image-header keywords on the left provide default values for the column specific keywords on the right.

Note that, according to Sect. 8.1 of WCS Paper III, and Sect. 5.2 of WCS Paper VII, the following are always inherited:

RESTFREQ	for **RFRQ**na
RESTFRQa	for **RFRQ**na
RESTWAVa	for **RWAV**na

being those actually used by WCSLIB, together with the following auxiliary keywords, many of which do not have binary table equivalents

and therefore can only be inherited:

TIMESYS		
TREFPOS	for **MJDA**n	
TREFDIR	for **MJDA**n	
PLEPHEM		
TIMEUNIT		
DATEREF		
MJDREF		
MJDREFI		
MJDREFF		
JDREF		
JDREFI		
JDREFF		
TIMEOFFS		
DATE−BEG		
DATE−AVG	for **DAVG**n	
DATE−END		
MJD−BEG		
MJD−AVG	for **MJDA**n	
MJD−END		
JEPOCH		
BEPOCH		
TSTART		
TSTOP		
XPOSURE		
TELAPSE		
TIMSYER		
TIMRDER		
TIMEDEL		
TIMEPIXR		
OBSGEO−X	for **OBSGX**n	
OBSGEO−Y	for **OBSGY**n	
OBSGEO−Z	for **OBSGZ**n	
OBSGEO−L	for **OBSGL**n	
OBSGEO−B	for **OBSGB**n	
OBSGEO−H	for **OBSGH**n	
OBSORBIT		
SPECSYSa	for **SPEC**na	
SSYSOBSa	for **SOBS**na	
VELOSYSa	for **VSYS**na	
VSOURCEa	for **VSOU**na	... Only if WCSHDR_VSOURCE is set.
ZSOURCEa	for **ZSOU**na	
SSYSSRCa	for **SSRC**na	
VELANGLa	for **VANG**na	

Global image−header keywords, such as **MJD−OBS,** apply to all alternate
representations, and would therefore provide a default value for all
images in the header.

This auxiliary inheritance mechanism applies to binary table image
arrays and pixel lists alike. Most of these keywords have no default
value, the exceptions being **LONPOLE**a and **LATPOLE**a, and also **RADESYS**a
and **EQUINOX**a which provide defaults for each other. Thus one potential

difficulty in using WCSHDR_AUXIMG is that of erroneously inheriting one of these four keywords.

Also, beware of potential inconsistencies that may arise where, for example, **DATE-OBS** is inherited, but **MJD-OBS** is overridden by **MJDOB**n and specifies a different time. Pairs in this category are:

```
      DATE-OBS/DOBSn         versus        MJD-OBS/MJDOBn
      DATE-AVG/DAVGn         versus        MJD-AVG/MJDAn
      RESTFRQa/RFRQna        versus        RESTWAVa/RWAVna
 OBSGEO-[XYZ]/OBSG[XYZ]n  versus  OBSGEO-[LBH]/OBSG[LBH]n
```

The wcsfixi() routines datfix() and obsfix() are provided to check the consistency of these and other such pairs of keywords.

Unlike WCSHDR_ALLIMG, the existence of one (or all) of these auxiliary WCS image header keywords will not by itself cause a wcsprm struct to be created for alternate representation "a". This is because they do not provide sufficient information to create a non-trivial coordinate representation when used in conjunction with the default values of those keywords that are parameterized by axis number, such as **CTYPE**ia.

- WCSHDR_ALLIMG (**wcsbth**() only): Allow the image-header form of *all* image header WCS keywords to provide a default value for all image arrays in a binary table (n.b. not pixel list). This default may be overridden by the column-specific form of the keyword.

  For example, a keyword like **CRPIX**ja would apply to all image arrays in a binary table with alternate representation "a" unless overridden by j**CRP**na.

  Specifically the keywords are those listed above for WCSHDR_AUXIMG plus

WCSAXESa	for **WCAX**na

which defines the coordinate dimensionality, and the following keywords that are parameterized by axis number:

CRPIXja	for j**CRP**na	
PCi_ja	for ij**PC**na	
CDi_ja	for ij**CD**na	
CDELTia	for i**CDE**na	
CROTAi	for i**CROT**n	
CROTAia		... Only if WCSHDR_CROTAia is set.
CUNITia	for i**CUN**na	
CTYPEia	for i**CTY**na	
CRVALia	for i**CRV**na	
PVi_ma	for i**V**n_ma	
PSi_ma	for i**S**n_ma	
CNAMEia	for i**CNA**na	
CRDERia	for i**CRD**na	
CSYERia	for i**CSY**na	
CZPHSia	for **TCZP**na	
CPERIia	for i**CPR**na	

where the image-header keywords on the left provide default values for the column specific keywords on the right.

This full inheritance mechanism only applies to binary table image arrays, not pixel lists, because in the latter case there is no well-defined association between coordinate axis number and column number (see note 9 below).

Note that **CNAME**ia, **CRDER**ia, **CSYER**ia, and their variants are not used by WCSLIB but are stored in the wcsprm struct as auxiliary information. Note especially that at least one wcsprm struct will be returned for each "a" found in one of the image header keywords listed above:

- If the image header keywords for "a" *are not* inherited by a binary table, then the struct will not be associated with any particular table column number and it is up to the user to provide an association.
- If the image header keywords for "a" *are* inherited by a binary table image array, then those keywords are considered to be "exhausted" and do not result in a separate wcsprm struct.

For example, to accept **CD00**i**00**j and **PC00**i**00**j and reject all other extensions, use

relax = WCSHDR_reject | WCSHDR_CD00i00j | WCSHDR_PC00i00j;

The parser always treats **EPOCH** as subordinate to **EQUINOX**a if both are present, and **VSOURCE**a is always subordinate to **ZSOURCE**a.

Likewise, **VELREF** is subordinate to the formalism of WCS Paper III, see spcaips().

Neither wcspih() nor **wcsbth**() currently recognize the AIPS-convention keywords **ALTRPIX** or **ALTRVAL** which effectively define an alternative representation for a spectral axis.

6. Depending on what flags have been set in its *relax* argument, **wcsbth**() could return as many as 27027 wcsprm structs:

   - Up to 27 unattached representations derived from image header keywords.
   - Up to 27 structs for each of up to 999 columns containing an image arrays.
   - Up to 27 structs for a pixel list.

   Note that it is considered legitimate for a column to contain an image array and also form part of a pixel list, and in particular that **wcsbth**() does not check the **TFORM** keyword for a pixel list column to check that it is scalar.

   In practice, of course, a realistic binary table header is unlikely to contain more than a handful of images.

   In order for **wcsbth**() to create a wcsprm struct for a particular coordinate representation, at least one WCS keyword that defines an axis number must be present, either directly or by inheritance if WCSHDR_ALLIMG is set.

   When the image header keywords for an alternate representation are inherited by a binary table image array via WCSHDR_ALLIMG, those keywords are considered to be "exhausted" and do not result in a separate wcsprm struct. Otherwise they do.

7. Neither wcspih() nor **wcsbth**() check for duplicated keywords, in most cases they accept the last encountered.

8. wcspih() and **wcsbth**() use wcsnpv() and wcsnps() (refer to the prologue of wcs.h) to match the size of the pv[] and ps[] arrays in the wcsprm structs to the number in the header. Consequently there are no unused elements in the pv[] and ps[] arrays, indeed they will often be of zero length.

9. The FITS WCS standard for pixel lists assumes that a pixel list defines one and only one image, i.e. that each row of the binary table refers to just one event, e.g. the detection of a single photon or neutrino,

for which the device "pixel" coordinates are stored in separate scalar columns of the table.

In the absence of a standard for pixel lists – or even an informal description! – let alone a formal mechanism for identifying the columns containing pixel coordinates (as opposed to pixel values or metadata recorded at the time the photon or neutrino was detected), WCS Paper I discusses how the WCS keywords themselves may be used to identify them.

In practice, however, pixel lists have been used to store multiple images. Besides not specifying how to identify columns, the pixel list convention is also silent on the method to be used to associate table columns with image axes.

An additional shortcoming is the absence of a formal method for associating global binary-table WCS keywords, such as **WCSN**na or **MJDOB**n, with a pixel list image, whether one or several.

In light of these uncertainties, **wcsbth**() simply collects all WCS keywords for a particular pixel list coordinate representation (i.e. the "a" value in **TCTY**na) into one wcsprm struct. However, these alternates need not be associated with the same table columns and this allows a pixel list to contain up to 27 separate images. As usual, if one of these representations happened to contain more than two celestial axes, for example, then an error would result when wcsset() is invoked on it. In this case the "colsel" argument could be used to restrict the columns used to construct the representation so that it only contained one pair of celestial axes.

Global, binary-table WCS keywords are considered to apply to the pixel list image with matching alternate (e.g. the "a" value in **LONP**na or **EQUI**na), regardless of the table columns the image occupies. In other words, the column number is ignored (the "n" value in **LONP**na or **EQUI**na). This also applies for global, binary-table WCS keywords that have no alternates, such as **MJDOB**n and **OBSGX**n, which match all images in a pixel list. Take heed that this may lead to counterintuitive behaviour, especially where such a keyword references a column that does not store pixel coordinates, and moreso where the pixel list stores only a single image. In fact, as the column number, n, is ignored for such keywords, it would make no difference even if they referenced non-existent columns. Moreover, there is no requirement for consistency in the column numbers used for such keywords, even for **OBSGX**n, **OBSGY**n, and **OBSGZ**n which are meant to define the elements of a coordinate vector. Although it would surely be perverse to construct a pixel list like this, such a situation may still arise in practice where columns are deleted from a binary table.

The situation with global, binary-table WCS keywords becomes potentially even more confusing when image arrays and pixel list images coexist in one binary table. In that case, a keyword such as **MJDOB**n may legitimately appear multiple times with n referencing different image arrays. Which then is the one that applies to the pixel list images? In this implementation, it is the last instance that appears in the header, whether or not it is also associated with an image array.

19.29.4.3 wcstab() int wcstab (
            struct wcsprm * wcs)

wcstab() assists in filling in the information in the wcsprm struct relating to coordinate lookup tables.

Tabular coordinates (`'TAB'`) present certain difficulties in that the main components of the lookup table - the multidimensional coordinate array plus an index vector for each dimension - are stored in a FITS binary table extension (BINTABLE). Information required to locate these arrays is stored in **PV**i_ma and **PS**i_ma keywords in the image header.

wcstab() parses the **PV**i_ma and **PS**i_ma keywords associated with each `'TAB'` axis and allocates memory in the wcsprm struct for the required number of tabprm structs. It sets as much of the tabprm struct as can be gleaned from the image header, and also sets up an array of wtbarr structs (described in the prologue of wtbarr.h) to assist in extracting the required arrays from the BINTABLE extension(s).

It is then up to the user to allocate memory for, and copy arrays from the BINTABLE extension(s) into the tabprm structs. A CFITSIO routine, fits_read_wcstab(), has been provided for this purpose, see getwcstab.h. wcsset() will automatically take control of this allocated memory, in particular causing it to be freed by wcsfree(); the user must not attempt to free it after wcsset() has been called.

Note that wcspih() and wcsbth() automatically invoke **wcstab**() on each of the wcsprm structs that they return.

Parameters

in,out	*wcs*	Coordinate transformation parameters (see below).
		wcstab() sets ntab, tab, nwtb and wtb, allocating memory for the tab and wtb arrays. This allocated memory will be freed automatically by wcsfree().

Returns

Status return value:

- 0: Success.
- 1: Null wcsprm pointer passed.
- 2: Memory allocation failed.
- 3: Invalid tabular parameters.

For returns > 1, a detailed error message is set in wcsprm::err if enabled, see wcserr_enable().

19.29.4.4   wcsidx()   `int wcsidx (`
`          int nwcs,`
`          struct wcsprm ** wcs,`
`          int alts[27] )`

wcsidx() returns an array of 27 indices for the alternate coordinate representations in the array of wcsprm structs returned by wcspih(). For the array returned by wcsbth() it returns indices for the unattached (colnum == 0) representations derived from image header keywords - use wcsbdx() for those derived from binary table image arrays or pixel lists keywords.

Parameters

in	*nwcs*	Number of coordinate representations in the array.
in	*wcs*	Pointer to an array of wcsprm structs returned by wcspih() or wcsbth().
out	*alts*	Index of each alternate coordinate representation in the array: alts[0] for the primary, alts[1] for 'A', etc., set to -1 if not present.
		For example, if there was no 'P' representation then
		`alts['P'-'A'+1] == -1;`
		Otherwise, the address of its wcsprm struct would be
		`wcs + alts['P'-'A'+1];`

Returns

Status return value:

- 0: Success.

- 1: Null wcsprm pointer passed.

19.29.4.5 wcsbdx() `int wcsbdx (`
          `int nwcs,`
          `struct wcsprm ** wcs,`
          `int type,`
          `short alts[1000][28] )`

wcsbdx() returns an array of 999 x 27 indices for the alternate coordinate representions for binary table image arrays xor pixel lists in the array of wcsprm structs returned by wcsbth(). Use wcsidx() for the unattached representations derived from image header keywords.

Parameters

in	*nwcs*	Number of coordinate representations in the array.
in	*wcs*	Pointer to an array of wcsprm structs returned by wcsbth().

in	*type*	Select the type of coordinate representation:
		
		    • 0: binary table image arrays,
		
		    • 1: pixel lists.

out	*alts*	Index of each alternate coordinate represention in the array: alts[col][0] for the primary, alts[col][1] for 'A', to alts[col][26] for 'Z', where col is the 1-relative column number, and col == 0 is used for unattached image headers. Set to -1 if not present.
		alts[col][27] counts the number of coordinate representations of the chosen type for each column.
		For example, if there was no 'P' representation for column 13 then
		`alts[13]['P'-'A'+1] == -1;`
		
		Otherwise, the address of its wcsprm struct would be
		`wcs + alts[13]['P'-'A'+1];`

Returns

Status return value:

- 0: Success.
- 1: Null wcsprm pointer passed.

19.29.4.6 wcsvfree() `int wcsvfree (`
          `int * nwcs,`
          `struct wcsprm ** wcs )`

wcsvfree() frees the memory allocated by wcspih() or wcsbth() for the array of wcsprm structs, first invoking wcsfree() on each of the array members.

Parameters

in,out	*nwcs*	Number of coordinate representations found; set to 0 on return.
in,out	*wcs*	Pointer to the array of wcsprm structs; set to 0x0 on return.

Returns

Status return value:

- 0: Success.
- 1: Null wcsprm pointer passed.

19.29.4.7   wcshdo()   int wcshdo (
        int *ctrl,*
        struct wcsprm * *wcs,*
        int * *nkeyrec,*
        char ** *header*)

wcshdo() translates a wcsprm struct into a FITS header. If the colnum member of the struct is non-zero then a binary table image array header will be produced. Otherwise, if the colax[] member of the struct is set non-zero then a pixel list header will be produced. Otherwise, a primary image or image extension header will be produced.

If the struct was originally constructed from a header, e.g. by wcspih(), the output header will almost certainly differ in a number of respects:

- The output header only contains WCS-related keywords. In particular, it does not contain syntactically-required keywords such as **SIMPLE**, **NAXIS**, **BITPIX**, or **END**.

- Elements of the **PC**i_ja matrix will be written if and only if they differ from the unit matrix. Thus, if the matrix is unity then no elements will be written.

- The redundant keywords **MJDREF**, **JDREF**, **JDREFI**, **JDREFF**, all of which duplicate **MJDREFI** + **MJDREFF**, are never written. OBSGEO-[LBH] are not written if OBSGEO-[XYZ] are defined.

- Deprecated (e.g. **CROTA**n, **RESTFREQ**, **VELREF**, **RADECSYS**, **EPOCH**, **VSOURCE**a) or non-standard usage will be translated to standard (this is partially dependent on whether wcsfix() was applied).

- Additional keywords such as **WCSAXES**a, **CUNIT**ia, **LONPOLE**a and **LATPOLE**a may appear.

- Quantities will be converted to the units used internally, basically SI with the addition of degrees.

- Floating-point quantities may be given to a different decimal precision.

- The original keycomments will be lost, although **wcshdo**() tries hard to write meaningful comments.

- Keyword order will almost certainly be changed.

Keywords can be translated between the image array, binary table, and pixel lists forms by manipulating the colnum or colax[] members of the wcsprm struct.

Parameters

in	*ctrl*	Vector of flag bits that controls the degree of permissiveness in departing from the published WCS standard, and also controls the formatting of floating-point keyvalues. Set it to zero to get the default behaviour. Flag bits for the degree of permissiveness: • WCSHDO_none: Recognize only FITS keywords defined by the published WCS standard. • WCSHDO_all: Admit all recognized informal extensions of the WCS standard. Fine-grained control of the degree of permissiveness is also possible as explained in the notes below. As for controlling floating-point formatting, by default **wcshdo**() uses "%20.12G" for non-parameterized keywords such as **LONPOLE**a, and attempts to make the header more human-readable by using the same "f" format for all values of each of the following parameterized keywords: **CRPIX**ja, **PC**i_ja, and **CDELT**ia (n.b. excluding **CRVAL**ia). Each has the same field width and precision so that the decimal points line up. The precision, allowing for up to 15 significant digits, is chosen so that there are no excess trailing zeroes. A similar formatting scheme applies by default for distortion function parameters. However, where the values of, for example, **CDELT**ia differ by many orders of magnitude, the default formatting scheme may cause unacceptable loss of precision for the lower-valued keyvalues. Thus the default behaviour may be overridden: • WCSHDO_P12: Use "%20.12G" format for all floating- point keyvalues (12 significant digits). • WCSHDO_P13: Use "%21.13G" format for all floating- point keyvalues (13 significant digits). • WCSHDO_P14: Use "%22.14G" format for all floating- point keyvalues (14 significant digits). • WCSHDO_P15: Use "%23.15G" format for all floating- point keyvalues (15 significant digits). • WCSHDO_P16: Use "%24.16G" format for all floating- point keyvalues (16 significant digits). • WCSHDO_P17: Use "%25.17G" format for all floating- point keyvalues (17 significant digits). If more than one of the above flags are set, the highest number of significant digits prevails. In addition, there is an anciliary flag: • WCSHDO_EFMT: Use "E" format instead of the default "G" format above. Note that excess trailing zeroes are stripped off the fractional part with "G" (which never occurs with "E"). Note also that the higher-precision options eat into the keycomment area. In this regard, WCSHDO_P14 causes minimal disruption with "G" format, while WCSHDO_P13 is appropriate with "E".
in,out	*wcs*	Pointer to a wcsprm struct containing coordinate transformation parameters. Will be initialized if necessary.
out	*nkeyrec*	Number of FITS header keyrecords returned in the "header" array.
out	*header*	Pointer to an array of char holding the header. Storage for the array is allocated by **wcshdo**() in blocks of 2880 bytes (32 x 80-character keyrecords) and must be freed by the user to avoid memory leaks. See wcsdealloc(). Each keyrecord is 80 characters long and is ∗NOT∗ null-terminated, so the first keyrecord starts at (∗header)[0], the second at (∗header)[80], etc.

Returns

Status return value (associated with wcs_errmsg[]):

- 0: Success.
- 1: Null wcsprm pointer passed.
- 2: Memory allocation failed.
- 3: Linear transformation matrix is singular.
- 4: Inconsistent or unrecognized coordinate axis types.
- 5: Invalid parameter value.
- 6: Invalid coordinate transformation parameters.
- 7: Ill-conditioned coordinate transformation parameters.

For returns > 1, a detailed error message is set in wcsprm::err if enabled, see wcserr_enable().

Notes:

1. **wcshdo**() interprets the *relax* argument as a vector of flag bits to provide fine-grained control over what non-standard WCS keywords to write. The flag bits are subject to change in future and should be set by using the preprocessor macros (see below) for the purpose.

   - WCSHDO_none: Don't use any extensions.
   - WCSHDO_all: Write all recognized extensions, equivalent to setting each flag bit.
   - WCSHDO_safe: Write all extensions that are considered to be safe and recommended.
   - WCSHDO_DOBSn: Write **DOBS**n, the column-specific analogue of **DATE−OBS** for use in binary tables and pixel lists. WCS Paper III introduced **DATE−AVG** and **DAVG**n but by an oversight **DOBS**n (the obvious analogy) was never formally defined by the standard. The alternative to using **DOBS**n is to write **DATE−OBS** which applies to the whole table. This usage is considered to be safe and is recommended.
   - WCSHDO_TPCn_ka: WCS Paper I defined

     – **TP**n_ka and **TC**n_ka for pixel lists

     but WCS Paper II uses **TPC**n_ka in one example and subsequently the errata for the WCS papers legitimized the use of

     – **TPC**n_ka and **TCD**n_ka for pixel lists

     provided that the keyword does not exceed eight characters. This usage is considered to be safe and is recommended because of the non-mnemonic terseness of the shorter forms.
   - WCSHDO_PVn_ma: WCS Paper I defined

     – i**V**n_ma and i**S**n_ma for bintables and
     – **TV**n_ma and **TS**n_ma for pixel lists

     but WCS Paper II uses i**PV**n_ma and **TPV**n_ma in the examples and subsequently the errata for the WCS papers legitimized the use of

     – i**PV**n_ma and i**PS**n_ma for bintables and
     – **TPV**n_ma and **TPS**n_ma for pixel lists

     provided that the keyword does not exceed eight characters. This usage is considered to be safe and is recommended because of the non-mnemonic terseness of the shorter forms.
   - WCSHDO_CRPXna: For historical reasons WCS Paper I defined

     – j**CRPX**n, i**CDLT**n, i**CUNI**n, i**CTYP**n, and i**CRVL**n for bintables and
     – **TCRPX**n, **TCDLT**n, **TCUNI**n, **TCTYP**n, and **TCRVL**n for pixel lists

     for use without an alternate version specifier. However, because of the eight-character keyword constraint, in order to accommodate column numbers greater than 99 WCS Paper I also defined

          – j**CRP**na, i**CDE**na, i**CUN**na, i**CTY**na and i**CRV**na for bintables and

          – **TCRP**na, **TCDE**na, **TCUN**na, **TCTY**na and **TCRV**na for pixel lists

for use with an alternate version specifier (the "a"). Like the PC, CD, PV, and PS keywords there is an obvious tendency to confuse these two forms for column numbers up to 99. It is very unlikely that any parser would reject keywords in the first set with a non-blank alternate version specifier so this usage is considered to be safe and is recommended.

- **WCSHDO_CNAMna**: WCS Papers I and III defined

          – i**CNA**na, i**CRD**na, and i**CSY**na for bintables and

          – **TCNA**na, **TCRD**na, and **TCSY**na for pixel lists

By analogy with the above, the long forms would be

          – i**CNAM**na, i**CRDE**na, and i**CSYE**na for bintables and

          – **TCNAM**na, **TCRDE**na, and **TCSYE**na for pixel lists

Note that these keywords provide auxiliary information only, none of them are needed to compute world coordinates. This usage is potentially unsafe and is not recommended at this time.

- **WCSHDO_WCSNna**: In light of wcsbth() note 4, write **WCSN**na instead of **TWCS**na for pixel lists. While wcsbth() treats **WCSN**na and **TWCS**na as equivalent, other parsers may not. Consequently, this usage is potentially unsafe and is not recommended at this time.

19.29.5 Variable Documentation

19.29.5.1 wcshdr_errmsg const char * wcshdr_errmsg[] [extern]

Error messages to match the status value returned from each function. Use wcs_errmsg[] for status returns from **wcshdo**().

19.30 wcshdr.h

Go to the documentation of this file.
```
1  /*============================================================================
2    WCSLIB 7.11 - an implementation of the FITS WCS standard.
3    Copyright (C) 1995-2022, Mark Calabretta
4
5    This file is part of WCSLIB.
6
7    WCSLIB is free software: you can redistribute it and/or modify it under the
8    terms of the GNU Lesser General Public License as published by the Free
9    Software Foundation, either version 3 of the License, or (at your option)
10   any later version.
11
12   WCSLIB is distributed in the hope that it will be useful, but WITHOUT ANY
13   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
14   FOR A PARTICULAR PURPOSE.  See the GNU Lesser General Public License for
15   more details.
16
17   You should have received a copy of the GNU Lesser General Public License
18   along with WCSLIB.  If not, see http://www.gnu.org/licenses.
19
20   Author: Mark Calabretta, Australia Telescope National Facility, CSIRO.
21   http://www.atnf.csiro.au/people/Mark.Calabretta
22   $Id: wcshdr.h,v 7.11 2022/04/26 06:13:52 mcalabre Exp $
23  *============================================================================
24  *
25  * WCSLIB 7.11 - C routines that implement the FITS World Coordinate System
26  * (WCS) standard.  Refer to the README file provided with WCSLIB for an
27  * overview of the library.
28  *
29  *
30  * Summary of the wcshdr routines
31  * ------------------------------
```

```
32 * Routines in this suite are aimed at extracting WCS information from a FITS
33 * file.  The information is encoded via keywords defined in
34 *
35 =   "Representations of world coordinates in FITS",
36 =   Greisen, E.W., & Calabretta, M.R. 2002, A&A, 395, 1061 (WCS Paper I)
37 =
38 =   "Representations of celestial coordinates in FITS",
39 =   Calabretta, M.R., & Greisen, E.W. 2002, A&A, 395, 1077 (WCS Paper II)
40 =
41 =   "Representations of spectral coordinates in FITS",
42 =   Greisen, E.W., Calabretta, M.R., Valdes, F.G., & Allen, S.L.
43 =   2006, A&A, 446, 747 (WCS Paper III)
44 =
45 =   "Representations of distortions in FITS world coordinate systems",
46 =   Calabretta, M.R. et al. (WCS Paper IV, draft dated 2004/04/22),
47 =   available from http://www.atnf.csiro.au/people/Mark.Calabretta
48 =
49 =   "Representations of time coordinates in FITS -
50 =    Time and relative dimension in space",
51 =   Rots, A.H., Bunclark, P.S., Calabretta, M.R., Allen, S.L.,
52 =   Manchester, R.N., & Thompson, W.T. 2015, A&A, 574, A36 (WCS Paper VII)
53 *
54 * These routines provide the high-level interface between the FITS file and
55 * the WCS coordinate transformation routines.
56 *
57 * Additionally, function wcshdo() is provided to write out the contents of a
58 * wcsprm struct as a FITS header.
59 *
60 * Briefly, the anticipated sequence of operations is as follows:
61 *
62 *   - 1: Open the FITS file and read the image or binary table header, e.g.
63 *        using CFITSIO routine fits_hdr2str().
64 *
65 *   - 2: Parse the header using wcspih() or wcsbth(); they will automatically
66 *        interpret 'TAB' header keywords using wcstab().
67 *
68 *   - 3: Allocate memory for, and read 'TAB' arrays from the binary table
69 *        extension, e.g. using CFITSIO routine fits_read_wcstab() - refer to
70 *        the prologue of getwcstab.h.  wcsset() will automatically take
71 *        control of this allocated memory, in particular causing it to be
72 *        freed by wcsfree().
73 *
74 *   - 4: Translate non-standard WCS usage using wcsfix(), see wcsfix.h.
75 *
76 *   - 5: Initialize wcsprm struct(s) using wcsset() and calculate coordinates
77 *        using wcsp2s() and/or wcss2p().  Refer to the prologue of wcs.h for a
78 *        description of these and other high-level WCS coordinate
79 *        transformation routines.
80 *
81 *   - 6: Clean up by freeing memory with wcsvfree().
82 *
83 * In detail:
84 *
85 * - wcspih() is a high-level FITS WCS routine that parses an image header.  It
86 *   returns an array of up to 27 wcsprm structs on each of which it invokes
87 *   wcstab().
88 *
89 * - wcsbth() is the analogue of wcspih() for use with binary tables; it
90 *   handles image array and pixel list keywords.  As an extension of the FITS
91 *   WCS standard, it also recognizes image header keywords which may be used
92 *   to provide default values via an inheritance mechanism.
93 *
94 * - wcstab() assists in filling in members of the wcsprm struct associated
95 *   with coordinate lookup tables ('TAB').  These are based on arrays stored
96 *   in a FITS binary table extension (BINTABLE) that are located by PVi_ma
97 *   keywords in the image header.
98 *
99 * - wcsidx() and wcsbdx() are utility routines that return the index for a
100 *   specified alternate coordinate descriptor in the array of wcsprm structs
101 *   returned by wcspih() or wcsbth().
102 *
103 * - wcsvfree() deallocates memory for an array of wcsprm structs, such as
104 *   returned by wcspih() or wcsbth().
105 *
106 * - wcshdo() writes out a wcsprm struct as a FITS header.
107 *
108 *
109 * wcspih() - FITS WCS parser routine for image headers
110 * ----------------------------------------------------
111 * wcspih() is a high-level FITS WCS routine that parses an image header,
112 * either that of a primary HDU or of an image extension.  All WCS keywords
113 * defined in Papers I, II, III, IV, and VII are recognized, and also those
114 * used by the AIPS convention and certain other keywords that existed in early
115 * drafts of the WCS papers as explained in wcsbth() note 5.  wcspih() also
116 * handles keywords associated with non-standard distortion functions described
117 * in the prologue of dis.h.
118 *
```

```
119 * Given a character array containing a FITS image header, wcspih() identifies
120 * and reads all WCS keywords for the primary coordinate representation and up
121 * to 26 alternate representations.  It returns this information as an array of
122 * wcsprm structs.
123 *
124 * wcspih() invokes wcstab() on each of the wcsprm structs that it returns.
125 *
126 * Use wcsbth() in preference to wcspih() for FITS headers of unknown type;
127 * wcsbth() can parse image headers as well as binary table and pixel list
128 * headers, although it cannot handle keywords relating to distortion
129 * functions, which may only exist in an image header (primary or extension).
130 *
131 * Given and returned:
132 *   header    char[]   Character array containing the (entire) FITS image
133 *                      header from which to identify and construct the
134 *                      coordinate representations, for example, as might be
135 *                      obtained conveniently via the CFITSIO routine
136 *                      fits_hdr2str().
137 *
138 *                      Each header "keyrecord" (formerly "card image")
139 *                      consists of exactly 80 7-bit ASCII printing characters
140 *                      in the range 0x20 to 0x7e (which excludes NUL, BS,
141 *                      TAB, LF, FF and CR) especially noting that the
142 *                      keyrecords are NOT null-terminated.
143 *
144 *                      For negative values of ctrl (see below), header[] is
145 *                      modified so that WCS keyrecords processed by wcspih()
146 *                      are removed from it.
147 *
148 * Given:
149 *   nkeyrec   int      Number of keyrecords in header[].
150 *
151 *   relax     int      Degree of permissiveness:
152 *                        0: Recognize only FITS keywords defined by the
153 *                           published WCS standard.
154 *                        WCSHDR_all: Admit all recognized informal
155 *                           extensions of the WCS standard.
156 *                      Fine-grained control of the degree of permissiveness
157 *                      is also possible as explained in wcsbth() note 5.
158 *
159 *   ctrl      int      Error reporting and other control options for invalid
160 *                      WCS and other header keyrecords:
161 *                          0: Do not report any rejected header keyrecords.
162 *                          1: Produce a one-line message stating the number
163 *                             of WCS keyrecords rejected (nreject).
164 *                          2: Report each rejected keyrecord and the reason
165 *                             why it was rejected.
166 *                          3: As above, but also report all non-WCS
167 *                             keyrecords that were discarded, and the number
168 *                             of coordinate representations (nwcs) found.
169 *                          4: As above, but also report the accepted WCS
170 *                             keyrecords, with a summary of the number
171 *                             accepted as well as rejected.
172 *                      The report is written to stderr by default, or the
173 *                      stream set by wcsprintf_set().
174 *
175 *                      For ctrl < 0, WCS keyrecords processed by wcspih()
176 *                      are removed from header[]:
177 *                         -1: Remove only valid WCS keyrecords whose values
178 *                             were successfully extracted, nothing is
179 *                             reported.
180 *                         -2: As above, but also remove WCS keyrecords that
181 *                             were rejected, reporting each one and the
182 *                             reason that it was rejected.
183 *                         -3: As above, and also report the number of
184 *                             coordinate representations (nwcs) found.
185 *                        -11: Same as -1 but preserving global WCS-related
186 *                             keywords such as '{DATE,MJD}-{OBS,BEG,AVG,END}'
187 *                             and the other basic time-related keywords, and
188 *                             'OBSGEO-{X,Y,Z,L,B,H}'.
189 *                      If any keyrecords are removed from header[] it will
190 *                      be null-terminated (NUL not being a legal FITS header
191 *                      character), otherwise it will contain its original
192 *                      complement of nkeyrec keyrecords and possibly not be
193 *                      null-terminated.
194 *
195 * Returned:
196 *   nreject   int*     Number of WCS keywords rejected for syntax errors,
197 *                      illegal values, etc.  Keywords not recognized as WCS
198 *                      keywords are simply ignored.  Refer also to wcsbth()
199 *                      note 5.
200 *
201 *   nwcs      int*     Number of coordinate representations found.
202 *
203 *   wcs       struct wcsprm**
204 *                      Pointer to an array of wcsprm structs containing up to
205 *                      27 coordinate representations.
```

```
206 *
207 *                              Memory for the array is allocated by wcspih() which
208 *                              also invokes wcsini() for each struct to allocate
209 *                              memory for internal arrays and initialize their
210 *                              members to default values.  Refer also to wcsbth()
211 *                              note 8.  Note that wcsset() is not invoked on these
212 *                              structs.
213 *
214 *                              This allocated memory must be freed by the user, first
215 *                              by invoking wcsfree() for each struct, and then by
216 *                              freeing the array itself.  A routine, wcsvfree(), is
217 *                              provided to do this (see below).
218 *
219 * Function return value:
220 *              int        Status return value:
221 *                            0: Success.
222 *                            1: Null wcsprm pointer passed.
223 *                            2: Memory allocation failed.
224 *                            4: Fatal error returned by Flex parser.
225 *
226 * Notes:
227 *   1: Refer to wcsbth() notes 1, 2, 3, 5, 7, and 8.
228 *
229 *
230 * wcsbth() - FITS WCS parser routine for binary table and image headers
231 * ---------------------------------------------------------------
232 * wcsbth() is a high-level FITS WCS routine that parses a binary table header.
233 * It handles image array and pixel list WCS keywords which may be present
234 * together in one header.
235 *
236 * As an extension of the FITS WCS standard, wcsbth() also recognizes image
237 * header keywords in a binary table header.  These may be used to provide
238 * default values via an inheritance mechanism discussed in note 5 (c.f.
239 * WCSHDR_AUXIMG and WCSHDR_ALLIMG), or may instead result in wcsprm structs
240 * that are not associated with any particular column.  Thus wcsbth() can
241 * handle primary image and image extension headers in addition to binary table
242 * headers (it ignores NAXIS and does not rely on the presence of the TFIELDS
243 * keyword).
244 *
245 * All WCS keywords defined in Papers I, II, III, and VII are recognized, and
246 * also those used by the AIPS convention and certain other keywords that
247 * existed in early drafts of the WCS papers as explained in note 5 below.
248 *
249 * wcsbth() sets the colnum or colax[] members of the wcsprm structs that it
250 * returns with the column number of an image array or the column numbers
251 * associated with each pixel coordinate element in a pixel list.  wcsprm
252 * structs that are not associated with any particular column, as may be
253 * derived from image header keywords, have colnum == 0.
254 *
255 * Note 6 below discusses the number of wcsprm structs returned by wcsbth(),
256 * and the circumstances in which image header keywords cause a struct to be
257 * created.  See also note 9 concerning the number of separate images that may
258 * be stored in a pixel list.
259 *
260 * The API to wcsbth() is similar to that of wcspih() except for the addition
261 * of extra arguments that may be used to restrict its operation.  Like
262 * wcspih(), wcsbth() invokes wcstab() on each of the wcsprm structs that it
263 * returns.
264 *
265 * Given and returned:
266 *    header     char[]     Character array containing the (entire) FITS binary
267 *                          table, primary image, or image extension header from
268 *                          which to identify and construct the coordinate
269 *                          representations, for example, as might be obtained
270 *                          conveniently via the CFITSIO routine fits_hdr2str().
271 *
272 *                          Each header "keyrecord" (formerly "card image")
273 *                          consists of exactly 80 7-bit ASCII printing
274 *                          characters in the range 0x20 to 0x7e (which excludes
275 *                          NUL, BS, TAB, LF, FF and CR) especially noting that
276 *                          the keyrecords are NOT null-terminated.
277 *
278 *                          For negative values of ctrl (see below), header[] is
279 *                          modified so that WCS keyrecords processed by wcsbth()
280 *                          are removed from it.
281 *
282 * Given:
283 *    nkeyrec    int        Number of keyrecords in header[].
284 *
285 *    relax      int        Degree of permissiveness:
286 *                            0: Recognize only FITS keywords defined by the
287 *                               published WCS standard.
288 *                            WCSHDR_all: Admit all recognized informal
289 *                               extensions of the WCS standard.
290 *                          Fine-grained control of the degree of permissiveness
291 *                          is also possible, as explained in note 5 below.
292 *
```

```
293 *   ctrl      int         Error reporting and other control options for invalid
294 *                         WCS and other header keyrecords:
295 *                           0: Do not report any rejected header keyrecords.
296 *                           1: Produce a one-line message stating the number
297 *                              of WCS keyrecords rejected (nreject).
298 *                           2: Report each rejected keyrecord and the reason
299 *                              why it was rejected.
300 *                           3: As above, but also report all non-WCS
301 *                              keyrecords that were discarded, and the number
302 *                              of coordinate representations (nwcs) found.
303 *                           4: As above, but also report the accepted WCS
304 *                              keyrecords, with a summary of the number
305 *                              accepted as well as rejected.
306 *                         The report is written to stderr by default, or the
307 *                         stream set by wcsprintf_set().
308 *
309 *                         For ctrl < 0, WCS keyrecords processed by wcsbth()
310 *                         are removed from header[]:
311 *                          -1: Remove only valid WCS keyrecords whose values
312 *                              were successfully extracted, nothing is
313 *                              reported.
314 *                          -2: Also remove WCS keyrecords that were rejected,
315 *                              reporting each one and the reason that it was
316 *                              rejected.
317 *                          -3: As above, and also report the number of
318 *                              coordinate representations (nwcs) found.
319 *                         -11: Same as -1 but preserving global WCS-related
320 *                              keywords such as '{DATE,MJD}-{OBS,BEG,AVG,END}'
321 *                              and the other basic time-related keywords, and
322 *                              'OBSGEO-{X,Y,Z,L,B,H}'.
323 *                         If any keyrecords are removed from header[] it will
324 *                         be null-terminated (NUL not being a legal FITS header
325 *                         character), otherwise it will contain its original
326 *                         complement of nkeyrec keyrecords and possibly not be
327 *                         null-terminated.
328 *
329 *   keysel    int         Vector of flag bits that may be used to restrict the
330 *                         keyword types considered:
331 *                           WCSHDR_IMGHEAD: Image header keywords.
332 *                           WCSHDR_BIMGARR: Binary table image array.
333 *                           WCSHDR_PIXLIST: Pixel list keywords.
334 *                         If zero, there is no restriction.
335 *
336 *                         Keywords such as EQUIna or RFRQna that are common to
337 *                         binary table image arrays and pixel lists (including
338 *                         WCSNna and TWCSna, as explained in note 4 below) are
339 *                         selected by both WCSHDR_BIMGARR and WCSHDR_PIXLIST.
340 *                         Thus if inheritance via WCSHDR_ALLIMG is enabled as
341 *                         discussed in note 5 and one of these shared keywords
342 *                         is present, then WCSHDR_IMGHEAD and WCSHDR_PIXLIST
343 *                         alone may be sufficient to cause the construction of
344 *                         coordinate descriptions for binary table image arrays.
345 *
346 *   colsel    int*        Pointer to an array of table column numbers used to
347 *                         restrict the keywords considered by wcsbth().
348 *
349 *                         A null pointer may be specified to indicate that there
350 *                         is no restriction.  Otherwise, the magnitude of
351 *                         cols[0] specifies the length of the array:
352 *                           cols[0] > 0: the columns are included,
353 *                           cols[0] < 0: the columns are excluded.
354 *
355 *                         For the pixel list keywords TPn_ka and TCn_ka (and
356 *                         TPCn_ka and TCDn_ka if WCSHDR_LONGKEY is enabled), it
357 *                         is an error for one column to be selected but not the
358 *                         other.  This is unlike the situation with invalid
359 *                         keyrecords, which are simply rejected, because the
360 *                         error is not intrinsic to the header itself but
361 *                         arises in the way that it is processed.
362 *
363 * Returned:
364 *   nreject   int*        Number of WCS keywords rejected for syntax errors,
365 *                         illegal values, etc.  Keywords not recognized as WCS
366 *                         keywords are simply ignored, refer also to note 5
367 *                         below.
368 *
369 *   nwcs      int*        Number of coordinate representations found.
370 *
371 *   wcs       struct wcsprm**
372 *                         Pointer to an array of wcsprm structs containing up
373 *                         to 27027 coordinate representations, refer to note 6
374 *                         below.
375 *
376 *                         Memory for the array is allocated by wcsbth() which
377 *                         also invokes wcsini() for each struct to allocate
378 *                         memory for internal arrays and initialize their
379 *                         members to default values.  Refer also to note 8
```

```
380 *                          below.  Note that wcsset() is not invoked on these
381 *                          structs.
382 *
383 *                          This allocated memory must be freed by the user, first
384 *                          by invoking wcsfree() for each struct, and then by
385 *                          freeing the array itself.  A routine, wcsvfree(), is
386 *                          provided to do this (see below).
387 *
388 * Function return value:
389 *              int         Status return value:
390 *                            0: Success.
391 *                            1: Null wcsprm pointer passed.
392 *                            2: Memory allocation failed.
393 *                            3: Invalid column selection.
394 *                            4: Fatal error returned by Flex parser.
395 *
396 * Notes:
397 *   1: wcspih() determines the number of coordinate axes independently for
398 *      each alternate coordinate representation (denoted by the "a" value in
399 *      keywords like CTYPEia) from the higher of
400 *
401 *        a: NAXIS,
402 *        b: WCSAXESa,
403 *        c: The highest axis number in any parameterized WCS keyword.  The
404 *           keyvalue, as well as the keyword, must be syntactically valid
405 *           otherwise it will not be considered.
406 *
407 *      If none of these keyword types is present, i.e. if the header only
408 *      contains auxiliary WCS keywords for a particular coordinate
409 *      representation, then no coordinate description is constructed for it.
410 *
411 *      wcsbth() is similar except that it ignores the NAXIS keyword if given
412 *      an image header to process.
413 *
414 *      The number of axes, which is returned as a member of the wcsprm
415 *      struct, may differ for different coordinate representations of the
416 *      same image.
417 *
418 *   2: wcspih() and wcsbth() enforce correct FITS "keyword = value" syntax
419 *      with regard to "= " occurring in columns 9 and 10.
420 *
421 *      However, they do recognize free-format character (NOST 100-2.0,
422 *      Sect. 5.2.1), integer (Sect. 5.2.3), and floating-point values
423 *      (Sect. 5.2.4) for all keywords.
424 *
425 *   3: Where CROTAn, CDi_ja, and PCi_ja occur together in one header wcspih()
426 *      and wcsbth() treat them as described in the prologue to wcs.h.
427 *
428 *   4: WCS Paper I mistakenly defined the pixel list form of WCSNAMEa as
429 *      TWCSna instead of WCSNna; the 'T' is meant to substitute for the axis
430 *      number in the binary table form of the keyword – note that keywords
431 *      defined in WCS Papers II, III, and VII that are not parameterized by
432 *      axis number have identical forms for binary tables and pixel lists.
433 *      Consequently wcsbth() always treats WCSNna and TWCSna as equivalent.
434 *
435 *   5: wcspih() and wcsbth() interpret the "relax" argument as a vector of
436 *      flag bits to provide fine-grained control over what non-standard WCS
437 *      keywords to accept.  The flag bits are subject to change in future and
438 *      should be set by using the preprocessor macros (see below) for the
439 *      purpose.
440 *
441 *      – WCSHDR_none: Don't accept any extensions (not even those in the
442 *               errata).  Treat non-conformant keywords in the same way as
443 *               non-WCS keywords in the header, i.e. simply ignore them.
444 *
445 *      – WCSHDR_all: Accept all extensions recognized by the parser.
446 *
447 *      – WCSHDR_reject: Reject non-standard keyrecords (that are not otherwise
448 *               explicitly accepted by one of the flags below).  A message will
449 *               optionally be printed on stderr by default, or the stream set
450 *               by wcsprintf_set(), as determined by the ctrl argument, and
451 *               nreject will be incremented.
452 *
453 *               This flag may be used to signal the presence of non-standard
454 *               keywords, otherwise they are simply passed over as though they
455 *               did not exist in the header.  It is mainly intended for testing
456 *               conformance of a FITS header to the WCS standard.
457 *
458 *               Keyrecords may be non-standard in several ways:
459 *
460 *                  – The keyword may be syntactically valid but with keyvalue of
461 *                    incorrect type or invalid syntax, or the keycomment may be
462 *                    malformed.
463 *
464 *                  – The keyword may strongly resemble a WCS keyword but not, in
465 *                    fact, be one because it does not conform to the standard.
466 *                    For example, "CRPIX01" looks like a CRPIXja keyword, but in
```

```
467 *                      fact the leading zero on the axis number violates the basic
468 *                      FITS standard.  Likewise, "LONPOLE2" is not a valid
469 *                      LONPOLEa keyword in the WCS standard, and indeed there is
470 *                      nothing the parser can sensibly do with it.
471 *
472 *                    - Use of the keyword may be deprecated by the standard.  Such
473 *                      will be rejected if not explicitly accepted via one of the
474 *                      flags below.
475 *
476 *           - WCSHDR_strict: As for WCSHDR_reject, but also reject AIPS-convention
477 *                    keywords and all other deprecated usage that is not explicitly
478 *                    accepted.
479 *
480 *           - WCSHDR_CROTAia: Accept CROTAia (wcspih()),
481 *                              iCROTna (wcsbth()),
482 *                              TCROTna (wcsbth()).
483 *           - WCSHDR_VELREFa: Accept VELREFa.
484 *                    wcspih() always recognizes the AIPS-convention keywords,
485 *                    CROTAn, EPOCH, and VELREF for the primary representation
486 *                    (a = ' ') but alternates are non-standard.
487 *
488 *                    wcsbth() accepts EPOCHa and VELREFa only if WCSHDR_AUXIMG is
489 *                    also enabled.
490 *
491 *           - WCSHDR_CD00i00j: Accept CD00i00j (wcspih()).
492 *           - WCSHDR_PC00i00j: Accept PC00i00j (wcspih()).
493 *           - WCSHDR_PROJPn:   Accept PROJPn   (wcspih()).
494 *                    These appeared in early drafts of WCS Paper I+II (before they
495 *                    were split) and are equivalent to CDi_ja, PCi_ja, and PVi_ma
496 *                    for the primary representation (a = ' ').  PROJPn is
497 *                    equivalent to PVi_ma with m = n <= 9, and is associated
498 *                    exclusively with the latitude axis.
499 *
500 *           - WCSHDR_CD0i_0ja: Accept CD0i_0ja (wcspih()).
501 *           - WCSHDR_PC0i_0ja: Accept PC0i_0ja (wcspih()).
502 *           - WCSHDR_PV0i_0ma: Accept PV0i_0ja (wcspih()).
503 *           - WCSHDR_PS0i_0ma: Accept PS0i_0ja (wcspih()).
504 *                    Allow the numerical index to have a leading zero in doubly-
505 *                    parameterized keywords, for example, PC01_01.  WCS Paper I
506 *                    (Sects 2.1.2 & 2.1.4) explicitly disallows leading zeroes.
507 *                    The FITS 3.0 standard document (Sect. 4.1.2.1) states that the
508 *                    index in singly-parameterized keywords (e.g. CTYPEia) "shall
509 *                    not have leading zeroes", and later in Sect. 8.1 that "leading
510 *                    zeroes must not be used" on PVi_ma and PSi_ma.  However, by an
511 *                    oversight, it is silent on PCi_ja and CDi_ja.
512 *
513 *           - WCSHDR_DOBSn (wcsbth() only): Allow DOBSn, the column-specific
514 *                    analogue of DATE-OBS.  By an oversight this was never formally
515 *                    defined in the standard.
516 *
517 *           - WCSHDR_OBSGLBHn (wcsbth() only): Allow OBSGLn, OBSGBn, and OBSGHn,
518 *                    the column-specific analogues of OBSGEO-L, OBSGEO-B, and
519 *                    OBSGEO-H.  By an oversight these were never formally defined in
520 *                    the standard.
521 *
522 *           - WCSHDR_RADECSYS: Accept RADECSYS.  This appeared in early drafts of
523 *                    WCS Paper I+II and was subsequently replaced by RADESYSa.
524 *
525 *                    wcsbth() accepts RADECSYS only if WCSHDR_AUXIMG is also
526 *                    enabled.
527 *
528 *           - WCSHDR_EPOCHa:  Accept EPOCHa.
529 *
530 *           - WCSHDR_VSOURCE: Accept VSOURCEa or VSOUna (wcsbth()).  This appeared
531 *                    in early drafts of WCS Paper III and was subsequently dropped
532 *                    in favour of ZSOURCEa and ZSOUna.
533 *
534 *                    wcsbth() accepts VSOURCEa only if WCSHDR_AUXIMG is also
535 *                    enabled.
536 *
537 *           - WCSHDR_DATEREF: Accept DATE-REF, MJD-REF, MJD-REFI, MJD-REFF, JDREF,
538 *                    JD-REFI, and JD-REFF as synonyms for the standard keywords,
539 *                    DATEREF, MJDREF, MJDREFI, MJDREFF, JDREF, and JDREFF.
540 *                    The latter buck the pattern set by the other date keywords
541 *                    ({DATE,MJD}-{OBS,BEG,AVG,END}), thereby increasing the
542 *                    potential for confusion and error.
543 *
544 *           - WCSHDR_LONGKEY (wcsbth() only): Accept long forms of the alternate
545 *                    binary table and pixel list WCS keywords, i.e. with "a" non-
546 *                    blank.  Specifically
547 *
548 #                    jCRPXna  TCRPXna  :  jCRPXn  jCRPna  TCRPXn  TCRPna  CRPIXja
549 #                       -     TPCn_ka  :    -     ijPCna    -     TPn_ka  PCi_ja
550 #                       -     TCDn_ka  :    -     ijCDna    -     TCn_ka  CDi_ja
551 #                    iCDLTna  TCDLTna  :  iCDLTn  iCDEna  TCDLTn  TCDEna  CDELTia
552 #                    iCUNIna  TCUNIna  :  iCUNIn  iCUNna  TCUNIn  TCUNna  CUNITia
553 #                    iCTYPna  TCTYPna  :  iCTYPn  iCTYna  TCTYPn  TCTYna  CTYPEia
```

```
554 #               iCRVLna  TCRVLna  :   iCRVLn  iCRVna  TCRVLn  TCRVna  CRVALia
555 #               iPVn_ma  TPVn_ma  :    -      iVn_ma    -     TVn_ma  PVi_ma
556 #               iPSn_ma  TPSn_ma  :    -      iSn_ma    -     TSn_ma  PSi_ma
557 *
558 *             where the primary and standard alternate forms together with
559 *             the image-header equivalent are shown rightwards of the colon.
560 *
561 *             The long form of these keywords could be described as quasi-
562 *             standard.  TPCn_ka, iPVn_ma, and TPVn_ma appeared by mistake
563 *             in the examples in WCS Paper II and subsequently these and
564 *             also TCDn_ka, iPSn_ma and TPSn_ma were legitimized by the
565 *             errata to the WCS papers.
566 *
567 *             Strictly speaking, the other long forms are non-standard and
568 *             in fact have never appeared in any draft of the WCS papers nor
569 *             in the errata.  However, as natural extensions of the primary
570 *             form they are unlikely to be written with any other intention.
571 *             Thus it should be safe to accept them provided, of course,
572 *             that the resulting keyword does not exceed the 8-character
573 *             limit.
574 *
575 *             If WCSHDR_CNAMn is enabled then also accept
576 *
577 #               iCNAMna  TCNAMna  :   ---   iCNAna   ---   TCNAna  CNAMEia
578 #               iCRDEna  TCRDEna  :   ---   iCRDna   ---   TCRDna  CRDERia
579 #               iCSYEna  TCSYEna  :   ---   iCSYna   ---   TCSYna  CSYERia
580 #               iCZPHna  TCZPHna  :   ---   iCZPna   ---   TCZPna  CZPHSia
581 #               iCPERna  TCPERna  :   ---   iCPRna   ---   TCPRna  CPERIia
582 *
583 *             Note that CNAMEia, CRDERia, CSYERia, CZPHSia, CPERIia, and
584 *             their variants are not used by WCSLIB but are stored in the
585 *             wcsprm struct as auxiliary information.
586 *
587 *       - WCSHDR_CNAMn (wcsbth() only): Accept iCNAMn, iCRDEn, iCSYEn, iCZPHn,
588 *             iCPERn, TCNAMn, TCRDEn, TCSYEn, TCZPHn, and TCPERn, i.e. with
589 *             "a" blank.  While non-standard, these are the obvious analogues
590 *             of iCTYPn, TCTYPn, etc.
591 *
592 *       - WCSHDR_AUXIMG (wcsbth() only): Allow the image-header form of an
593 *             auxiliary WCS keyword with representation-wide scope to
594 *             provide a default value for all images.  This default may be
595 *             overridden by the column-specific form of the keyword.
596 *
597 *             For example, a keyword like EQUINOXa would apply to all image
598 *             arrays in a binary table, or all pixel list columns with
599 *             alternate representation "a" unless overridden by EQUIna.
600 *
601 *             Specifically the keywords are:
602 *
603 #               LONPOLEa  for LONPna
604 #               LATPOLEa  for LATPna
605 #               VELREF       -       ... (No column-specific form.)
606 #               VELREFa      -       ... Only if WCSHDR_VELREFa is set.
607 *
608 *             whose keyvalues are actually used by WCSLIB, and also keywords
609 *             providing auxiliary information that is simply stored in the
610 *             wcsprm struct:
611 *
612 #               WCSNAMEa  for WCSNna  ... Or TWCSna (see below).
613 #
614 #               DATE-OBS  for DOBSn
615 #               MJD-OBS   for MJDOBn
616 #
617 #               RADESYSa  for RADEna
618 #               RADECSYS  for RADEna  ... Only if WCSHDR_RADECSYS is set.
619 #               EPOCH        -        ... (No column-specific form.)
620 #               EPOCHa       -        ... Only if WCSHDR_EPOCHa is set.
621 #               EQUINOXa  for EQUIna
622 *
623 *             where the image-header keywords on the left provide default
624 *             values for the column specific keywords on the right.
625 *
626 *             Note that, according to Sect. 8.1 of WCS Paper III, and
627 *             Sect. 5.2 of WCS Paper VII, the following are always inherited:
628 *
629 #               RESTFREQ  for RFRQna
630 #               RESTFRQa  for RFRQna
631 #               RESTWAVa  for RWAVna
632 *
633 *             being those actually used by WCSLIB, together with the
634 *             following auxiliary keywords, many of which do not have binary
635 *             table equivalents and therefore can only be inherited:
636 *
637 #               TIMESYS       -
638 #               TREFPOS   for TRPOSn
639 #               TREFDIR   for TRDIRn
640 #               PLEPHEM       -
```

```
641 #               TIMEUNIT    -
642 #               DATEREF     -
643 #               MJDREF      -
644 #               MJDREFI     -
645 #               MJDREFF     -
646 #               JDREF       -
647 #               JDREFI      -
648 #               JDREFF      -
649 #               TIMEOFFS    -
650 #
651 #               DATE-BEG    -
652 #               DATE-AVG  for DAVGn
653 #               DATE-END    -
654 #               MJD-BEG     -
655 #               MJD-AVG   for MJDAn
656 #               MJD-END     -
657 #               JEPOCH      -
658 #               BEPOCH      -
659 #               TSTART      -
660 #               TSTOP       -
661 #               XPOSURE     -
662 #               TELAPSE     -
663 #
664 #               TIMSYER     -
665 #               TIMRDER     -
666 #               TIMEDEL     -
667 #               TIMEPIXR    -
668 #
669 #               OBSGEO-X  for OBSGXn
670 #               OBSGEO-Y  for OBSGYn
671 #               OBSGEO-Z  for OBSGZn
672 #               OBSGEO-L  for OBSGLn
673 #               OBSGEO-B  for OBSGBn
674 #               OBSGEO-H  for OBSGHn
675 #               OBSORBIT    -
676 #
677 #               SPECSYSa  for SPECna
678 #               SSYSOBSa  for SOBSna
679 #               VELOSYSa  for VSYSna
680 #               VSOURCEa  for VSOUna  ... Only if WCSHDR_VSOURCE is set.
681 #               ZSOURCEa  for ZSOUna
682 #               SSYSSRCa  for SSRCna
683 #               VELANGLa  for VANGna
684 *
685 *           Global image-header keywords, such as MJD-OBS, apply to all
686 *           alternate representations, and would therefore provide a
687 *           default value for all images in the header.
688 *
689 *           This auxiliary inheritance mechanism applies to binary table
690 *           image arrays and pixel lists alike.  Most of these keywords
691 *           have no default value, the exceptions being LONPOLEa and
692 *           LATPOLEa, and also RADESYSa and EQUINOXa which provide
693 *           defaults for each other.  Thus one potential difficulty in
694 *           using WCSHDR_AUXIMG is that of erroneously inheriting one of
695 *           these four keywords.
696 *
697 *           Also, beware of potential inconsistencies that may arise where,
698 *           for example, DATE-OBS is inherited, but MJD-OBS is overridden
699 *           by MJDOBn and specifies a different time.  Pairs in this
700 *           category are:
701 *
702 =               DATE-OBS/DOBSn       versus      MJD-OBS/MJDOBn
703 =               DATE-AVG/DAVGn       versus      MJD-AVG/MJDAn
704 =               RESTFRQa/RFRQna      versus      RESTWAVa/RWAVna
705 =           OBSGEO-[XYZ]/OBSG[XYZ]n  versus  OBSGEO-[LBH]/OBSG[LBH]n
706 *
707 *           The wcsfixi() routines datfix() and obsfix() are provided to
708 *           check the consistency of these and other such pairs of
709 *           keywords.
710 *
711 *           Unlike WCSHDR_ALLIMG, the existence of one (or all) of these
712 *           auxiliary WCS image header keywords will not by itself cause a
713 *           wcsprm struct to be created for alternate representation "a".
714 *           This is because they do not provide sufficient information to
715 *           create a non-trivial coordinate representation when used in
716 *           conjunction with the default values of those keywords that are
717 *           parameterized by axis number, such as CTYPEia.
718 *
719 *       - WCSHDR_ALLIMG (wcsbth() only): Allow the image-header form of *all*
720 *           image header WCS keywords to provide a default value for all
721 *           image arrays in a binary table (n.b. not pixel list).  This
722 *           default may be overridden by the column-specific form of the
723 *           keyword.
724 *
725 *           For example, a keyword like CRPIXja would apply to all image
726 *           arrays in a binary table with alternate representation "a"
727 *           unless overridden by jCRPna.
```

```
728 *
729 *              Specifically the keywords are those listed above for
730 *              WCSHDR_AUXIMG plus
731 *
732 #                WCSAXESa  for WCAXna
733 *
734 *              which defines the coordinate dimensionality, and the following
735 *              keywords that are parameterized by axis number:
736 *
737 #                CRPIXja   for jCRPna
738 #                PCi_ja    for ijPCna
739 #                CDi_ja    for ijCDna
740 #                CDELTia   for iCDEna
741 #                CROTAi    for iCROTn
742 #                CROTAia     -        ... Only if WCSHDR_CROTAia is set.
743 #                CUNITia   for iCUNna
744 #                CTYPEia   for iCTYna
745 #                CRVALia   for iCRVna
746 #                PVi_ma    for iVn_ma
747 #                PSi_ma    for iSn_ma
748 #
749 #                CNAMEia   for iCNAna
750 #                CRDERia   for iCRDna
751 #                CSYERia   for iCSYna
752 #                CZPHSia   for iCZPna
753 #                CPERIia   for iCPRna
754 *
755 *              where the image-header keywords on the left provide default
756 *              values for the column specific keywords on the right.
757 *
758 *              This full inheritance mechanism only applies to binary table
759 *              image arrays, not pixel lists, because in the latter case
760 *              there is no well-defined association between coordinate axis
761 *              number and column number (see note 9 below).
762 *
763 *              Note that CNAMEia, CRDERia, CSYERia, and their variants are
764 *              not used by WCSLIB but are stored in the wcsprm struct as
765 *              auxiliary information.
766 *
767 *              Note especially that at least one wcsprm struct will be
768 *              returned for each "a" found in one of the image header
769 *              keywords listed above:
770 *
771 *              - If the image header keywords for "a" ARE NOT inherited by a
772 *                binary table, then the struct will not be associated with
773 *                any particular table column number and it is up to the user
774 *                to provide an association.
775 *
776 *              - If the image header keywords for "a" ARE inherited by a
777 *                binary table image array, then those keywords are considered
778 *                to be "exhausted" and do not result in a separate wcsprm
779 *                struct.
780 *
781 *        For example, to accept CD00i00j and PC00i00j and reject all other
782 *        extensions, use
783 *
784 =            relax = WCSHDR_reject | WCSHDR_CD00i00j | WCSHDR_PC00i00j;
785 *
786 *        The parser always treats EPOCH as subordinate to EQUINOXa if both are
787 *        present, and VSOURCEa is always subordinate to ZSOURCEa.
788 *
789 *        Likewise, VELREF is subordinate to the formalism of WCS Paper III, see
790 *        spcaips().
791 *
792 *        Neither wcspih() nor wcsbth() currently recognize the AIPS-convention
793 *        keywords ALTRPIX or ALTRVAL which effectively define an alternative
794 *        representation for a spectral axis.
795 *
796 *   6: Depending on what flags have been set in its "relax" argument,
797 *      wcsbth() could return as many as 27027 wcsprm structs:
798 *
799 *      - Up to 27 unattached representations derived from image header
800 *        keywords.
801 *
802 *      - Up to 27 structs for each of up to 999 columns containing an image
803 *        arrays.
804 *
805 *      - Up to 27 structs for a pixel list.
806 *
807 *      Note that it is considered legitimate for a column to contain an image
808 *      array and also form part of a pixel list, and in particular that
809 *      wcsbth() does not check the TFORM keyword for a pixel list column to
810 *      check that it is scalar.
811 *
812 *      In practice, of course, a realistic binary table header is unlikely to
813 *      contain more than a handful of images.
814 *
```

```
815 *      In order for wcsbth() to create a wcsprm struct for a particular
816 *      coordinate representation, at least one WCS keyword that defines an
817 *      axis number must be present, either directly or by inheritance if
818 *      WCSHDR_ALLIMG is set.
819 *
820 *      When the image header keywords for an alternate representation are
821 *      inherited by a binary table image array via WCSHDR_ALLIMG, those
822 *      keywords are considered to be "exhausted" and do not result in a
823 *      separate wcsprm struct.  Otherwise they do.
824 *
825 *   7: Neither wcspih() nor wcsbth() check for duplicated keywords, in most
826 *      cases they accept the last encountered.
827 *
828 *   8: wcspih() and wcsbth() use wcsnpv() and wcsnps() (refer to the prologue
829 *      of wcs.h) to match the size of the pv[] and ps[] arrays in the wcsprm
830 *      structs to the number in the header.  Consequently there are no unused
831 *      elements in the pv[] and ps[] arrays, indeed they will often be of
832 *      zero length.
833 *
834 *   9: The FITS WCS standard for pixel lists assumes that a pixel list
835 *      defines one and only one image, i.e. that each row of the binary table
836 *      refers to just one event, e.g. the detection of a single photon or
837 *      neutrino, for which the device "pixel" coordinates are stored in
838 *      separate scalar columns of the table.
839 *
840 *      In the absence of a standard for pixel lists – or even an informal
841 *      description! – let alone a formal mechanism for identifying the columns
842 *      containing pixel coordinates (as opposed to pixel values or metadata
843 *      recorded at the time the photon or neutrino was detected), WCS Paper I
844 *      discusses how the WCS keywords themselves may be used to identify them.
845 *
846 *      In practice, however, pixel lists have been used to store multiple
847 *      images.  Besides not specifying how to identify columns, the pixel list
848 *      convention is also silent on the method to be used to associate table
849 *      columns with image axes.
850 *
851 *      An additional shortcoming is the absence of a formal method for
852 *      associating global binary-table WCS keywords, such as WCSNna or MJDOBn,
853 *      with a pixel list image, whether one or several.
854 *
855 *      In light of these uncertainties, wcsbth() simply collects all WCS
856 *      keywords for a particular pixel list coordinate representation (i.e.
857 *      the "a" value in TCTYna) into one wcsprm struct.  However, these
858 *      alternates need not be associated with the same table columns and this
859 *      allows a pixel list to contain up to 27 separate images.  As usual, if
860 *      one of these representations happened to contain more than two
861 *      celestial axes, for example, then an error would result when wcsset()
862 *      is invoked on it.  In this case the "colsel" argument could be used to
863 *      restrict the columns used to construct the representation so that it
864 *      only contained one pair of celestial axes.
865 *
866 *      Global, binary-table WCS keywords are considered to apply to the pixel
867 *      list image with matching alternate (e.g. the "a" value in LONPna or
868 *      EQUIna), regardless of the table columns the image occupies.  In other
869 *      words, the column number is ignored (the "n" value in LONPna or
870 *      EQUIna).  This also applies for global, binary-table WCS keywords that
871 *      have no alternates, such as MJDOBn and OBSGXn, which match all images
872 *      in a pixel list.  Take heed that this may lead to counterintuitive
873 *      behaviour, especially where such a keyword references a column that
874 *      does not store pixel coordinates, and moreso where the pixel list
875 *      stores only a single image.  In fact, as the column number, n, is
876 *      ignored for such keywords, it would make no difference even if they
877 *      referenced non-existent columns.  Moreover, there is no requirement for
878 *      consistency in the column numbers used for such keywords, even for
879 *      OBSGXn, OBSGYn, and OBSGZn which are meant to define the elements of a
880 *      coordinate vector.  Although it would surely be perverse to construct a
881 *      pixel list like this, such a situation may still arise in practice
882 *      where columns are deleted from a binary table.
883 *
884 *      The situation with global, binary-table WCS keywords becomes
885 *      potentially even more confusing when image arrays and pixel list images
886 *      coexist in one binary table.  In that case, a keyword such as MJDOBn
887 *      may legitimately appear multiple times with n referencing different
888 *      image arrays.  Which then is the one that applies to the pixel list
889 *      images?  In this implementation, it is the last instance that appears
890 *      in the header, whether or not it is also associated with an image
891 *      array.
892 *
893 *
894 * wcstab() – Tabular construction routine
895 * ---------------------------------------
896 * wcstab() assists in filling in the information in the wcsprm struct relating
897 * to coordinate lookup tables.
898 *
899 * Tabular coordinates ('TAB') present certain difficulties in that the main
900 * components of the lookup table – the multidimensional coordinate array plus
901 * an index vector for each dimension – are stored in a FITS binary table
```

```
902 * extension (BINTABLE).  Information required to locate these arrays is stored
903 * in PVi_ma and PSi_ma keywords in the image header.
904 *
905 * wcstab() parses the PVi_ma and PSi_ma keywords associated with each 'TAB'
906 * axis and allocates memory in the wcsprm struct for the required number of
907 * tabprm structs.  It sets as much of the tabprm struct as can be gleaned from
908 * the image header, and also sets up an array of wtbarr structs (described in
909 * the prologue of wtbarr.h) to assist in extracting the required arrays from
910 * the BINTABLE extension(s).
911 *
912 * It is then up to the user to allocate memory for, and copy arrays from the
913 * BINTABLE extension(s) into the tabprm structs.  A CFITSIO routine,
914 * fits_read_wcstab(), has been provided for this purpose, see getwcstab.h.
915 * wcsset() will automatically take control of this allocated memory, in
916 * particular causing it to be freed by wcsfree(); the user must not attempt
917 * to free it after wcsset() has been called.
918 *
919 * Note that wcspih() and wcsbth() automatically invoke wcstab() on each of the
920 * wcsprm structs that they return.
921 *
922 * Given and returned:
923 *   wcs       struct wcsprm*
924 *                       Coordinate transformation parameters (see below).
925 *
926 *                       wcstab() sets ntab, tab, nwtb and wtb, allocating
927 *                       memory for the tab and wtb arrays.  This allocated
928 *                       memory will be freed automatically by wcsfree().
929 *
930 * Function return value:
931 *             int       Status return value:
932 *                        0: Success.
933 *                        1: Null wcsprm pointer passed.
934 *                        2: Memory allocation failed.
935 *                        3: Invalid tabular parameters.
936 *
937 *                       For returns > 1, a detailed error message is set in
938 *                       wcsprm::err if enabled, see wcserr_enable().
939 *
940 *
941 * wcsidx() - Index alternate coordinate representations
942 * -----------------------------------------------------
943 * wcsidx() returns an array of 27 indices for the alternate coordinate
944 * representations in the array of wcsprm structs returned by wcspih().  For
945 * the array returned by wcsbth() it returns indices for the unattached
946 * (colnum == 0) representations derived from image header keywords - use
947 * wcsbdx() for those derived from binary table image arrays or pixel lists
948 * keywords.
949 *
950 * Given:
951 *   nwcs      int       Number of coordinate representations in the array.
952 *
953 *   wcs       const struct wcsprm**
954 *                       Pointer to an array of wcsprm structs returned by
955 *                       wcspih() or wcsbth().
956 *
957 * Returned:
958 *   alts      int[27]   Index of each alternate coordinate representation in
959 *                       the array: alts[0] for the primary, alts[1] for 'A',
960 *                       etc., set to -1 if not present.
961 *
962 *                       For example, if there was no 'P' representation then
963 *
964 =                         alts['P'-'A'+1] == -1;
965 *
966 *                       Otherwise, the address of its wcsprm struct would be
967 *
968 =                         wcs + alts['P'-'A'+1];
969 *
970 * Function return value:
971 *             int       Status return value:
972 *                        0: Success.
973 *                        1: Null wcsprm pointer passed.
974 *
975 *
976 * wcsbdx() - Index alternate coordinate representions
977 * ---------------------------------------------------
978 * wcsbdx() returns an array of 999 x 27 indices for the alternate coordinate
979 * representions for binary table image arrays xor pixel lists in the array of
980 * wcsprm structs returned by wcsbth().  Use wcsidx() for the unattached
981 * representations derived from image header keywords.
982 *
983 * Given:
984 *   nwcs      int       Number of coordinate representations in the array.
985 *
986 *   wcs       const struct wcsprm**
987 *                       Pointer to an array of wcsprm structs returned by
988 *                       wcsbth().
```

```
989 *
990 *   type      int       Select the type of coordinate representation:
991 *                           0: binary table image arrays,
992 *                           1: pixel lists.
993 *
994 * Returned:
995 *   alts      short[1000][28]
996 *                           Index of each alternate coordinate represention in the
997 *                           array: alts[col][0] for the primary, alts[col][1] for
998 *                           'A', to alts[col][26] for 'Z', where col is the
999 *                           1-relative column number, and col == 0 is used for
1000 *                           unattached image headers.  Set to -1 if not present.
1001 *
1002 *                           alts[col][27] counts the number of coordinate
1003 *                           representations of the chosen type for each column.
1004 *
1005 *                           For example, if there was no 'P' represention for
1006 *                           column 13 then
1007 *
1008 =                               alts[13]['P'-'A'+1] == -1;
1009 *
1010 *                           Otherwise, the address of its wcsprm struct would be
1011 *
1012 =                               wcs + alts[13]['P'-'A'+1];
1013 *
1014 * Function return value:
1015 *             int       Status return value:
1016 *                           0: Success.
1017 *                           1: Null wcsprm pointer passed.
1018 *
1019 *
1020 * wcsvfree() - Free the array of wcsprm structs
1021 * ---------------------------------------------
1022 * wcsvfree() frees the memory allocated by wcspih() or wcsbth() for the array
1023 * of wcsprm structs, first invoking wcsfree() on each of the array members.
1024 *
1025 * Given and returned:
1026 *   nwcs      int*      Number of coordinate representations found; set to 0
1027 *                           on return.
1028 *
1029 *   wcs       struct wcsprm**
1030 *                           Pointer to the array of wcsprm structs; set to 0x0 on
1031 *                           return.
1032 *
1033 * Function return value:
1034 *             int       Status return value:
1035 *                           0: Success.
1036 *                           1: Null wcsprm pointer passed.
1037 *
1038 *
1039 * wcshdo() - Write out a wcsprm struct as a FITS header
1040 * -----------------------------------------------------
1041 * wcshdo() translates a wcsprm struct into a FITS header.  If the colnum
1042 * member of the struct is non-zero then a binary table image array header will
1043 * be produced.  Otherwise, if the colax[] member of the struct is set non-zero
1044 * then a pixel list header will be produced.  Otherwise, a primary image or
1045 * image extension header will be produced.
1046 *
1047 * If the struct was originally constructed from a header, e.g. by wcspih(),
1048 * the output header will almost certainly differ in a number of respects:
1049 *
1050 *   - The output header only contains WCS-related keywords.  In particular, it
1051 *     does not contain syntactically-required keywords such as SIMPLE, NAXIS,
1052 *     BITPIX, or END.
1053 *
1054 *   - Elements of the PCi_ja matrix will be written if and only if they differ
1055 *     from the unit matrix.  Thus, if the matrix is unity then no elements
1056 *     will be written.
1057 *
1058 *   - The redundant keywords MJDREF, JDREF, JDREFI, JDREFF, all of which
1059 *     duplicate MJDREFI + MJDREFF, are never written.  OBSGEO-[LBH] are not
1060 *     written if OBSGEO-[XYZ] are defined.
1061 *
1062 *   - Deprecated (e.g. CROTAn, RESTFREQ, VELREF, RADECSYS, EPOCH, VSOURCEa) or
1063 *     non-standard usage will be translated to standard (this is partially
1064 *     dependent on whether wcsfix() was applied).
1065 *
1066 *   - Additional keywords such as WCSAXESa, CUNITia, LONPOLEa and LATPOLEa may
1067 *     appear.
1068 *
1069 *   - Quantities will be converted to the units used internally, basically SI
1070 *     with the addition of degrees.
1071 *
1072 *   - Floating-point quantities may be given to a different decimal precision.
1073 *
1074 *   - The original keycomments will be lost, although wcshdo() tries hard to
1075 *     write meaningful comments.
```

```
1076 *
1077 *   - Keyword order will almost certainly be changed.
1078 *
1079 * Keywords can be translated between the image array, binary table, and pixel
1080 * lists forms by manipulating the colnum or colax[] members of the wcsprm
1081 * struct.
1082 *
1083 * Given:
1084 *   ctrl      int       Vector of flag bits that controls the degree of
1085 *                       permissiveness in departing from the published WCS
1086 *                       standard, and also controls the formatting of
1087 *                       floating-point keyvalues.  Set it to zero to get the
1088 *                       default behaviour.
1089 *
1090 *                       Flag bits for the degree of permissiveness:
1091 *                         WCSHDO_none: Recognize only FITS keywords defined by
1092 *                             the published WCS standard.
1093 *                         WCSHDO_all: Admit all recognized informal extensions
1094 *                             of the WCS standard.
1095 *                       Fine-grained control of the degree of permissiveness
1096 *                       is also possible as explained in the notes below.
1097 *
1098 *                       As for controlling floating-point formatting, by
1099 *                       default wcshdo() uses "%20.12G" for non-parameterized
1100 *                       keywords such as LONPOLEa, and attempts to make the
1101 *                       header more human-readable by using the same "%f"
1102 *                       format for all values of each of the following
1103 *                       parameterized keywords: CRPIXja, PCi_ja, and CDELTia
1104 *                       (n.b. excluding CRVALia).  Each has the same field
1105 *                       width and precision so that the decimal points line
1106 *                       up.  The precision, allowing for up to 15 significant
1107 *                       digits, is chosen so that there are no excess trailing
1108 *                       zeroes.  A similar formatting scheme applies by
1109 *                       default for distortion function parameters.
1110 *
1111 *                       However, where the values of, for example, CDELTia
1112 *                       differ by many orders of magnitude, the default
1113 *                       formatting scheme may cause unacceptable loss of
1114 *                       precision for the lower-valued keyvalues.  Thus the
1115 *                       default behaviour may be overridden:
1116 *                         WCSHDO_P12: Use "%20.12G" format for all floating-
1117 *                             point keyvalues (12 significant digits).
1118 *                         WCSHDO_P13: Use "%21.13G" format for all floating-
1119 *                             point keyvalues (13 significant digits).
1120 *                         WCSHDO_P14: Use "%22.14G" format for all floating-
1121 *                             point keyvalues (14 significant digits).
1122 *                         WCSHDO_P15: Use "%23.15G" format for all floating-
1123 *                             point keyvalues (15 significant digits).
1124 *                         WCSHDO_P16: Use "%24.16G" format for all floating-
1125 *                             point keyvalues (16 significant digits).
1126 *                         WCSHDO_P17: Use "%25.17G" format for all floating-
1127 *                             point keyvalues (17 significant digits).
1128 *                       If more than one of the above flags are set, the
1129 *                       highest number of significant digits prevails.  In
1130 *                       addition, there is an anciliary flag:
1131 *                         WCSHDO_EFMT: Use "%E" format instead of the default
1132 *                             "%G" format above.
1133 *                       Note that excess trailing zeroes are stripped off the
1134 *                       fractional part with "%G" (which never occurs with
1135 *                       "%E").  Note also that the higher-precision options
1136 *                       eat into the keycomment area.  In this regard,
1137 *                       WCSHDO_P14 causes minimal disruption with "%G" format,
1138 *                       while WCSHDO_P13 is appropriate with "%E".
1139 *
1140 * Given and returned:
1141 *   wcs       struct wcsprm*
1142 *                       Pointer to a wcsprm struct containing coordinate
1143 *                       transformation parameters.  Will be initialized if
1144 *                       necessary.
1145 *
1146 * Returned:
1147 *   nkeyrec   int*      Number of FITS header keyrecords returned in the
1148 *                       "header" array.
1149 *
1150 *   header    char**    Pointer to an array of char holding the header.
1151 *                       Storage for the array is allocated by wcshdo() in
1152 *                       blocks of 2880 bytes (32 x 80-character keyrecords)
1153 *                       and must be freed by the user to avoid memory leaks.
1154 *                       See wcsdealloc().
1155 *
1156 *                       Each keyrecord is 80 characters long and is *NOT*
1157 *                       null-terminated, so the first keyrecord starts at
1158 *                       (*header)[0], the second at (*header)[80], etc.
1159 *
1160 * Function return value:
1161 *             int       Status return value (associated with wcs_errmsg[]):
1162 *                         0: Success.
```

```
1163 *                        1: Null wcsprm pointer passed.
1164 *                        2: Memory allocation failed.
1165 *                        3: Linear transformation matrix is singular.
1166 *                        4: Inconsistent or unrecognized coordinate axis
1167 *                           types.
1168 *                        5: Invalid parameter value.
1169 *                        6: Invalid coordinate transformation parameters.
1170 *                        7: Ill-conditioned coordinate transformation
1171 *                           parameters.
1172 *
1173 *                     For returns > 1, a detailed error message is set in
1174 *                     wcsprm::err if enabled, see wcserr_enable().
1175 *
1176 * Notes:
1177 *   1: wcshdo() interprets the "relax" argument as a vector of flag bits to
1178 *      provide fine-grained control over what non-standard WCS keywords to
1179 *      write.  The flag bits are subject to change in future and should be set
1180 *      by using the preprocessor macros (see below) for the purpose.
1181 *
1182 *      - WCSHDO_none: Don't use any extensions.
1183 *
1184 *      - WCSHDO_all: Write all recognized extensions, equivalent to setting
1185 *            each flag bit.
1186 *
1187 *      - WCSHDO_safe: Write all extensions that are considered to be safe and
1188 *            recommended.
1189 *
1190 *      - WCSHDO_DOBSn: Write DOBSn, the column-specific analogue of DATE-OBS
1191 *            for use in binary tables and pixel lists.  WCS Paper III
1192 *            introduced DATE-AVG and DAVGn but by an oversight DOBSn (the
1193 *            obvious analogy) was never formally defined by the standard.
1194 *            The alternative to using DOBSn is to write DATE-OBS which
1195 *            applies to the whole table.  This usage is considered to be
1196 *            safe and is recommended.
1197 *
1198 *      - WCSHDO_TPCn_ka: WCS Paper I defined
1199 *
1200 *            - TPn_ka and TCn_ka for pixel lists
1201 *
1202 *            but WCS Paper II uses TPCn_ka in one example and subsequently
1203 *            the errata for the WCS papers legitimized the use of
1204 *
1205 *            - TPCn_ka and TCDn_ka for pixel lists
1206 *
1207 *            provided that the keyword does not exceed eight characters.
1208 *            This usage is considered to be safe and is recommended because
1209 *            of the non-mnemonic terseness of the shorter forms.
1210 *
1211 *      - WCSHDO_PVn_ma: WCS Paper I defined
1212 *
1213 *            - iVn_ma and iSn_ma for bintables and
1214 *            - TVn_ma and TSn_ma for pixel lists
1215 *
1216 *            but WCS Paper II uses iPVn_ma and TPVn_ma in the examples and
1217 *            subsequently the errata for the WCS papers legitimized the use
1218 *            of
1219 *
1220 *            - iPVn_ma and iPSn_ma for bintables and
1221 *            - TPVn_ma and TPSn_ma for pixel lists
1222 *
1223 *            provided that the keyword does not exceed eight characters.
1224 *            This usage is considered to be safe and is recommended because
1225 *            of the non-mnemonic terseness of the shorter forms.
1226 *
1227 *      - WCSHDO_CRPXna: For historical reasons WCS Paper I defined
1228 *
1229 *            - jCRPXn, iCDLTn, iCUNIn, iCTYPn, and iCRVLn for bintables and
1230 *            - TCRPXn, TCDLTn, TCUNIn, TCTYPn, and TCRVLn for pixel lists
1231 *
1232 *            for use without an alternate version specifier.  However,
1233 *            because of the eight-character keyword constraint, in order to
1234 *            accommodate column numbers greater than 99 WCS Paper I also
1235 *            defined
1236 *
1237 *            - jCRPna, iCDEna, iCUNna, iCTYna and iCRVna for bintables and
1238 *            - TCRPna, TCDEna, TCUNna, TCTYna and TCRVna for pixel lists
1239 *
1240 *            for use with an alternate version specifier (the "a").  Like
1241 *            the PC, CD, PV, and PS keywords there is an obvious tendency to
1242 *            confuse these two forms for column numbers up to 99.  It is
1243 *            very unlikely that any parser would reject keywords in the
1244 *            first set with a non-blank alternate version specifier so this
1245 *            usage is considered to be safe and is recommended.
1246 *
1247 *      - WCSHDO_CNAMna: WCS Papers I and III defined
1248 *
1249 *            - iCNAna,  iCRDna,  and iCSYna  for bintables and
```

```
1250 *                   - TCNAna,  TCRDna,  and TCSYna  for pixel lists
1251 *
1252 *                   By analogy with the above, the long forms would be
1253 *
1254 *                   - iCNAMna, iCRDEna, and iCSYEna for bintables and
1255 *                   - TCNAMna, TCRDEna, and TCSYEna for pixel lists
1256 *
1257 *                   Note that these keywords provide auxiliary information only,
1258 *                   none of them are needed to compute world coordinates.  This
1259 *                   usage is potentially unsafe and is not recommended at this
1260 *                   time.
1261 *
1262 *         - WCSHDO_WCSNna: In light of wcsbth() note 4, write WCSNna instead of
1263 *                   TWCSna for pixel lists.  While wcsbth() treats WCSNna and
1264 *                   TWCSna as equivalent, other parsers may not.  Consequently,
1265 *                   this usage is potentially unsafe and is not recommended at this
1266 *                   time.
1267 *
1268 *
1269 * Global variable: const char *wcshdr_errmsg[] - Status return messages
1270 * ----------------------------------------------------------------------
1271 * Error messages to match the status value returned from each function.
1272 * Use wcs_errmsg[] for status returns from wcshdo().
1273 *
1274 *===========================================================================*/
1275
1276 #ifndef WCSLIB_WCSHDR
1277 #define WCSLIB_WCSHDR
1278
1279 #include "wcs.h"
1280
1281 #ifdef __cplusplus
1282 extern "C" {
1283 #endif
1284
1285 #define WCSHDR_none      0x00000000
1286 #define WCSHDR_all       0x000FFFFF
1287 #define WCSHDR_reject    0x10000000
1288 #define WCSHDR_strict    0x20000000
1289
1290 #define WCSHDR_CROTAia  0x00000001
1291 #define WCSHDR_VELREFa  0x00000002
1292 #define WCSHDR_CD00i00j 0x00000004
1293 #define WCSHDR_PC00i00j 0x00000008
1294 #define WCSHDR_PROJPn   0x00000010
1295 #define WCSHDR_CD0i_0ja 0x00000020
1296 #define WCSHDR_PC0i_0ja 0x00000040
1297 #define WCSHDR_PV0i_0ma 0x00000080
1298 #define WCSHDR_PS0i_0ma 0x00000100
1299 #define WCSHDR_DOBSn     0x00000200
1300 #define WCSHDR_OBSGLBHn 0x00000400
1301 #define WCSHDR_RADECSYS 0x00000800
1302 #define WCSHDR_EPOCHa    0x00001000
1303 #define WCSHDR_VSOURCE   0x00002000
1304 #define WCSHDR_DATEREF   0x00004000
1305 #define WCSHDR_LONGKEY   0x00008000
1306 #define WCSHDR_CNAMn     0x00010000
1307 #define WCSHDR_AUXIMG    0x00020000
1308 #define WCSHDR_ALLIMG    0x00040000
1309
1310 #define WCSHDR_IMGHEAD  0x00100000
1311 #define WCSHDR_BIMGARR  0x00200000
1312 #define WCSHDR_PIXLIST  0x00400000
1313
1314 #define WCSHDO_none      0x00000
1315 #define WCSHDO_all       0x000FF
1316 #define WCSHDO_safe      0x0000F
1317 #define WCSHDO_DOBSn     0x00001
1318 #define WCSHDO_TPCn_ka   0x00002
1319 #define WCSHDO_PVn_ma    0x00004
1320 #define WCSHDO_CRPXna    0x00008
1321 #define WCSHDO_CNAMna    0x00010
1322 #define WCSHDO_WCSNna    0x00020
1323 #define WCSHDO_P12       0x01000
1324 #define WCSHDO_P13       0x02000
1325 #define WCSHDO_P14       0x04000
1326 #define WCSHDO_P15       0x08000
1327 #define WCSHDO_P16       0x10000
1328 #define WCSHDO_P17       0x20000
1329 #define WCSHDO_EFMT      0x40000
1330
1331
1332 extern const char *wcshdr_errmsg[];
1333
1334 enum wcshdr_errmsg_enum {
1335   WCSHDRERR_SUCCESS          = 0,    // Success.
1336   WCSHDRERR_NULL_POINTER     = 1,        // Null wcsprm pointer passed.
```

```
1337    WCSHDRERR_MEMORY            = 2,     // Memory allocation failed.
1338    WCSHDRERR_BAD_COLUMN        = 3, // Invalid column selection.
1339    WCSHDRERR_PARSER            = 4,     // Fatal error returned by Flex
1340                                         // parser.
1341    WCSHDRERR_BAD_TABULAR_PARAMS = 5  // Invalid tabular parameters.
1342 };
1343
1344 int wcspih(char *header, int nkeyrec, int relax, int ctrl, int *nreject,
1345           int *nwcs, struct wcsprm **wcs);
1346
1347 int wcsbth(char *header, int nkeyrec, int relax, int ctrl, int keysel,
1348           int *colsel, int *nreject, int *nwcs, struct wcsprm **wcs);
1349
1350 int wcstab(struct wcsprm *wcs);
1351
1352 int wcsidx(int nwcs, struct wcsprm **wcs, int alts[27]);
1353
1354 int wcsbdx(int nwcs, struct wcsprm **wcs, int type, short alts[1000][28]);
1355
1356 int wcsvfree(int *nwcs, struct wcsprm **wcs);
1357
1358 int wcshdo(int ctrl, struct wcsprm *wcs, int *nkeyrec, char **header);
1359
1360
1361 #ifdef __cplusplus
1362 }
1363 #endif
1364
1365 #endif // WCSLIB_WCSHDR
```

19.31 wcsmath.h File Reference

Macros

- #define PI 3.141592653589793238462643
- #define D2R PI/180.0

    Degrees to radians conversion factor.
- #define R2D 180.0/PI

    Radians to degrees conversion factor.
- #define SQRT2 1.4142135623730950488
- #define SQRT2INV 1.0/SQRT2
- #define UNDEFINED 987654321.0e99

    Value used to indicate an undefined quantity.
- #define undefined(value) (value == UNDEFINED)

    Macro used to test for an undefined quantity.

19.31.1 Detailed Description

Definition of mathematical constants used by WCSLIB.

19.31.2 Macro Definition Documentation

19.31.2.1 PI #define PI 3.141592653589793238462643

19.31.2.2   D2R   #define D2R PI/180.0

Factor $\pi/180°$ to convert from degrees to radians.

19.31.2.3   R2D   #define R2D 180.0/PI

Factor $180°/\pi$ to convert from radians to degrees.

19.31.2.4   SQRT2   #define SQRT2 1.4142135623730950488

$\sqrt{2}$, used only by molset() (**MOL** projection).

19.31.2.5   SQRT2INV   #define SQRT2INV 1.0/SQRT2

$1/\sqrt{2}$, used only by qscx2s() (**QSC** projection).

19.31.2.6   UNDEFINED   #define UNDEFINED 987654321.0e99

Value used to indicate an undefined quantity (noting that NaNs cannot be used portably).

19.31.2.7   undefined   #define undefined(
                 value) (value == UNDEFINED)

Macro used to test for an undefined value.

19.32   wcsmath.h

Go to the documentation of this file.
```
1 /*============================================================================
2    WCSLIB 7.11 - an implementation of the FITS WCS standard.
3    Copyright (C) 1995-2022, Mark Calabretta
4
5    This file is part of WCSLIB.
6
7    WCSLIB is free software: you can redistribute it and/or modify it under the
8    terms of the GNU Lesser General Public License as published by the Free
9    Software Foundation, either version 3 of the License, or (at your option)
10    any later version.
11
12    WCSLIB is distributed in the hope that it will be useful, but WITHOUT ANY
13    WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
14    FOR A PARTICULAR PURPOSE.  See the GNU Lesser General Public License for
15    more details.
16
17    You should have received a copy of the GNU Lesser General Public License
18    along with WCSLIB.  If not, see http://www.gnu.org/licenses.
19
20    Author: Mark Calabretta, Australia Telescope National Facility, CSIRO.
21    http://www.atnf.csiro.au/people/Mark.Calabretta
22    $Id: wcsmath.h,v 7.11 2022/04/26 06:13:52 mcalabre Exp $
23  *=============================================================================
24  *
25  * WCSLIB 7.11 - C routines that implement the FITS World Coordinate System
26  * (WCS) standard.  Refer to the README file provided with WCSLIB for an
27  * overview of the library.
28  *
29  *
30  * Summary of wcsmath.h
31  * --------------------
32  * Definition of mathematical constants used by WCSLIB.
33  *
34  *=============================================================================*/
```

```
35
36  #ifndef WCSLIB_WCSMATH
37  #define WCSLIB_WCSMATH
38
39  #ifdef PI
40  #undef PI
41  #endif
42
43  #ifdef D2R
44  #undef D2R
45  #endif
46
47  #ifdef R2D
48  #undef R2D
49  #endif
50
51  #ifdef SQRT2
52  #undef SQRT2
53  #endif
54
55  #ifdef SQRT2INV
56  #undef SQRT2INV
57  #endif
58
59  #define PI 3.141592653589793238462643
60  #define D2R PI/180.0
61  #define R2D 180.0/PI
62  #define SQRT2 1.4142135623730950488
63  #define SQRT2INV 1.0/SQRT2
64
65  #ifdef UNDEFINED
66  #undef UNDEFINED
67  #endif
68
69  #define UNDEFINED 987654321.0e99
70  #define undefined(value) (value == UNDEFINED)
71
72  #endif // WCSLIB_WCSMATH
```

19.33 wcsprintf.h File Reference

```
#include <inttypes.h>
#include <stdio.h>
```

Macros

- #define WCSPRINTF_PTR(str1, ptr, str2)

    Print addresses in a consistent way.

Functions

- int wcsprintf_set (FILE ∗wcsout)

    Set output disposition for wcsprintf() and wcsfprintf().
- int wcsprintf (const char ∗format,...)

    Print function used by WCSLIB diagnostic routines.
- int wcsfprintf (FILE ∗stream, const char ∗format,...)

    Print function used by WCSLIB diagnostic routines.
- const char ∗ wcsprintf_buf (void)

    Get the address of the internal string buffer.

19.33.1 Detailed Description

Routines in this suite allow diagnostic output from celprt(), linprt(), prjprt(), spcprt(), tabprt(), wcsprt(), and wcserr_prt() to be redirected to a file or captured in a string buffer. Those routines all use wcsprintf() for output. Likewise wcsfprintf() is used by wcsbth() and wcspih(). Both functions may be used by application programmers to have other output go to the same place.

19.33.2 Macro Definition Documentation

19.33.2.1 WCSPRINTF_PTR #define WCSPRINTF_PTR(

   str1,

   ptr,

   str2)

Value:
```
if (ptr) { \
  wcsprintf("%s%#" PRIxPTR "%s", (str1), (uintptr_t)(ptr), (str2)); \
} else { \
  wcsprintf("%s0x0%s", (str1), (str2)); \
}
```

WCSPRINTF_PTR() is a preprocessor macro used to print addresses in a consistent way.

On some systems the "p" format descriptor renders a NULL pointer as the string "0x0". On others, however, it produces "0" or even "(nil)". On some systems a non-zero address is prefixed with "0x", on others, not.

The **WCSPRINTF_PTR**() macro ensures that a NULL pointer is always rendered as "0x0" and that non-zero addresses are prefixed with "0x" thus providing consistency, for example, for comparing the output of test programs.

19.33.3 Function Documentation

19.33.3.1 wcsprintf_set() int wcsprintf_set (

   FILE * *wcsout*)

wcsprintf_set() sets the output disposition for wcsprintf() which is used by the celprt(), linprt(), prjprt(), spcprt(), tabprt(), wcsprt(), and wcserr_prt() routines, and for wcsfprintf() which is used by wcsbth() and wcspih().

Parameters

in	*wcsout*	Pointer to an output stream that has been opened for writing, e.g. by the fopen() stdio library function, or one of the predefined stdio output streams - stdout and stderr. If zero (NULL), output is written to an internally-allocated string buffer, the address of which may be obtained by wcsprintf_buf().

Returns

   Status return value:

- 0: Success.

19.33.3.2 wcsprintf() `int wcsprintf (`
        `const char * format,`
        `... )`

wcsprintf() is used by [celprt()](), [linprt()](), [prjprt()](), [spcprt()](), [tabprt()](), [wcsprt()](), and [wcserr_prt()]() for diagnostic output which by default goes to stdout. However, it may be redirected to a file or string buffer via **wcsprintf_set**().

Parameters

in	*format*	Format string, passed to one of the printf(3) family of stdio library functions.
in	*...*	Argument list matching format, as per printf(3).

Returns

Number of bytes written.

19.33.3.3 wcsfprintf() `int wcsfprintf (`
        `FILE * stream,`
        `const char * format,`
        `... )`

wcsfprintf() is used by [wcsbth()](), and [wcspih()]() for diagnostic output which they send to stderr. However, it may be redirected to a file or string buffer via [wcsprintf_set()]().

Parameters

in	*stream*	The output stream if not overridden by a call to [wcsprintf_set()]().
in	*format*	Format string, passed to one of the printf(3) family of stdio library functions.
in	*...*	Argument list matching format, as per printf(3).

Returns

Number of bytes written.

19.33.3.4 wcsprintf_buf() `wcsprintf_buf (`
        `void )`

wcsprintf_buf() returns the address of the internal string buffer created when [wcsprintf_set()]() is invoked with its FILE∗ argument set to zero.

Returns

Address of the internal string buffer. The user may free this buffer by calling [wcsprintf_set()]() with a valid FILE∗, e.g. stdout. The free() stdlib library function must NOT be invoked on this const pointer.

19.34 wcsprintf.h

Go to the documentation of this file.
```
1 /*============================================================================
2   WCSLIB 7.11 - an implementation of the FITS WCS standard.
3   Copyright (C) 1995-2022, Mark Calabretta
4
5   This file is part of WCSLIB.
6
7   WCSLIB is free software: you can redistribute it and/or modify it under the
8   terms of the GNU Lesser General Public License as published by the Free
9   Software Foundation, either version 3 of the License, or (at your option)
10   any later version.
11
12   WCSLIB is distributed in the hope that it will be useful, but WITHOUT ANY
13   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
14   FOR A PARTICULAR PURPOSE.  See the GNU Lesser General Public License for
15   more details.
16
17   You should have received a copy of the GNU Lesser General Public License
18   along with WCSLIB.  If not, see http://www.gnu.org/licenses.
19
20   Author: Mark Calabretta, Australia Telescope National Facility, CSIRO.
21   http://www.atnf.csiro.au/people/Mark.Calabretta
22   $Id: wcsprintf.h,v 7.11 2022/04/26 06:13:52 mcalabre Exp $
23 *=============================================================================
24 *
25 * WCSLIB 7.11 - C routines that implement the FITS World Coordinate System
26 * (WCS) standard.  Refer to the README file provided with WCSLIB for an
27 * overview of the library.
28 *
29 *
30 * Summary of the wcsprintf routines
31 * ---------------------------------
32 * Routines in this suite allow diagnostic output from celprt(), linprt(),
33 * prjprt(), spcprt(), tabprt(), wcsprt(), and wcserr_prt() to be redirected to
34 * a file or captured in a string buffer.  Those routines all use wcsprintf()
35 * for output.  Likewise wcsfprintf() is used by wcsbth() and wcspih().  Both
36 * functions may be used by application programmers to have other output go to
37 * the same place.
38 *
39 *
40 * wcsprintf() - Print function used by WCSLIB diagnostic routines
41 * --------------------------------------------------------------
42 * wcsprintf() is used by celprt(), linprt(), prjprt(), spcprt(), tabprt(),
43 * wcsprt(), and wcserr_prt() for diagnostic output which by default goes to
44 * stdout.  However, it may be redirected to a file or string buffer via
45 * wcsprintf_set().
46 *
47 * Given:
48 *   format    char*    Format string, passed to one of the printf(3) family
49 *                      of stdio library functions.
50 *
51 *   ...       mixed    Argument list matching format, as per printf(3).
52 *
53 * Function return value:
54 *             int      Number of bytes written.
55 *
56 *
57 * wcsfprintf() - Print function used by WCSLIB diagnostic routines
58 * ---------------------------------------------------------------
59 * wcsfprintf() is used by wcsbth(), and wcspih() for diagnostic output which
60 * they send to stderr.  However, it may be redirected to a file or string
61 * buffer via wcsprintf_set().
62 *
63 * Given:
64 *   stream    FILE*    The output stream if not overridden by a call to
65 *                      wcsprintf_set().
66 *
67 *   format    char*    Format string, passed to one of the printf(3) family
68 *                      of stdio library functions.
69 *
70 *   ...       mixed    Argument list matching format, as per printf(3).
71 *
72 * Function return value:
73 *             int      Number of bytes written.
74 *
75 *
76 * wcsprintf_set() - Set output disposition for wcsprintf() and wcsfprintf()
77 * ------------------------------------------------------------------------
78 * wcsprintf_set() sets the output disposition for wcsprintf() which is used by
79 * the celprt(), linprt(), prjprt(), spcprt(), tabprt(), wcsprt(), and
80 * wcserr_prt() routines, and for wcsfprintf() which is used by wcsbth() and
81 * wcspih().
82 *
83 * Given:
```

```
84  *   wcsout     FILE*     Pointer to an output stream that has been opened for
85  *                        writing, e.g. by the fopen() stdio library function,
86  *                        or one of the predefined stdio output streams - stdout
87  *                        and stderr.  If zero (NULL), output is written to an
88  *                        internally-allocated string buffer, the address of
89  *                        which may be obtained by wcsprintf_buf().
90  *
91  * Function return value:
92  *             int        Status return value:
93  *                          0: Success.
94  *
95  *
96  * wcsprintf_buf() - Get the address of the internal string buffer
97  * ---------------------------------------------------------------
98  * wcsprintf_buf() returns the address of the internal string buffer created
99  * when wcsprintf_set() is invoked with its FILE* argument set to zero.
100 *
101 * Function return value:
102 *             const char *
103 *                        Address of the internal string buffer.  The user may
104 *                        free this buffer by calling wcsprintf_set() with a
105 *                        valid FILE*, e.g. stdout.  The free() stdlib library
106 *                        function must NOT be invoked on this const pointer.
107 *
108 *
109 * WCSPRINTF_PTR() macro - Print addresses in a consistent way
110 * -----------------------------------------------------------
111 * WCSPRINTF_PTR() is a preprocessor macro used to print addresses in a
112 * consistent way.
113 *
114 * On some systems the "%p" format descriptor renders a NULL pointer as the
115 * string "0x0".  On others, however, it produces "0" or even "(nil)".  On
116 * some systems a non-zero address is prefixed with "0x", on others, not.
117 *
118 * The WCSPRINTF_PTR() macro ensures that a NULL pointer is always rendered as
119 * "0x0" and that non-zero addresses are prefixed with "0x" thus providing
120 * consistency, for example, for comparing the output of test programs.
121 *
122 *=========================================================================*/
123
124 #ifndef WCSLIB_WCSPRINTF
125 #define WCSLIB_WCSPRINTF
126
127 #include <inttypes.h>
128 #include <stdio.h>
129
130 #ifdef __cplusplus
131 extern "C" {
132 #endif
133
134 #define WCSPRINTF_PTR(str1, ptr, str2) \
135   if (ptr) { \
136     wcsprintf("%s%#" PRIxPTR "%s", (str1), (uintptr_t)(ptr), (str2)); \
137   } else { \
138     wcsprintf("%s0x0%s", (str1), (str2)); \
139   }
140
141 int wcsprintf_set(FILE *wcsout);
142 int wcsprintf(const char *format, ...);
143 int wcsfprintf(FILE *stream, const char *format, ...);
144 const char *wcsprintf_buf(void);
145
146 #ifdef __cplusplus
147 }
148 #endif
149
150 #endif // WCSLIB_WCSPRINTF
```

19.35 wcstrig.h File Reference

#include <math.h>
#include "wcsconfig.h"

Macros

- #define WCSTRIG_TOL 1e-10

  Domain tolerance for asin() and acos() functions.

Functions

- double cosd (double angle)

    Cosine of an angle in degrees.

- double sind (double angle)

    Sine of an angle in degrees.

- void sincosd (double angle, double ∗sin, double ∗cos)

    Sine and cosine of an angle in degrees.

- double tand (double angle)

    Tangent of an angle in degrees.

- double acosd (double x)

    Inverse cosine, returning angle in degrees.

- double asind (double y)

    Inverse sine, returning angle in degrees.

- double atand (double s)

    Inverse tangent, returning angle in degrees.

- double atan2d (double y, double x)

    Polar angle of (x, y), in degrees.

19.35.1   Detailed Description

When dealing with celestial coordinate systems and spherical projections (some moreso than others) it is often desirable to use an angular measure that provides an exact representation of the latitude of the north or south pole. The WCSLIB routines use the following trigonometric functions that take or return angles in degrees:

- cosd()

- sind()

- tand()

- acosd()

- asind()

- atand()

- atan2d()

- sincosd()

These "trigd" routines are expected to handle angles that are a multiple of $90°$ returning an exact result. Some C implementations provide these as part of a system library and in such cases it may (or may not!) be preferable to use them. WCSLIB provides wrappers on the standard trig functions based on radian measure, adding tests for multiples of $90°$.

However, wcstrig.h also provides the choice of using preprocessor macro implementations of the trigd functions that don't test for multiples of $90°$ (compile with -DWCSTRIG_MACRO). These are typically 20% faster but may lead to problems near the poles.

19.35.2   Macro Definition Documentation

19.35.2.1 WCSTRIG_TOL `#define WCSTRIG_TOL 1e-10`

Domain tolerance for the asin() and acos() functions to allow for floating point rounding errors.

If v lies in the range $1 < |v| < 1 + WCSTRIG\_TOL$ then it will be treated as $|v| == 1$.

19.35.3 Function Documentation

19.35.3.1 cosd() `double cosd (`
                `double angle )`

cosd() returns the cosine of an angle given in degrees.

Parameters

in	*angle*	[deg].

Returns

Cosine of the angle.

19.35.3.2 sind() `double sind (`
                `double angle )`

sind() returns the sine of an angle given in degrees.

Parameters

in	*angle*	[deg].

Returns

Sine of the angle.

19.35.3.3 sincosd() `void sincosd (`
                `double angle,`
                `double * sin,`
                `double * cos )`

sincosd() returns the sine and cosine of an angle given in degrees.

Parameters

in	*angle*	[deg].
out	*sin*	Sine of the angle.
out	*cos*	Cosine of the angle.

Returns

19.35.3.4 tand() `double tand (`
                  `double angle )`

tand() returns the tangent of an angle given in degrees.

Parameters

in	*angle*	[deg].

Returns

Tangent of the angle.

19.35.3.5 acosd() `double acosd (`
                  `double x )`

acosd() returns the inverse cosine in degrees.

Parameters

in	*x*	in the range [-1,1].

Returns

Inverse cosine of x [deg].

19.35.3.6 asind() `double asind (`
                  `double y )`

asind() returns the inverse sine in degrees.

Parameters

in	*y*	in the range [-1,1].

Returns

Inverse sine of y [deg].

19.35.3.7 atand() ```double atand (```
```   double s)```

atand() returns the inverse tangent in degrees.

Parameters

in	*s*	

Returns

Inverse tangent of s [deg].

19.35.3.8 atan2d() ```double atan2d (```
```   double y,```
```   double x)```

atan2d() returns the polar angle, β, in degrees, of polar coordinates (ρ, β) corresponding to Cartesian coordinates (x, y). It is equivalent to the $\arg(x, y)$ function of WCS Paper II, though with transposed arguments.

Parameters

in	*y*	Cartesian y-coordinate.
in	*x*	Cartesian x-coordinate.

Returns

Polar angle of (x, y) [deg].

19.36 wcstrig.h

[Go to the documentation of this file.](#)
```
1 /*============================================================================
2   WCSLIB 7.11 - an implementation of the FITS WCS standard.
3   Copyright (C) 1995-2022, Mark Calabretta
4
5   This file is part of WCSLIB.
6
```

```
 7   WCSLIB is free software: you can redistribute it and/or modify it under the
 8   terms of the GNU Lesser General Public License as published by the Free
 9   Software Foundation, either version 3 of the License, or (at your option)
10   any later version.
11
12   WCSLIB is distributed in the hope that it will be useful, but WITHOUT ANY
13   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
14   FOR A PARTICULAR PURPOSE.  See the GNU Lesser General Public License for
15   more details.
16
17   You should have received a copy of the GNU Lesser General Public License
18   along with WCSLIB.  If not, see http://www.gnu.org/licenses.
19
20   Author: Mark Calabretta, Australia Telescope National Facility, CSIRO.
21   http://www.atnf.csiro.au/people/Mark.Calabretta
22   $Id: wcstrig.h,v 7.11 2022/04/26 06:13:52 mcalabre Exp $
23 *=============================================================================
24 *
25 * WCSLIB 7.11 - C routines that implement the FITS World Coordinate System
26 * (WCS) standard.  Refer to the README file provided with WCSLIB for an
27 * overview of the library.
28 *
29 *
30 * Summary of the wcstrig routines
31 * -------------------------------
32 * When dealing with celestial coordinate systems and spherical projections
33 * (some moreso than others) it is often desirable to use an angular measure
34 * that provides an exact representation of the latitude of the north or south
35 * pole.  The WCSLIB routines use the following trigonometric functions that
36 * take or return angles in degrees:
37 *
38 *   - cosd()
39 *   - sind()
40 *   - tand()
41 *   - acosd()
42 *   - asind()
43 *   - atand()
44 *   - atan2d()
45 *   - sincosd()
46 *
47 * These "trigd" routines are expected to handle angles that are a multiple of
48 * 90 degrees returning an exact result.  Some C implementations provide these
49 * as part of a system library and in such cases it may (or may not!) be
50 * preferable to use them.  WCSLIB provides wrappers on the standard trig
51 * functions based on radian measure, adding tests for multiples of 90 degrees.
52 *
53 * However, wcstrig.h also provides the choice of using preprocessor macro
54 * implementations of the trigd functions that don't test for multiples of
55 * 90 degrees (compile with -DWCSTRIG_MACRO).  These are typically 20% faster
56 * but may lead to problems near the poles.
57 *
58 *
59 * cosd() - Cosine of an angle in degrees
60 * --------------------------------------
61 * cosd() returns the cosine of an angle given in degrees.
62 *
63 * Given:
64 *   angle     double    [deg].
65 *
66 * Function return value:
67 *             double    Cosine of the angle.
68 *
69 *
70 * sind() - Sine of an angle in degrees
71 * ------------------------------------
72 * sind() returns the sine of an angle given in degrees.
73 *
74 * Given:
75 *   angle     double    [deg].
76 *
77 * Function return value:
78 *             double    Sine of the angle.
79 *
80 *
81 * sincosd() - Sine and cosine of an angle in degrees
82 * --------------------------------------------------
83 * sincosd() returns the sine and cosine of an angle given in degrees.
84 *
85 * Given:
86 *   angle     double    [deg].
87 *
88 * Returned:
89 *   sin       *double   Sine of the angle.
90 *
91 *   cos       *double   Cosine of the angle.
92 *
93 * Function return value:
```

```
 94 *              void
 95 *
 96 *
 97 * tand() - Tangent of an angle in degrees
 98 * ----------------------------------------
 99 * tand() returns the tangent of an angle given in degrees.
100 *
101 * Given:
102 *   angle     double    [deg].
103 *
104 * Function return value:
105 *             double    Tangent of the angle.
106 *
107 *
108 * acosd() - Inverse cosine, returning angle in degrees
109 * -----------------------------------------------------
110 * acosd() returns the inverse cosine in degrees.
111 *
112 * Given:
113 *   x         double    in the range [-1,1].
114 *
115 * Function return value:
116 *             double    Inverse cosine of x [deg].
117 *
118 *
119 * asind() - Inverse sine, returning angle in degrees
120 * ---------------------------------------------------
121 * asind() returns the inverse sine in degrees.
122 *
123 * Given:
124 *   y         double    in the range [-1,1].
125 *
126 * Function return value:
127 *             double    Inverse sine of y [deg].
128 *
129 *
130 * atand() - Inverse tangent, returning angle in degrees
131 * ------------------------------------------------------
132 * atand() returns the inverse tangent in degrees.
133 *
134 * Given:
135 *   s         double
136 *
137 * Function return value:
138 *             double    Inverse tangent of s [deg].
139 *
140 *
141 * atan2d() - Polar angle of (x,y), in degrees
142 * --------------------------------------------
143 * atan2d() returns the polar angle, beta, in degrees, of polar coordinates
144 * (rho,beta) corresponding to Cartesian coordinates (x,y).  It is equivalent
145 * to the arg(x,y) function of WCS Paper II, though with transposed arguments.
146 *
147 * Given:
148 *   y         double    Cartesian y-coordinate.
149 *
150 *   x         double    Cartesian x-coordinate.
151 *
152 * Function return value:
153 *             double    Polar angle of (x,y) [deg].
154 *
155 *===========================================================================*/
156
157 #ifndef WCSLIB_WCSTRIG
158 #define WCSLIB_WCSTRIG
159
160 #include <math.h>
161
162 #include "wcsconfig.h"
163
164 #ifdef HAVE_SINCOS
165   void sincos(double angle, double *sin, double *cos);
166 #endif
167
168 #ifdef __cplusplus
169 extern "C" {
170 #endif
171
172
173 #ifdef WCSTRIG_MACRO
174
175 // Macro implementation of the trigd functions.
176 #include "wcsmath.h"
177
178 #define cosd(X) cos((X)*D2R)
179 #define sind(X) sin((X)*D2R)
180 #define tand(X) tan((X)*D2R)
```

```
181 #define acosd(X) acos(X)*R2D
182 #define asind(X) asin(X)*R2D
183 #define atand(X) atan(X)*R2D
184 #define atan2d(Y,X) atan2(Y,X)*R2D
185 #ifdef HAVE_SINCOS
186   #define sincosd(X,S,C) sincos((X)*D2R,(S),(C))
187 #else
188   #define sincosd(X,S,C) *(S) = sin((X)*D2R); *(C) = cos((X)*D2R);
189 #endif
190
191 #else
192
193 // Use WCSLIB wrappers or native trigd functions.
194
195 double cosd(double angle);
196 double sind(double angle);
197 void sincosd(double angle, double *sin, double *cos);
198 double tand(double angle);
199 double acosd(double x);
200 double asind(double y);
201 double atand(double s);
202 double atan2d(double y, double x);
203
204 // Domain tolerance for asin() and acos() functions.
205 #define WCSTRIG_TOL 1e-10
206
207 #endif // WCSTRIG_MACRO
208
209
210 #ifdef __cplusplus
211 }
212 #endif
213
214 #endif // WCSLIB_WCSTRIG
```

19.37   wcsunits.h File Reference

```
#include "wcserr.h"
```

Macros

- #define WCSUNITS_PLANE_ANGLE 0

  Array index for plane angle units type.
- #define WCSUNITS_SOLID_ANGLE 1

  Array index for solid angle units type.
- #define WCSUNITS_CHARGE 2

  Array index for charge units type.
- #define WCSUNITS_MOLE 3

  Array index for mole units type.
- #define WCSUNITS_TEMPERATURE 4

  Array index for temperature units type.
- #define WCSUNITS_LUMINTEN 5

  Array index for luminous intensity units type.
- #define WCSUNITS_MASS 6

  Array index for mass units type.
- #define WCSUNITS_LENGTH 7

  Array index for length units type.
- #define WCSUNITS_TIME 8

  Array index for time units type.
- #define WCSUNITS_BEAM 9

  Array index for beam units type.
- #define WCSUNITS_BIN 10

Array index for bin units type.

- #define WCSUNITS_BIT 11

    Array index for bit units type.

- #define WCSUNITS_COUNT 12

    Array index for count units type.

- #define WCSUNITS_MAGNITUDE 13

    Array index for stellar magnitude units type.

- #define WCSUNITS_PIXEL 14

    Array index for pixel units type.

- #define WCSUNITS_SOLRATIO 15

    Array index for solar mass ratio units type.

- #define WCSUNITS_VOXEL 16

    Array index for voxel units type.

- #define WCSUNITS_NTYPE 17

    Number of entries in the units array.

Enumerations

- enum wcsunits_errmsg_enum {
  UNITSERR_SUCCESS = 0 , UNITSERR_BAD_NUM_MULTIPLIER = 1 , UNITSERR_DANGLING_BINOP =
  2 , UNITSERR_BAD_INITIAL_SYMBOL = 3 ,
  UNITSERR_FUNCTION_CONTEXT = 4 , UNITSERR_BAD_EXPON_SYMBOL = 5 , UNITSERR_UNBAL_BRACKET
  = 6 , UNITSERR_UNBAL_PAREN = 7 ,
  UNITSERR_CONSEC_BINOPS = 8 , UNITSERR_PARSER_ERROR = 9 , UNITSERR_BAD_UNIT_SPEC =
  10 , UNITSERR_BAD_FUNCS = 11 ,
  UNITSERR_UNSAFE_TRANS = 12 }

Functions

- int wcsunitse (const char have[ ], const char want[ ], double *scale, double *offset, double *power, struct
  wcserr **err)

    FITS units specification conversion.

- int wcsutrne (int ctrl, char unitstr[ ], struct wcserr **err)

    Translation of non-standard unit specifications.

- int wcsulexe (const char unitstr[ ], int *func, double *scale, double units[WCSUNITS_NTYPE], struct wcserr
  **err)

    FITS units specification parser.

- int wcsunits (const char have[ ], const char want[ ], double *scale, double *offset, double *power)
- int wcsutrn (int ctrl, char unitstr[ ])
- int wcsulex (const char unitstr[ ], int *func, double *scale, double units[WCSUNITS_NTYPE])

Variables

- const char * wcsunits_errmsg [ ]

    Status return messages.

- const char * wcsunits_types [ ]

    Names of physical quantities.

- const char * wcsunits_units [ ]

    Names of units.

19.37.1 Detailed Description

Routines in this suite deal with units specifications and conversions, as described in
```
"Representations of world coordinates in FITS",
Greisen, E.W., & Calabretta, M.R. 2002, A&A, 395, 1061 (WCS Paper I)
```

The Flexible Image Transport System (FITS), a data format widely used in astronomy for data interchange and archive, is described in
```
"Definition of the Flexible Image Transport System (FITS), version 3.0",
Pence, W.D., Chiappetti, L., Page, C.G., Shaw, R.A., & Stobie, E. 2010,
A&A, 524, A42 - http://dx.doi.org/10.1051/0004-6361/201015362
```

See also http:
These routines perform basic units-related operations:

- wcsunitse(): given two unit specifications, derive the conversion from one to the other.

- wcsutrne(): translates certain commonly used but non-standard unit strings. It is intended to be called before wcsulexe() which only handles standard FITS units specifications.

- wcsulexe(): parses a standard FITS units specification of arbitrary complexity, deriving the conversion to canonical units.

19.37.2 Macro Definition Documentation

19.37.2.1 WCSUNITS_PLANE_ANGLE #define WCSUNITS_PLANE_ANGLE 0

Array index for plane angle units in the *units* array returned by wcsulex(), and the wcsunits_types[] and wcsunits_units[] global variables.

19.37.2.2 WCSUNITS_SOLID_ANGLE #define WCSUNITS_SOLID_ANGLE 1

Array index for solid angle units in the *units* array returned by wcsulex(), and the wcsunits_types[] and wcsunits_units[] global variables.

19.37.2.3 WCSUNITS_CHARGE #define WCSUNITS_CHARGE 2

Array index for charge units in the *units* array returned by wcsulex(), and the wcsunits_types[] and wcsunits_units[] global variables.

19.37.2.4 WCSUNITS_MOLE #define WCSUNITS_MOLE 3

Array index for mole ("gram molecular weight") units in the *units* array returned by wcsulex(), and the wcsunits_types[] and wcsunits_units[] global variables.

19.37.2.5 WCSUNITS_TEMPERATURE #define WCSUNITS_TEMPERATURE 4

Array index for temperature units in the *units* array returned by wcsulex(), and the wcsunits_types[] and wcsunits_units[] global variables.

19.37.2.6 WCSUNITS_LUMINTEN `#define WCSUNITS_LUMINTEN 5`

Array index for luminous intensity units in the *units* array returned by wcsulex(), and the wcsunits_types[] and wcsunits_units[] global variables.

19.37.2.7 WCSUNITS_MASS `#define WCSUNITS_MASS 6`

Array index for mass units in the *units* array returned by wcsulex(), and the wcsunits_types[] and wcsunits_units[] global variables.

19.37.2.8 WCSUNITS_LENGTH `#define WCSUNITS_LENGTH 7`

Array index for length units in the *units* array returned by wcsulex(), and the wcsunits_types[] and wcsunits_units[] global variables.

19.37.2.9 WCSUNITS_TIME `#define WCSUNITS_TIME 8`

Array index for time units in the *units* array returned by wcsulex(), and the wcsunits_types[] and wcsunits_units[] global variables.

19.37.2.10 WCSUNITS_BEAM `#define WCSUNITS_BEAM 9`

Array index for beam units in the *units* array returned by wcsulex(), and the wcsunits_types[] and wcsunits_units[] global variables.

19.37.2.11 WCSUNITS_BIN `#define WCSUNITS_BIN 10`

Array index for bin units in the *units* array returned by wcsulex(), and the wcsunits_types[] and wcsunits_units[] global variables.

19.37.2.12 WCSUNITS_BIT `#define WCSUNITS_BIT 11`

Array index for bit units in the *units* array returned by wcsulex(), and the wcsunits_types[] and wcsunits_units[] global variables.

19.37.2.13 WCSUNITS_COUNT `#define WCSUNITS_COUNT 12`

Array index for count units in the *units* array returned by wcsulex(), and the wcsunits_types[] and wcsunits_units[] global variables.

19.37.2.14 WCSUNITS_MAGNITUDE `#define WCSUNITS_MAGNITUDE 13`

Array index for stellar magnitude units in the *units* array returned by wcsulex(), and the wcsunits_types[] and wcsunits_units[] global variables.

19.37.2.15 WCSUNITS_PIXEL `#define WCSUNITS_PIXEL 14`

Array index for pixel units in the *units* array returned by wcsulex(), and the wcsunits_types[] and wcsunits_units[] global variables.

19.37.2.16 WCSUNITS_SOLRATIO #define WCSUNITS_SOLRATIO 15

Array index for solar mass ratio units in the *units* array returned by wcsulex(), and the wcsunits_types[] and wcsunits_units[] global variables.

19.37.2.17 WCSUNITS_VOXEL #define WCSUNITS_VOXEL 16

Array index for voxel units in the *units* array returned by wcsulex(), and the wcsunits_types[] and wcsunits_units[] global variables.

19.37.2.18 WCSUNITS_NTYPE #define WCSUNITS_NTYPE 17

Number of entries in the *units* array returned by wcsulex(), and the wcsunits_types[] and wcsunits_units[] global variables.

19.37.3 Enumeration Type Documentation

19.37.3.1 wcsunits_errmsg_enum enum wcsunits_errmsg_enum

Enumerator

UNITSERR_SUCCESS	
UNITSERR_BAD_NUM_MULTIPLIER	
UNITSERR_DANGLING_BINOP	
UNITSERR_BAD_INITIAL_SYMBOL	
UNITSERR_FUNCTION_CONTEXT	
UNITSERR_BAD_EXPON_SYMBOL	
UNITSERR_UNBAL_BRACKET	
UNITSERR_UNBAL_PAREN	
UNITSERR_CONSEC_BINOPS	
UNITSERR_PARSER_ERROR	
UNITSERR_BAD_UNIT_SPEC	
UNITSERR_BAD_FUNCS	
UNITSERR_UNSAFE_TRANS	

19.37.4 Function Documentation

19.37.4.1 wcsunitse() int wcsunitse (
        const char *have[],*
        const char *want[],*
        double * *scale,*
        double * *offset,*

```
        double * power,
        struct wcserr ** err )
```

wcsunitse() derives the conversion from one system of units to another.

A deprecated form of this function, wcsunits(), lacks the wcserr∗∗ parameter.

Parameters

in	*have*	FITS units specification to convert from (null- terminated), with or without surrounding square brackets (for inline specifications); text following the closing bracket is ignored.
in	*want*	FITS units specification to convert to (null- terminated), with or without surrounding square brackets (for inline specifications); text following the closing bracket is ignored.
out	*scale,offset,power*	Convert units using `pow(scale*value + offset, power);` Normally *offset* is zero except for log() or ln() conversions, e.g. "log(MHz)" to "ln(Hz)". Likewise, *power* is normally unity except for exp() conversions, e.g. "exp(ms)" to "exp(/Hz)". Thus conversions ordinarily consist of `value *= scale;`
out	*err*	If enabled, for function return values > 1, this struct will contain a detailed error message, see wcserr_enable(). May be NULL if an error message is not desired. Otherwise, the user is responsible for deleting the memory allocated for the wcserr struct.

Returns

Status return value:

- 0: Success.
- 1-9: Status return from wcsulexe().
- 10: Non-conformant unit specifications.
- 11: Non-conformant functions.

scale is zeroed on return if an error occurs.

19.37.4.2 wcsutrne() `int wcsutrne (`
```
        int ctrl,
        char unitstr[],
        struct wcserr ** err )
```

wcsutrne() translates certain commonly used but non-standard unit strings, e.g. "DEG", "MHZ", "KELVIN", that are not recognized by wcsulexe(), refer to the notes below for a full list. Compounds are also recognized, e.g. "JY/BEAM" and "KM/SEC/SEC". Extraneous embedded blanks are removed.

A deprecated form of this function, wcsutrn(), lacks the wcserr∗∗ parameter.

Parameters

in	*ctrl*	Although "S" is commonly used to represent seconds, its translation to "s" is potentially unsafe since the standard recognizes "S" formally as Siemens, however rarely that may be used. The same applies to "H" for hours (Henry), and "D" for days (Debye). This bit-flag controls what to do in such cases:
		• 1: Translate "S" to "s".
		• 2: Translate "H" to "h".
		• 4: Translate "D" to "d".
		Thus ctrl == 0 doesn't do any unsafe translations, whereas ctrl == 7 does all of them.
in,out	*unitstr*	Null-terminated character array containing the units specification to be translated. Inline units specifications in a FITS header keycomment are also handled. If the first non-blank character in unitstr is '[' then the unit string is delimited by its matching ']'. Blanks preceding '[' will be stripped off, but text following the closing bracket will be preserved without modification.
in,out	*err*	If enabled, for function return values > 1, this struct will contain a detailed error message, see [wcserr_enable()](). May be NULL if an error message is not desired. Otherwise, the user is responsible for deleting the memory allocated for the wcserr struct.

Returns

Status return value:

- -1: No change was made, other than stripping blanks (not an error).
- 0: Success.
- 9: Internal parser error.
- 12: Potentially unsafe translation, whether applied or not (see notes).

Notes:

1. Translation of non-standard unit specifications: apart from leading and trailing blanks, a case-sensitive match is required for the aliases listed below, in particular the only recognized aliases with metric prefixes are "KM", "KHZ", "MHZ", and "GHZ". Potentially unsafe translations of "D", "H", and "S", shown in parentheses, are optional.

```
Unit      Recognized aliases
----      -------------------------------------------------------
Angstrom  Angstroms angstrom angstroms
arcmin    arcmins, ARCMIN, ARCMINS
arcsec    arcsecs, ARCSEC, ARCSECS
beam      BEAM
byte      Byte
d         day, days, (D), DAY, DAYS
deg       degree, degrees, Deg, Degree, Degrees, DEG, DEGREE,
          DEGREES
GHz       GHZ
h         hr, (H), HR
Hz        hz, HZ
kHz       KHZ
Jy        JY
K         kelvin, kelvins, Kelvin, Kelvins, KELVIN, KELVINS
km        KM
m         metre, meter, metres, meters, M, METRE, METER, METRES,
          METERS
min       MIN
MHz       MHZ
Ohm       ohm
Pa        pascal, pascals, Pascal, Pascals, PASCAL, PASCALS
pixel     pixels, PIXEL, PIXELS
```

```
rad          radian, radians, RAD, RADIAN, RADIANS
s            sec, second, seconds, (S), SEC, SECOND, SECONDS
V            volt, volts, Volt, Volts, VOLT, VOLTS
yr           year, years, YR, YEAR, YEARS
```

The aliases "angstrom", "ohm", and "Byte" for (Angstrom, Ohm, and byte) are recognized by wcsulexe() itself as an unofficial extension of the standard, but they are converted to the standard form here.

19.37.4.3 wcsulexe() `int wcsulexe (`
            `const char unitstr[],`
            `int * func,`
            `double * scale,`
            `double units[WCSUNITS_NTYPE],`
            `struct wcserr ** err )`

wcsulexe() parses a standard FITS units specification of arbitrary complexity, deriving the scale factor required to convert to canonical units - basically SI with degrees and "dimensionless" additions such as byte, pixel and count.

A deprecated form of this function, wcsulex(), lacks the wcserr** parameter.

Parameters

in	*unitstr*	Null-terminated character array containing the units specification, with or without surrounding square brackets (for inline specifications); text following the closing bracket is ignored.
out	*func*	Special function type, see note 4: • 0: None • 1: log() ...base 10 • 2: ln() ...base e • 3: exp()
out	*scale*	Scale factor for the unit specification; multiply a value expressed in the given units by this factor to convert it to canonical units.
out	*units*	A units specification is decomposed into powers of 16 fundamental unit types: angle, mass, length, time, count, pixel, etc. Preprocessor macro WCSUNITS_NTYPE is defined to dimension this vector, and others such WCSUNITS_PLANE_ANGLE, WCSUNITS_LENGTH, etc. to access its elements. Corresponding character strings, wcsunits_types[] and wcsunits_units[], are predefined to describe each quantity and its canonical units.
out	*err*	If enabled, for function return values > 1, this struct will contain a detailed error message, see wcserr_enable(). May be NULL if an error message is not desired. Otherwise, the user is responsible for deleting the memory allocated for the wcserr struct.

Returns

Status return value:

• 0: Success.

• 1: Invalid numeric multiplier.

• 2: Dangling binary operator.

• 3: Invalid symbol in INITIAL context.

• 4: Function in invalid context.

  - 5: Invalid symbol in EXPON context.
  - 6: Unbalanced bracket.
  - 7: Unbalanced parenthesis.
  - 8: Consecutive binary operators.
  - 9: Internal parser error.

scale and units[] are zeroed on return if an error occurs.

Notes:

1. **wcsulexe**() is permissive in accepting whitespace in all contexts in a units specification where it does not create ambiguity (e.g. not between a metric prefix and a basic unit string), including in strings like "log (m $**$ 2)" which is formally disallowed.

2. Supported extensions:

   - "angstrom" (OGIP usage) is allowed in addition to "Angstrom".
   - "ohm" (OGIP usage) is allowed in addition to "Ohm".
   - "Byte" (common usage) is allowed in addition to "byte".

3. Table 6 of WCS Paper I lists eleven units for which metric prefixes are allowed. However, in this implementation only prefixes greater than unity are allowed for "a" (annum), "yr" (year), "pc" (parsec), "bit", and "byte", and only prefixes less than unity are allowed for "mag" (stellar magnitude).

   Metric prefix "P" (peta) is specifically forbidden for "a" (annum) to avoid confusion with "Pa" (Pascal, not peta-annum). Note that metric prefixes are specifically disallowed for "h" (hour) and "d" (day) so that "ph" (photons) cannot be interpreted as pico-hours, nor "cd" (candela) as centi-days.

4. Function types log(), ln() and exp() may only occur at the start of the units specification. The scale and units[] returned for these refers to the string inside the function "argument", e.g. to "MHz" in log(MHz) for which a scale of 10^6 will be returned.

19.37.4.4   wcsunits() `int wcsunits (`
```
        const char have[],
        const char want[],
        double * scale,
        double * offset,
        double * power )
```

19.37.4.5   wcsutrn() `int wcsutrn (`
```
        int ctrl,
        char unitstr[] )
```

19.37.4.6   wcsulex() `int wcsulex (`
```
        const char unitstr[],
        int * func,
        double * scale,
        double units[WCSUNITS_NTYPE] )
```

19.37.5 Variable Documentation

19.37.5.1 wcsunits_errmsg `const char * wcsunits_errmsg[]` `[extern]`

Error messages to match the status value returned from each function.

19.37.5.2 wcsunits_types `const char * wcsunits_types[]` `[extern]`

Names for physical quantities to match the units vector returned by **wcsulexe**():

- 0: plane angle
- 1: solid angle
- 2: charge
- 3: mole
- 4: temperature
- 5: luminous intensity
- 6: mass
- 7: length
- 8: time
- 9: beam
- 10: bin
- 11: bit
- 12: count
- 13: stellar magnitude
- 14: pixel
- 15: solar ratio
- 16: voxel

19.37.5.3 wcsunits_units `const char * wcsunits_units[]` `[extern]`

Names for the units (SI) to match the units vector returned by **wcsulexe**():

- 0: degree
- 1: steradian
- 2: Coulomb
- 3: mole
- 4: Kelvin
- 5: candela
- 6: kilogram
- 7: metre
- 8: second

The remainder are dimensionless.

19.38 wcsunits.h

```
1 /*============================================================================
2    WCSLIB 7.11 - an implementation of the FITS WCS standard.
3    Copyright (C) 1995-2022, Mark Calabretta
4
5    This file is part of WCSLIB.
6
7    WCSLIB is free software: you can redistribute it and/or modify it under the
8    terms of the GNU Lesser General Public License as published by the Free
9    Software Foundation, either version 3 of the License, or (at your option)
10   any later version.
11
12   WCSLIB is distributed in the hope that it will be useful, but WITHOUT ANY
13   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
14   FOR A PARTICULAR PURPOSE.  See the GNU Lesser General Public License for
15   more details.
16
17   You should have received a copy of the GNU Lesser General Public License
18   along with WCSLIB.  If not, see http://www.gnu.org/licenses.
19
20   Author: Mark Calabretta, Australia Telescope National Facility, CSIRO.
21   http://www.atnf.csiro.au/people/Mark.Calabretta
22   $Id: wcsunits.h,v 7.11 2022/04/26 06:13:52 mcalabre Exp $
23 *============================================================================
24 *
25 * WCSLIB 7.11 - C routines that implement the FITS World Coordinate System
26 * (WCS) standard.  Refer to the README file provided with WCSLIB for an
27 * overview of the library.
28 *
29 *
30 * Summary of the wcsunits routines
31 * --------------------------------
32 * Routines in this suite deal with units specifications and conversions, as
33 * described in
34 *
35 =    "Representations of world coordinates in FITS",
36 =    Greisen, E.W., & Calabretta, M.R. 2002, A&A, 395, 1061 (WCS Paper I)
37 *
38 * The Flexible Image Transport System (FITS), a data format widely used in
39 * astronomy for data interchange and archive, is described in
40 *
41 =    "Definition of the Flexible Image Transport System (FITS), version 3.0",
42 =    Pence, W.D., Chiappetti, L., Page, C.G., Shaw, R.A., & Stobie, E. 2010,
43 =    A&A, 524, A42 - http://dx.doi.org/10.1051/0004-6361/201015362
44 *
45 * See also http://fits.gsfc.nasa.gov
46 *
47 * These routines perform basic units-related operations:
48 *
49 *    - wcsunitse(): given two unit specifications, derive the conversion from
50 *      one to the other.
51 *
52 *    - wcsutrne(): translates certain commonly used but non-standard unit
53 *      strings.  It is intended to be called before wcsulexe() which only
54 *      handles standard FITS units specifications.
55 *
56 *    - wcsulexe(): parses a standard FITS units specification of arbitrary
57 *      complexity, deriving the conversion to canonical units.
58 *
59 *
60 * wcsunitse() - FITS units specification conversion
61 * -------------------------------------------------
62 * wcsunitse() derives the conversion from one system of units to another.
63 *
64 * A deprecated form of this function, wcsunits(), lacks the wcserr**
65 * parameter.
66 *
67 * Given:
68 *    have      const char []
69 *                      FITS units specification to convert from (null-
70 *                      terminated), with or without surrounding square
71 *                      brackets (for inline specifications); text following
72 *                      the closing bracket is ignored.
73 *
74 *    want      const char []
75 *                      FITS units specification to convert to (null-
76 *                      terminated), with or without surrounding square
77 *                      brackets (for inline specifications); text following
78 *                      the closing bracket is ignored.
79 *
80 * Returned:
81 *    scale,
82 *    offset,
83 *    power     double*   Convert units using
```

```
 84 *
 85 =                        pow(scale*value + offset, power);
 86 *
 87 *                     Normally offset is zero except for log() or ln()
 88 *                     conversions, e.g. "log(MHz)" to "ln(Hz)".  Likewise,
 89 *                     power is normally unity except for exp() conversions,
 90 *                     e.g. "exp(ms)" to "exp(/Hz)".  Thus conversions
 91 *                     ordinarily consist of
 92 *
 93 =                        value *= scale;
 94 *
 95 *   err       struct wcserr **
 96 *                     If enabled, for function return values > 1, this
 97 *                     struct will contain a detailed error message, see
 98 *                     wcserr_enable().  May be NULL if an error message is
 99 *                     not desired.  Otherwise, the user is responsible for
100 *                      deleting the memory allocated for the wcserr struct.
101 *
102 * Function return value:
103 *             int       Status return value:
104 *                          0: Success.
105 *                        1-9: Status return from wcsulexe().
106 *                         10: Non-conformant unit specifications.
107 *                         11: Non-conformant functions.
108 *
109 *                     scale is zeroed on return if an error occurs.
110 *
111 *
112 * wcsutrne() - Translation of non-standard unit specifications
113 * ------------------------------------------------------------
114 * wcsutrne() translates certain commonly used but non-standard unit strings,
115 * e.g. "DEG", "MHZ", "KELVIN", that are not recognized by wcsulexe(), refer to
116 * the notes below for a full list.  Compounds are also recognized, e.g.
117 * "JY/BEAM" and "KM/SEC/SEC".  Extraneous embedded blanks are removed.
118 *
119 * A deprecated form of this function, wcsutrn(), lacks the wcserr** parameter.
120 *
121 * Given:
122 *   ctrl      int       Although "S" is commonly used to represent seconds,
123 *                     its translation to "s" is potentially unsafe since the
124 *                     standard recognizes "S" formally as Siemens, however
125 *                     rarely that may be used.  The same applies to "H" for
126 *                     hours (Henry), and "D" for days (Debye).  This
127 *                     bit-flag controls what to do in such cases:
128 *                        1: Translate "S" to "s".
129 *                        2: Translate "H" to "h".
130 *                        4: Translate "D" to "d".
131 *                     Thus ctrl == 0 doesn't do any unsafe translations,
132 *                     whereas ctrl == 7 does all of them.
133 *
134 * Given and returned:
135 *   unitstr   char []   Null-terminated character array containing the units
136 *                     specification to be translated.
137 *
138 *                     Inline units specifications in a FITS header
139 *                     keycomment are also handled.  If the first non-blank
140 *                     character in unitstr is '[' then the unit string is
141 *                     delimited by its matching ']'.  Blanks preceding '['
142 *                     will be stripped off, but text following the closing
143 *                     bracket will be preserved without modification.
144 *
145 *   err       struct wcserr **
146 *                     If enabled, for function return values > 1, this
147 *                     struct will contain a detailed error message, see
148 *                     wcserr_enable().  May be NULL if an error message is
149 *                     not desired.  Otherwise, the user is responsible for
150 *                     deleting the memory allocated for the wcserr struct.
151 *
152 * Function return value:
153 *             int       Status return value:
154 *                         -1: No change was made, other than stripping blanks
155 *                             (not an error).
156 *                          0: Success.
157 *                          9: Internal parser error.
158 *                         12: Potentially unsafe translation, whether applied
159 *                             or not (see notes).
160 *
161 * Notes:
162 *   1: Translation of non-standard unit specifications: apart from leading and
163 *      trailing blanks, a case-sensitive match is required for the aliases
164 *      listed below, in particular the only recognized aliases with metric
165 *      prefixes are "KM", "KHZ", "MHZ", and "GHZ".  Potentially unsafe
166 *      translations of "D", "H", and "S", shown in parentheses, are optional.
167 *
168 =          Unit       Recognized aliases
169 =          ----       -----------------------------------------------------------
170 =          Angstrom   Angstroms angstrom angstroms
```

```
171 =         arcmin    arcmins, ARCMIN, ARCMINS
172 =         arcsec    arcsecs, ARCSEC, ARCSECS
173 =         beam      BEAM
174 =         byte      Byte
175 =         d         day, days, (D), DAY, DAYS
176 =         deg       degree, degrees, Deg, Degree, Degrees, DEG, DEGREE,
177 =                   DEGREES
178 =         GHz       GHZ
179 =         h         hr, (H), HR
180 =         Hz        hz, HZ
181 =         kHz       KHZ
182 =         Jy        JY
183 =         K         kelvin, kelvins, Kelvin, Kelvins, KELVIN, KELVINS
184 =         km        KM
185 =         m         metre, meter, metres, meters, M, METRE, METER, METRES,
186 =                   METERS
187 =         min       MIN
188 =         MHz       MHZ
189 =         Ohm       ohm
190 =         Pa        pascal, pascals, Pascal, Pascals, PASCAL, PASCALS
191 =         pixel     pixels, PIXEL, PIXELS
192 =         rad       radian, radians, RAD, RADIAN, RADIANS
193 =         s         sec, second, seconds, (S), SEC, SECOND, SECONDS
194 =         V         volt, volts, Volt, Volts, VOLT, VOLTS
195 =         yr        year, years, YR, YEAR, YEARS
196 *
197 *      The aliases "angstrom", "ohm", and "Byte" for (Angstrom, Ohm, and byte)
198 *      are recognized by wcsulexe() itself as an unofficial extension of the
199 *      standard, but they are converted to the standard form here.
200 *
201 *
202 * wcsulexe() - FITS units specification parser
203 * -------------------------------------------
204 * wcsulexe() parses a standard FITS units specification of arbitrary
205 * complexity, deriving the scale factor required to convert to canonical
206 * units - basically SI with degrees and "dimensionless" additions such as
207 * byte, pixel and count.
208 *
209 * A deprecated form of this function, wcsulex(), lacks the wcserr** parameter.
210 *
211 * Given:
212 *   unitstr   const char []
213 *                       Null-terminated character array containing the units
214 *                       specification, with or without surrounding square
215 *                       brackets (for inline specifications); text following
216 *                       the closing bracket is ignored.
217 *
218 * Returned:
219 *   func      int*      Special function type, see note 4:
220 *                         0: None
221 *                         1: log()  ...base 10
222 *                         2: ln()   ...base e
223 *                         3: exp()
224 *
225 *   scale     double*   Scale factor for the unit specification; multiply a
226 *                       value expressed in the given units by this factor to
227 *                       convert it to canonical units.
228 *
229 *   units     double[WCSUNITS_NTYPE]
230 *                       A units specification is decomposed into powers of 16
231 *                       fundamental unit types: angle, mass, length, time,
232 *                       count, pixel, etc.  Preprocessor macro WCSUNITS_NTYPE
233 *                       is defined to dimension this vector, and others such
234 *                       WCSUNITS_PLANE_ANGLE, WCSUNITS_LENGTH, etc. to access
235 *                       its elements.
236 *
237 *                       Corresponding character strings, wcsunits_types[] and
238 *                       wcsunits_units[], are predefined to describe each
239 *                       quantity and its canonical units.
240 *
241 *   err       struct wcserr **
242 *                       If enabled, for function return values > 1, this
243 *                       struct will contain a detailed error message, see
244 *                       wcserr_enable().  May be NULL if an error message is
245 *                       not desired.  Otherwise, the user is responsible for
246 *                       deleting the memory allocated for the wcserr struct.
247 *
248 * Function return value:
249 *             int       Status return value:
250 *                         0: Success.
251 *                         1: Invalid numeric multiplier.
252 *                         2: Dangling binary operator.
253 *                         3: Invalid symbol in INITIAL context.
254 *                         4: Function in invalid context.
255 *                         5: Invalid symbol in EXPON context.
256 *                         6: Unbalanced bracket.
257 *                         7: Unbalanced parenthesis.
```

```
258 *                              8: Consecutive binary operators.
259 *                              9: Internal parser error.
260 *
261 *                     scale and units[] are zeroed on return if an error
262 *                     occurs.
263 *
264 * Notes:
265 *   1: wcsulexe() is permissive in accepting whitespace in all contexts in a
266 *      units specification where it does not create ambiguity (e.g. not
267 *      between a metric prefix and a basic unit string), including in strings
268 *      like "log (m ** 2)" which is formally disallowed.
269 *
270 *   2: Supported extensions:
271 *      - "angstrom" (OGIP usage) is allowed in addition to "Angstrom".
272 *      - "ohm"      (OGIP usage) is allowed in addition to "Ohm".
273 *      - "Byte"   (common usage) is allowed in addition to "byte".
274 *
275 *   3: Table 6 of WCS Paper I lists eleven units for which metric prefixes are
276 *      allowed.  However, in this implementation only prefixes greater than
277 *      unity are allowed for "a" (annum), "yr" (year), "pc" (parsec), "bit",
278 *      and "byte", and only prefixes less than unity are allowed for "mag"
279 *      (stellar magnitude).
280 *
281 *      Metric prefix "P" (peta) is specifically forbidden for "a" (annum) to
282 *      avoid confusion with "Pa" (Pascal, not peta-annum).  Note that metric
283 *      prefixes are specifically disallowed for "h" (hour) and "d" (day) so
284 *      that "ph" (photons) cannot be interpreted as pico-hours, nor "cd"
285 *      (candela) as centi-days.
286 *
287 *   4: Function types log(), ln() and exp() may only occur at the start of the
288 *      units specification.  The scale and units[] returned for these refers
289 *      to the string inside the function "argument", e.g. to "MHz" in log(MHz)
290 *      for which a scale of 1e6 will be returned.
291 *
292 *
293 * Global variable: const char *wcsunits_errmsg[] - Status return messages
294 * -------------------------------------------------------------------------
295 * Error messages to match the status value returned from each function.
296 *
297 *
298 * Global variable: const char *wcsunits_types[] - Names of physical quantities
299 * -----------------------------------------------------------------------------
300 * Names for physical quantities to match the units vector returned by
301 * wcsulexe():
302 *   -  0: plane angle
303 *   -  1: solid angle
304 *   -  2: charge
305 *   -  3: mole
306 *   -  4: temperature
307 *   -  5: luminous intensity
308 *   -  6: mass
309 *   -  7: length
310 *   -  8: time
311 *   -  9: beam
312 *   - 10: bin
313 *   - 11: bit
314 *   - 12: count
315 *   - 13: stellar magnitude
316 *   - 14: pixel
317 *   - 15: solar ratio
318 *   - 16: voxel
319 *
320 *
321 * Global variable: const char *wcsunits_units[] - Names of units
322 * -------------------------------------------------------------------
323 * Names for the units (SI) to match the units vector returned by wcsulexe():
324 *   -  0: degree
325 *   -  1: steradian
326 *   -  2: Coulomb
327 *   -  3: mole
328 *   -  4: Kelvin
329 *   -  5: candela
330 *   -  6: kilogram
331 *   -  7: metre
332 *   -  8: second
333 *
334 * The remainder are dimensionless.
335 *===========================================================================*/
336
337 #ifndef WCSLIB_WCSUNITS
338 #define WCSLIB_WCSUNITS
339
340 #include "wcserr.h"
341
342 #ifdef __cplusplus
343 extern "C" {
344 #endif
```

```
345
346
347  extern const char *wcsunits_errmsg[];
348
349  enum wcsunits_errmsg_enum {
350    UNITSERR_SUCCESS            =  0,     // Success.
351    UNITSERR_BAD_NUM_MULTIPLIER =  1,  // Invalid numeric multiplier.
352    UNITSERR_DANGLING_BINOP     =  2,       // Dangling binary operator.
353    UNITSERR_BAD_INITIAL_SYMBOL =  3,  // Invalid symbol in INITIAL context.
354    UNITSERR_FUNCTION_CONTEXT   =  4,     // Function in invalid context.
355    UNITSERR_BAD_EXPON_SYMBOL   =  5,     // Invalid symbol in EXPON context.
356    UNITSERR_UNBAL_BRACKET      =  6,        // Unbalanced bracket.
357    UNITSERR_UNBAL_PAREN        =  7, // Unbalanced parenthesis.
358    UNITSERR_CONSEC_BINOPS      =  8,        // Consecutive binary operators.
359    UNITSERR_PARSER_ERROR       =  9,         // Internal parser error.
360    UNITSERR_BAD_UNIT_SPEC      = 10,        // Non-conformant unit specifications.
361    UNITSERR_BAD_FUNCS          = 11,    // Non-conformant functions.
362    UNITSERR_UNSAFE_TRANS       = 12 // Potentially unsafe translation.
363  };
364
365  extern const char *wcsunits_types[];
366  extern const char *wcsunits_units[];
367
368  #define WCSUNITS_PLANE_ANGLE 0
369  #define WCSUNITS_SOLID_ANGLE 1
370  #define WCSUNITS_CHARGE       2
371  #define WCSUNITS_MOLE         3
372  #define WCSUNITS_TEMPERATURE 4
373  #define WCSUNITS_LUMINTEN     5
374  #define WCSUNITS_MASS         6
375  #define WCSUNITS_LENGTH       7
376  #define WCSUNITS_TIME         8
377  #define WCSUNITS_BEAM         9
378  #define WCSUNITS_BIN         10
379  #define WCSUNITS_BIT         11
380  #define WCSUNITS_COUNT       12
381  #define WCSUNITS_MAGNITUDE   13
382  #define WCSUNITS_PIXEL       14
383  #define WCSUNITS_SOLRATIO    15
384  #define WCSUNITS_VOXEL       16
385
386  #define WCSUNITS_NTYPE       17
387
388
389  int wcsunitse(const char have[], const char want[], double *scale,
390               double *offset, double *power, struct wcserr **err);
391
392  int wcsutrne(int ctrl, char unitstr[], struct wcserr **err);
393
394  int wcsulexe(const char unitstr[], int *func, double *scale,
395               double units[WCSUNITS_NTYPE], struct wcserr **err);
396
397  // Deprecated.
398  int wcsunits(const char have[], const char want[], double *scale,
399               double *offset, double *power);
400  int wcsutrn(int ctrl, char unitstr[]);
401  int wcsulex(const char unitstr[], int *func, double *scale,
402               double units[WCSUNITS_NTYPE]);
403
404  #ifdef __cplusplus
405  }
406  #endif
407
408  #endif // WCSLIB_WCSUNITS
```

19.39   wcsutil.h File Reference

Functions

- void wcsdealloc (void ∗ptr)

    free memory allocated by WCSLIB functions.
- void wcsutil_strcvt (int n, char c, int nt, const char src[ ], char dst[ ])

    Copy character string with padding.
- void wcsutil_blank_fill (int n, char c[ ])

    Fill a character string with blanks.
- void wcsutil_null_fill (int n, char c[ ])

Fill a character string with NULLs.

- int wcsutil_all_ival (int nelem, int ival, const int iarr[ ])

    Test if all elements an int array have a given value.

- int wcsutil_all_dval (int nelem, double dval, const double darr[ ])

    Test if all elements a double array have a given value.

- int wcsutil_all_sval (int nelem, const char ∗sval, const char(∗sarr)[72])

    Test if all elements a string array have a given value.

- int wcsutil_allEq (int nvec, int nelem, const double ∗first)

    Test for equality of a particular vector element.

- int wcsutil_dblEq (int nelem, double tol, const double ∗arr1, const double ∗arr2)

    Test for equality of two arrays of type double.

- int wcsutil_intEq (int nelem, const int ∗arr1, const int ∗arr2)

    Test for equality of two arrays of type int.

- int wcsutil_strEq (int nelem, char(∗arr1)[72], char(∗arr2)[72])

    Test for equality of two string arrays.

- void wcsutil_setAll (int nvec, int nelem, double ∗first)

    Set a particular vector element.

- void wcsutil_setAli (int nvec, int nelem, int ∗first)

    Set a particular vector element.

- void wcsutil_setBit (int nelem, const int ∗sel, int bits, int ∗array)

    Set bits in selected elements of an array.

- char ∗ wcsutil_fptr2str (void(∗fptr)(void), char hext[19])

    Translate pointer-to-function to string.

- void wcsutil_double2str (char ∗buf, const char ∗format, double value)

    Translate double to string ignoring the locale.

- int wcsutil_str2double (const char ∗buf, double ∗value)

    Translate string to a double, ignoring the locale.

- int wcsutil_str2double2 (const char ∗buf, double ∗value)

    Translate string to doubles, ignoring the locale.

19.39.1 Detailed Description

Simple utility functions. With the exception of wcsdealloc(), these functions are intended for **internal use only** by WCSLIB.

The internal-use functions are documented here solely as an aid to understanding the code. They are not intended for external use - the API may change without notice!

19.39.2 Function Documentation

19.39.2.1 wcsdealloc() `void wcsdealloc (`
           `void ∗ ptr )`

wcsdealloc() invokes the free() system routine to free memory. Specifically, it is intended to free memory allocated (using calloc()) by certain WCSLIB functions (e.g. wcshdo(), wcsfixi(), fitshdr()), which it is the user's responsibility to deallocate.

In certain situations, for example multithreading, it may be important that this be done within the WCSLIB sharable library's runtime environment.

PLEASE NOTE: wcsdealloc() must not be used in place of the destructors for particular structs, such as wcsfree(), celfree(), etc.

Parameters

in,out	*ptr*	Address of the allocated memory.

Returns

19.39.2.2 wcsutil_strcvt() `void wcsutil_strcvt (`
            `int n,`
            `char c,`
            `int nt,`
            `const char src[],`
            `char dst[] )`

INTERNAL USE ONLY.

wcsutil_strcvt() copies one character string to another up to the specified maximum number of characters.

If the given string is null-terminated, then the NULL character copied to the returned string, and all characters following it up to the specified maximum, are replaced with the specified substitute character, either blank or NULL.

If the source string is not null-terminated and the substitute character is blank, then copy the maximum number of characters and do nothing further. However, if the substitute character is NULL, then the last character and all consecutive blank characters preceding it will be replaced with NULLs.

Used by the Fortran wrapper functions in translating C strings into Fortran CHARACTER variables and vice versa.

Parameters

in	*n*	Maximum number of characters to copy.
in	*c*	Substitute character, either NULL or blank (anything other than NULL).
in	*nt*	If true, then dst is of length n+1, with the last character always set to NULL.
in	*src*	Character string to be copied. If null-terminated, then need not be of length n, otherwise it must be.
out	*dst*	Destination character string, which must be long enough to hold n characters. Note that this string will not be null-terminated if the substitute character is blank.

Returns

19.39.2.3 wcsutil_blank_fill() `void wcsutil_blank_fill (`
            `int n,`
            `char c[] )`

INTERNAL USE ONLY.

wcsutil_blank_fill() pads a character sub-string with blanks starting with the terminating NULL character (if any).

Parameters

in		*n*	Length of the sub-string.
in,out		*c*	The character sub-string, which will not be null-terminated on return.

Returns

19.39.2.4 wcsutil_null_fill() `void wcsutil_null_fill (`
`            int n,`
`            char c[] )`

INTERNAL USE ONLY.

wcsutil_null_fill() strips trailing blanks from a string (or sub-string) and propagates the terminating NULL character (if any) to the end of the string.

If the string is not null-terminated, then the last character and all consecutive blank characters preceding it will be replaced with NULLs.

Mainly used in the C library to strip trailing blanks from FITS keyvalues. Also used to make character strings intelligible in the GNU debugger, which prints the rubbish following the terminating NULL character, thereby obscuring the valid part of the string.

Parameters

in		*n*	Number of characters.
in,out		*c*	The character (sub-)string.

Returns

19.39.2.5 wcsutil_all_ival() `int wcsutil_all_ival (`
`            int nelem,`
`            int ival,`
`            const int iarr[] )`

INTERNAL USE ONLY.

wcsutil_all_ival() tests whether all elements of an array of type int all have the specified value.

Parameters

in	*nelem*	The length of the array.
in	*ival*	Value to be tested.
in	*iarr*	Pointer to the first element of the array.

Returns

Status return value:

- 0: Not all equal.
- 1: All equal.

19.39.2.6 wcsutil_all_dval() `int wcsutil_all_dval (`
         `int nelem,`
         `double dval,`
         `const double darr[] )`

INTERNAL USE ONLY.

wcsutil_all_dval() tests whether all elements of an array of type double all have the specified value.

Parameters

in	*nelem*	The length of the array.
in	*dval*	Value to be tested.
in	*darr*	Pointer to the first element of the array.

Returns

Status return value:

- 0: Not all equal.
- 1: All equal.

19.39.2.7 wcsutil_all_sval() `int wcsutil_all_sval (`
         `int nelem,`
         `const char ∗ sval,`
         `const char(∗) sarr[72] )`

INTERNAL USE ONLY.

wcsutil_all_sval() tests whether the elements of an array of type char (∗)[72] all have the specified value.

Parameters

in	*nelem*	The length of the array.
in	*sval*	String to be tested.
in	*sarr*	Pointer to the first element of the array.

Returns

Status return value:

- 0: Not all equal.

- 1: All equal.

19.39.2.8 wcsutil_allEq() `int wcsutil_allEq (`
              `int nvec,`
              `int nelem,`
              `const double * first )`

INTERNAL USE ONLY.

wcsutil_allEq() tests for equality of a particular element in a set of vectors.

Parameters

in	*nvec*	The number of vectors.
in	*nelem*	The length of each vector.
in	*first*	Pointer to the first element to test in the array. The elements tested for equality are `*first == *(first + nelem) == *(first + nelem*2)` `:` `== *(first + nelem*(nvec-1));` The array might be dimensioned as `double v[nvec][nelem];`

Returns

Status return value:

- 0: Not all equal.

- 1: All equal.

19.39.2.9 wcsutil_dblEq() `int wcsutil_dblEq (`
              `int nelem,`
              `double tol,`
              `const double * arr1,`
              `const double * arr2 )`

INTERNAL USE ONLY.

wcsutil_dblEq() tests for equality of two double-precision arrays.

Parameters

in	*nelem*	The number of elements in each array.
in	*tol*	Tolerance for comparison of the floating-point values. For example, for tol == 1e-6, all floating-point values in the arrays must be equal to the first 6 decimal places. A value of 0 implies exact equality.
in	*arr1*	The first array.
in	*arr2*	The second array

Returns

Status return value:

- 0: Not equal.
- 1: Equal.

19.39.2.10   wcsutil_intEq() `int wcsutil_intEq (`
        `int nelem,`
        `const int * arr1,`
        `const int * arr2 )`

INTERNAL USE ONLY.

wcsutil_intEq() tests for equality of two int arrays.

Parameters

in	*nelem*	The number of elements in each array.
in	*arr1*	The first array.
in	*arr2*	The second array

Returns

Status return value:

- 0: Not equal.
- 1: Equal.

19.39.2.11   wcsutil_strEq() `int wcsutil_strEq (`
        `int nelem,`
        `char(*) arr1[72],`
        `char(*) arr2[72] )`

INTERNAL USE ONLY.

wcsutil_strEq() tests for equality of two string arrays.

Parameters

in	*nelem*	The number of elements in each array.
in	*arr1*	The first array.
in	*arr2*	The second array

Returns

Status return value:

- 0: Not equal.
- 1: Equal.

19.39.2.12 wcsutil_setAll() `void wcsutil_setAll (`
         `int nvec,`
         `int nelem,`
         `double * first )`

INTERNAL USE ONLY.

wcsutil_setAll() sets the value of a particular element in a set of vectors of type double.

Parameters

`in`	*nvec*	The number of vectors.
`in`	*nelem*	The length of each vector.

`in,out`	*first*	Pointer to the first element in the array, the value of which is used to set the others
		`*(first + nelem) = *first; *(first + nelem*2) = *first;`
		  `:`
		`*(first + nelem*(nvec-1)) = *first;`
		
		The array might be dimensioned as
		`double v[nvec][nelem];`

Returns

19.39.2.13 wcsutil_setAli() `void wcsutil_setAli (`
         `int nvec,`
         `int nelem,`
         `int * first )`

INTERNAL USE ONLY.

wcsutil_setAli() sets the value of a particular element in a set of vectors of type int.

Parameters

`in`	*nvec*	The number of vectors.
`in`	*nelem*	The length of each vector.

`in,out`	*first*	Pointer to the first element in the array, the value of which is used to set the others
		`*(first + nelem) = *first; *(first + nelem*2) = *first;`
		  `:`
		`*(first + nelem*(nvec-1)) = *first;`
		
		The array might be dimensioned as
		`int v[nvec][nelem];`

Returns

19.39.2.14 wcsutil_setBit() `void wcsutil_setBit (`
> `int` *`nelem,`*
> `const int * ` *`sel,`*
> `int` *`bits,`*
> `int * ` *`array` *`)`

INTERNAL USE ONLY.

wcsutil_setBit() sets bits in selected elements of an array.

Parameters

in	*nelem*	Number of elements in the array.
in	*sel*	Address of a selection array of length nelem. May be specified as the null pointer in which case all elements are selected.
in	*bits*	Bit mask.
in,out	*array*	Address of the array of length nelem.

Returns

19.39.2.15 wcsutil_fptr2str() `char * wcsutil_fptr2str (`
> `void(*)(void) ` *`fptr,`*
> `char ` *`hext[19]` *`)`

INTERNAL USE ONLY.

wcsutil_fptr2str() translates a pointer-to-function to hexadecimal string representation for output. It is used by the various routines that print the contents of WCSLIB structs, noting that it is not strictly legal to type-pun a function pointer to void∗. See `http://stackoverflow.com/questions/2741683/how-to-format-a-function-point`

Parameters

in	*fptr*	
out	*hext*	Null-terminated string. Should be at least 19 bytes in size to accomodate a 64-bit address (16 bytes in hex), plus the leading "0x" and trailing '\0'.

Returns

The address of hext.

19.39.2.16 wcsutil_double2str() `void wcsutil_double2str (`
    `char * buf,`
    `const char * format,`
    `double value )`

INTERNAL USE ONLY.

wcsutil_double2str() converts a double to a string, but unlike `sprintf()` it ignores the locale and always uses a '.' as the decimal separator. Also, unless it includes an exponent, the formatted value will always have a fractional part, ".0" being appended if necessary.

Parameters

out	*buf*	The buffer to write the string into.
in	*format*	The formatting directive, such as "f". This may be any of the forms accepted by `sprintf()`, but should only include a formatting directive and nothing else. For "g" and "G" formats, unless it includes an exponent, the formatted value will always have a fractional part, ".0" being appended if necessary.
in	*value*	The value to convert to a string.

19.39.2.17 wcsutil_str2double() `int wcsutil_str2double (`
    `const char * buf,`
    `double * value )`

INTERNAL USE ONLY.

wcsutil_str2double() converts a string to a double, but unlike `sscanf()` it ignores the locale and always expects a '.' as the decimal separator.

Parameters

in	*buf*	The string containing the value
out	*value*	The double value parsed from the string.

19.39.2.18 wcsutil_str2double2() `int wcsutil_str2double2 (`
    `const char * buf,`
    `double * value )`

INTERNAL USE ONLY.

wcsutil_str2double2() converts a string to a pair of doubles containing the integer and fractional parts. Unlike `sscanf()` it ignores the locale and always expects a '.' as the decimal separator.

Parameters

in	*buf*	The string containing the value
out	*value*	parts, parsed from the string.

19.40 wcsutil.h

```
1 /*============================================================================
2   WCSLIB 7.11 - an implementation of the FITS WCS standard.
3   Copyright (C) 1995-2022, Mark Calabretta
4
5   This file is part of WCSLIB.
6
7   WCSLIB is free software: you can redistribute it and/or modify it under the
8   terms of the GNU Lesser General Public License as published by the Free
9   Software Foundation, either version 3 of the License, or (at your option)
10  any later version.
11
12  WCSLIB is distributed in the hope that it will be useful, but WITHOUT ANY
13  WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
14  FOR A PARTICULAR PURPOSE.  See the GNU Lesser General Public License for
15  more details.
16
17  You should have received a copy of the GNU Lesser General Public License
18  along with WCSLIB.  If not, see http://www.gnu.org/licenses.
19
20  Author: Mark Calabretta, Australia Telescope National Facility, CSIRO.
21  http://www.atnf.csiro.au/people/Mark.Calabretta
22  $Id: wcsutil.h,v 7.11 2022/04/26 06:13:52 mcalabre Exp $
23 *=============================================================================
24 *
25 * WCSLIB 7.11 - C routines that implement the FITS World Coordinate System
26 * (WCS) standard.  Refer to the README file provided with WCSLIB for an
27 * overview of the library.
28 *
29 *
30 * Summary of the wcsutil routines
31 * -------------------------------
32 * Simple utility functions.  With the exception of wcsdealloc(), these
33 * functions are intended for internal use only by WCSLIB.
34 *
35 * The internal-use functions are documented here solely as an aid to
36 * understanding the code.  They are not intended for external use - the API
37 * may change without notice!
38 *
39 *
40 * wcsdealloc() - free memory allocated by WCSLIB functions
41 * --------------------------------------------------------
42 * wcsdealloc() invokes the free() system routine to free memory.
43 * Specifically, it is intended to free memory allocated (using calloc()) by
44 * certain WCSLIB functions (e.g. wcshdo(), wcsfixi(), fitshdr()), which it is
45 * the user's responsibility to deallocate.
46 *
47 * In certain situations, for example multithreading, it may be important that
48 * this be done within the WCSLIB sharable library's runtime environment.
49 *
50 * PLEASE NOTE: wcsdealloc() must not be used in place of the destructors for
51 * particular structs, such as wcsfree(), celfree(), etc.
52 *
53 * Given and returned:
54 *   ptr       void*     Address of the allocated memory.
55 *
56 * Function return value:
57 *             void
58 *
59 *
60 * wcsutil_strcvt() - Copy character string with padding
61 * -----------------------------------------------------
62 * INTERNAL USE ONLY.
63 *
64 * wcsutil_strcvt() copies one character string to another up to the specified
65 * maximum number of characters.
66 *
67 * If the given string is null-terminated, then the NULL character copied to
68 * the returned string, and all characters following it up to the specified
69 * maximum, are replaced with the specified substitute character, either blank
70 * or NULL.
71 *
72 * If the source string is not null-terminated and the substitute character is
73 * blank, then copy the maximum number of characters and do nothing further.
74 * However, if the substitute character is NULL, then the last character and
75 * all consecutive blank characters preceding it will be replaced with NULLs.
76 *
77 * Used by the Fortran wrapper functions in translating C strings into Fortran
78 * CHARACTER variables and vice versa.
79 *
80 * Given:
81 *   n         int       Maximum number of characters to copy.
82 *
83 *   c         char      Substitute character, either NULL or blank (anything
```

```
 84 *                      other than NULL).
 85 *
 86 *   nt       int       If true, then dst is of length n+1, with the last
 87 *                      character always set to NULL.
 88 *
 89 *   src      char[]    Character string to be copied.  If null-terminated,
 90 *                      then need not be of length n, otherwise it must be.
 91 *
 92 * Returned:
 93 *   dst      char[]    Destination character string, which must be long
 94 *                      enough to hold n characters.  Note that this string
 95 *                      will not be null-terminated if the substitute
 96 *                      character is blank.
 97 *
 98 * Function return value:
 99 *            void
100 *
101 *
102 * wcsutil_blank_fill() - Fill a character string with blanks
103 * ---------------------------------------------------------
104 * INTERNAL USE ONLY.
105 *
106 * wcsutil_blank_fill() pads a character sub-string with blanks starting with
107 * the terminating NULL character (if any).
108 *
109 * Given:
110 *   n        int       Length of the sub-string.
111 *
112 * Given and returned:
113 *   c        char[]    The character sub-string, which will not be
114 *                      null-terminated on return.
115 *
116 * Function return value:
117 *            void
118 *
119 *
120 * wcsutil_null_fill() - Fill a character string with NULLs
121 * -------------------------------------------------------
122 * INTERNAL USE ONLY.
123 *
124 * wcsutil_null_fill() strips trailing blanks from a string (or sub-string) and
125 * propagates the terminating NULL character (if any) to the end of the string.
126 *
127 * If the string is not null-terminated, then the last character and all
128 * consecutive blank characters preceding it will be replaced with NULLs.
129 *
130 * Mainly used in the C library to strip trailing blanks from FITS keyvalues.
131 * Also used to make character strings intelligible in the GNU debugger, which
132 * prints the rubbish following the terminating NULL character, thereby
133 * obscuring the valid part of the string.
134 *
135 * Given:
136 *   n        int       Number of characters.
137 *
138 * Given and returned:
139 *   c        char[]    The character (sub-)string.
140 *
141 * Function return value:
142 *            void
143 *
144 *
145 * wcsutil_all_ival() - Test if all elements an int array have a given value
146 * ------------------------------------------------------------------------
147 * INTERNAL USE ONLY.
148 *
149 * wcsutil_all_ival() tests whether all elements of an array of type int all
150 * have the specified value.
151 *
152 * Given:
153 *   nelem    int       The length of the array.
154 *
155 *   ival     int       Value to be tested.
156 *
157 *   iarr     const int[]
158 *                      Pointer to the first element of the array.
159 *
160 * Function return value:
161 *            int       Status return value:
162 *                        0: Not all equal.
163 *                        1: All equal.
164 *
165 *
166 * wcsutil_all_dval() - Test if all elements a double array have a given value
167 * --------------------------------------------------------------------------
168 * INTERNAL USE ONLY.
169 *
170 * wcsutil_all_dval() tests whether all elements of an array of type double all
```

```
171 * have the specified value.
172 *
173 * Given:
174 *   nelem     int       The length of the array.
175 *
176 *   dval      int       Value to be tested.
177 *
178 *   darr      const double[]
179 *                       Pointer to the first element of the array.
180 *
181 * Function return value:
182 *             int       Status return value:
183 *                         0: Not all equal.
184 *                         1: All equal.
185 *
186 *
187 * wcsutil_all_sval() - Test if all elements a string array have a given value
188 * ---------------------------------------------------------------------------
189 * INTERNAL USE ONLY.
190 *
191 * wcsutil_all_sval() tests whether the elements of an array of type
192 * char (*)[72] all have the specified value.
193 *
194 * Given:
195 *   nelem     int       The length of the array.
196 *
197 *   sval      const char *
198 *                       String to be tested.
199 *
200 *   sarr      const char (*)[72]
201 *                       Pointer to the first element of the array.
202 *
203 * Function return value:
204 *             int       Status return value:
205 *                         0: Not all equal.
206 *                         1: All equal.
207 *
208 *
209 * wcsutil_allEq() - Test for equality of a particular vector element
210 * ------------------------------------------------------------------
211 * INTERNAL USE ONLY.
212 *
213 * wcsutil_allEq() tests for equality of a particular element in a set of
214 * vectors.
215 *
216 * Given:
217 *   nvec      int       The number of vectors.
218 *
219 *   nelem     int       The length of each vector.
220 *
221 *   first     const double*
222 *                       Pointer to the first element to test in the array.
223 *                       The elements tested for equality are
224 *
225 =                         *first == *(first + nelem)
226 =                                == *(first + nelem*2)
227 =                                            :
228 =                                == *(first + nelem*(nvec-1));
229 *
230 *                       The array might be dimensioned as
231 *
232 =                         double v[nvec][nelem];
233 *
234 * Function return value:
235 *             int       Status return value:
236 *                         0: Not all equal.
237 *                         1: All equal.
238 *
239 *
240 * wcsutil_dblEq() - Test for equality of two arrays of type double
241 * ----------------------------------------------------------------
242 * INTERNAL USE ONLY.
243 *
244 * wcsutil_dblEq() tests for equality of two double-precision arrays.
245 *
246 * Given:
247 *   nelem     int       The number of elements in each array.
248 *
249 *   tol       double    Tolerance for comparison of the floating-point values.
250 *                       For example, for tol == 1e-6, all floating-point
251 *                       values in the arrays must be equal to the first 6
252 *                       decimal places.  A value of 0 implies exact equality.
253 *
254 *   arr1      const double*
255 *                       The first array.
256 *
257 *   arr2      const double*
```

```
258 *                      The second array
259 *
260 * Function return value:
261 *            int        Status return value:
262 *                         0: Not equal.
263 *                         1: Equal.
264 *
265 *
266 * wcsutil_intEq() - Test for equality of two arrays of type int
267 * -------------------------------------------------------------
268 * INTERNAL USE ONLY.
269 *
270 * wcsutil_intEq() tests for equality of two int arrays.
271 *
272 * Given:
273 *   nelem      int       The number of elements in each array.
274 *
275 *   arr1       const int*
276 *                        The first array.
277 *
278 *   arr2       const int*
279 *                        The second array
280 *
281 * Function return value:
282 *            int        Status return value:
283 *                         0: Not equal.
284 *                         1: Equal.
285 *
286 *
287 * wcsutil_strEq() - Test for equality of two string arrays
288 * --------------------------------------------------------
289 * INTERNAL USE ONLY.
290 *
291 * wcsutil_strEq() tests for equality of two string arrays.
292 *
293 * Given:
294 *   nelem      int       The number of elements in each array.
295 *
296 *   arr1       const char**
297 *                        The first array.
298 *
299 *   arr2       const char**
300 *                        The second array
301 *
302 * Function return value:
303 *            int        Status return value:
304 *                         0: Not equal.
305 *                         1: Equal.
306 *
307 *
308 * wcsutil_setAll() - Set a particular vector element
309 * --------------------------------------------------
310 * INTERNAL USE ONLY.
311 *
312 * wcsutil_setAll() sets the value of a particular element in a set of vectors
313 * of type double.
314 *
315 * Given:
316 *   nvec       int       The number of vectors.
317 *
318 *   nelem      int       The length of each vector.
319 *
320 * Given and returned:
321 *   first      double*   Pointer to the first element in the array, the value
322 *                        of which is used to set the others
323 *
324 =                            *(first + nelem) = *first;
325 =                            *(first + nelem*2) = *first;
326 =                                    :
327 =                            *(first + nelem*(nvec-1)) = *first;
328 *
329 *                        The array might be dimensioned as
330 *
331 =                            double v[nvec][nelem];
332 *
333 * Function return value:
334 *            void
335 *
336 *
337 * wcsutil_setAli() - Set a particular vector element
338 * --------------------------------------------------
339 * INTERNAL USE ONLY.
340 *
341 * wcsutil_setAli() sets the value of a particular element in a set of vectors
342 * of type int.
343 *
344 * Given:
```

```
345 *    nvec       int       The number of vectors.
346 *
347 *    nelem      int       The length of each vector.
348 *
349 * Given and returned:
350 *    first      int*      Pointer to the first element in the array, the value
351 *                         of which is used to set the others
352 *
353 =                             *(first + nelem) = *first;
354 =                             *(first + nelem*2) = *first;
355 =                                     :
356 =                             *(first + nelem*(nvec-1)) = *first;
357 *
358 *                         The array might be dimensioned as
359 *
360 =                             int v[nvec][nelem];
361 *
362 * Function return value:
363 *               void
364 *
365 *
366 * wcsutil_setBit() - Set bits in selected elements of an array
367 * ------------------------------------------------------------
368 * INTERNAL USE ONLY.
369 *
370 * wcsutil_setBit() sets bits in selected elements of an array.
371 *
372 * Given:
373 *    nelem      int       Number of elements in the array.
374 *
375 *    sel        const int*
376 *                         Address of a selection array of length nelem.  May
377 *                         be specified as the null pointer in which case all
378 *                         elements are selected.
379 *
380 *    bits       int       Bit mask.
381 *
382 * Given and returned:
383 *    array      int*      Address of the array of length nelem.
384 *
385 * Function return value:
386 *               void
387 *
388 *
389 * wcsutil_fptr2str() - Translate pointer-to-function to string
390 * ------------------------------------------------------------
391 * INTERNAL USE ONLY.
392 *
393 * wcsutil_fptr2str() translates a pointer-to-function to hexadecimal string
394 * representation for output.  It is used by the various routines that print
395 * the contents of WCSLIB structs, noting that it is not strictly legal to
396 * type-pun a function pointer to void*.  See
397 * http://stackoverflow.com/questions/2741683/how-to-format-a-function-pointer
398 *
399 * Given:
400 *    fptr       void(*)() Pointer to function.
401 *
402 * Returned:
403 *    hext       char[19]  Null-terminated string.  Should be at least 19 bytes
404 *                         in size to accomodate a 64-bit address (16 bytes in
405 *                         hex), plus the leading "0x" and trailing '\0'.
406 *
407 * Function return value:
408 *               char *    The address of hext.
409 *
410 *
411 * wcsutil_double2str() - Translate double to string ignoring the locale
412 * ---------------------------------------------------------------------
413 * INTERNAL USE ONLY.
414 *
415 * wcsutil_double2str() converts a double to a string, but unlike sprintf() it
416 * ignores the locale and always uses a '.' as the decimal separator.  Also,
417 * unless it includes an exponent, the formatted value will always have a
418 * fractional part, ".0" being appended if necessary.
419 *
420 * Returned:
421 *    buf        char *    The buffer to write the string into.
422 *
423 * Given:
424 *    format     char *    The formatting directive, such as "%f".  This
425 *                         may be any of the forms accepted by sprintf(), but
426 *                         should only include a formatting directive and
427 *                         nothing else.  For "%g" and "%G" formats, unless it
428 *                         includes an exponent, the formatted value will always
429 *                         have a fractional part, ".0" being appended if
430 *                         necessary.
431 *
```

```
432 *   value    double   The value to convert to a string.
433 *
434 *
435 * wcsutil_str2double() - Translate string to a double, ignoring the locale
436 * -------------------------------------------------------------------------
437 * INTERNAL USE ONLY.
438 *
439 * wcsutil_str2double() converts a string to a double, but unlike sscanf() it
440 * ignores the locale and always expects a '.' as the decimal separator.
441 *
442 * Given:
443 *   buf      char *   The string containing the value
444 *
445 * Returned:
446 *   value    double *  The double value parsed from the string.
447 *
448 *
449 * wcsutil_str2double2() - Translate string to doubles, ignoring the locale
450 * -------------------------------------------------------------------------
451 * INTERNAL USE ONLY.
452 *
453 * wcsutil_str2double2() converts a string to a pair of doubles containing the
454 * integer and fractional parts.  Unlike sscanf() it ignores the locale and
455 * always expects a '.' as the decimal separator.
456 *
457 * Given:
458 *   buf      char *   The string containing the value
459 *
460 * Returned:
461 *   value    double[2] The double value, split into integer and fractional
462 *                      parts, parsed from the string.
463 *
464 *===========================================================================*/
465
466 #ifndef WCSLIB_WCSUTIL
467 #define WCSLIB_WCSUTIL
468
469 #ifdef __cplusplus
470 extern "C" {
471 #endif
472
473 void wcsdealloc(void *ptr);
474
475 void wcsutil_strcvt(int n, char c, int nt, const char src[], char dst[]);
476
477 void wcsutil_blank_fill(int n, char c[]);
478 void wcsutil_null_fill (int n, char c[]);
479
480 int  wcsutil_all_ival(int nelem, int ival, const int iarr[]);
481 int  wcsutil_all_dval(int nelem, double dval, const double darr[]);
482 int  wcsutil_all_sval(int nelem, const char *sval, const char (*sarr)[72]);
483 int  wcsutil_allEq (int nvec, int nelem, const double *first);
484
485 int  wcsutil_dblEq(int nelem, double tol, const double *arr1,
486                    const double *arr2);
487 int  wcsutil_intEq(int nelem, const int *arr1, const int *arr2);
488 int  wcsutil_strEq(int nelem, char (*arr1)[72], char (*arr2)[72]);
489 void wcsutil_setAll(int nvec, int nelem, double *first);
490 void wcsutil_setAli(int nvec, int nelem, int *first);
491 void wcsutil_setBit(int nelem, const int *sel, int bits, int *array);
492 char *wcsutil_fptr2str(void (*fptr)(void), char hext[19]);
493 void wcsutil_double2str(char *buf, const char *format, double value);
494 int  wcsutil_str2double(const char *buf, double *value);
495 int  wcsutil_str2double2(const char *buf, double *value);
496
497 #ifdef __cplusplus
498 }
499 #endif
500
501 #endif // WCSLIB_WCSUTIL
```

19.41   wtbarr.h File Reference

Data Structures

- struct wtbarr

    Extraction of coordinate lookup tables from BINTABLE.

19.41.1 Detailed Description

The wtbarr struct is used by wcstab() in extracting coordinate lookup tables from a binary table extension (BINTABLE) and copying them into the tabprm structs stored in wcsprm.

19.42 wtbarr.h

Go to the documentation of this file.
```
1  /*============================================================================
2    WCSLIB 7.11 - an implementation of the FITS WCS standard.
3    Copyright (C) 1995-2022, Mark Calabretta
4
5    This file is part of WCSLIB.
6
7    WCSLIB is free software: you can redistribute it and/or modify it under the
8    terms of the GNU Lesser General Public License as published by the Free
9    Software Foundation, either version 3 of the License, or (at your option)
10   any later version.
11
12   WCSLIB is distributed in the hope that it will be useful, but WITHOUT ANY
13   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
14   FOR A PARTICULAR PURPOSE.  See the GNU Lesser General Public License for
15   more details.
16
17   You should have received a copy of the GNU Lesser General Public License
18   along with WCSLIB.  If not, see http://www.gnu.org/licenses.
19
20   Author: Mark Calabretta, Australia Telescope National Facility, CSIRO.
21   http://www.atnf.csiro.au/people/Mark.Calabretta
22   $Id: wtbarr.h,v 7.11 2022/04/26 06:13:52 mcalabre Exp $
23  *============================================================================
24  *
25  * WCSLIB 7.11 - C routines that implement the FITS World Coordinate System
26  * (WCS) standard.  Refer to the README file provided with WCSLIB for an
27  * overview of the library.
28  *
29  *
30  * Summary of the wtbarr struct
31  * ----------------------------
32  * The wtbarr struct is used by wcstab() in extracting coordinate lookup tables
33  * from a binary table extension (BINTABLE) and copying them into the tabprm
34  * structs stored in wcsprm.
35  *
36  *
37  * wtbarr struct - Extraction of coordinate lookup tables from BINTABLE
38  * -------------------------------------------------------------------
39  * Function wcstab(), which is invoked automatically by wcspih(), sets up an
40  * array of wtbarr structs to assist in extracting coordinate lookup tables
41  * from a binary table extension (BINTABLE) and copying them into the tabprm
42  * structs stored in wcsprm.  Refer to the usage notes for wcspih() and
43  * wcstab() in wcshdr.h, and also the prologue to tab.h.
44  *
45  * For C++ usage, because of a name space conflict with the wtbarr typedef
46  * defined in CFITSIO header fitsio.h, the wtbarr struct is renamed to wtbarr_s
47  * by preprocessor macro substitution with scope limited to wtbarr.h itself,
48  * and similarly in wcs.h.
49  *
50  *   int i
51  *     (Given) Image axis number.
52  *
53  *   int m
54  *     (Given) wcstab array axis number for index vectors.
55  *
56  *   int kind
57  *     (Given) Character identifying the wcstab array type:
58  *       - c: coordinate array,
59  *       - i: index vector.
60  *
61  *   char extnam[72]
62  *     (Given) EXTNAME identifying the binary table extension.
63  *
64  *   int extver
65  *     (Given) EXTVER identifying the binary table extension.
66  *
67  *   int extlev
68  *     (Given) EXTLEV identifying the binary table extension.
69  *
70  *   char ttype[72]
71  *     (Given) TTYPEn identifying the column of the binary table that contains
72  *     the wcstab array.
```

```
73 *
74 *    long row
75 *      (Given) Table row number.
76 *
77 *    int ndim
78 *      (Given) Expected dimensionality of the wcstab array.
79 *
80 *    int *dimlen
81 *      (Given) Address of the first element of an array of int of length ndim
82 *      into which the wcstab array axis lengths are to be written.
83 *
84 *    double **arrayp
85 *      (Given) Pointer to an array of double which is to be allocated by the
86 *      user and into which the wcstab array is to be written.
87 *
88 *===========================================================================*/
89
90 #ifndef WCSLIB_WTBARR
91 #define WCSLIB_WTBARR
92
93 #ifdef __cplusplus
94 extern "C" {
95 #define wtbarr wtbarr_s          // See prologue above.
96 #endif
97                                  // For extracting wcstab arrays.  Matches
98                                  // the wtbarr typedef defined in CFITSIO
99                                  // header fitsio.h.
100 struct wtbarr {
101   int  i;                        // Image axis number.
102   int  m;                        // Array axis number for index vectors.
103   int  kind;                       // wcstab array type.
104   char extnam[72];               // EXTNAME of binary table extension.
105   int  extver;                   // EXTVER  of binary table extension.
106   int  extlev;                   // EXTLEV  of binary table extension.
107   char ttype[72];          // TTYPEn of column containing the array.
108   long row;                  // Table row number.
109   int  ndim;                       // Expected wcstab array dimensionality.
110   int  *dimlen;                  // Where to write the array axis lengths.
111   double **arrayp;               // Where to write the address of the array
112                                  // allocated to store the wcstab array.
113 };
114
115 #ifdef __cplusplus
116 #undef wtbarr
117 }
118 #endif
119
120 #endif // WCSLIB_WTBARR
```

19.43   wcslib.h File Reference

```
#include "cel.h"
#include "dis.h"
#include "fitshdr.h"
#include "lin.h"
#include "log.h"
#include "prj.h"
#include "spc.h"
#include "sph.h"
#include "spx.h"
#include "tab.h"
#include "wcs.h"
#include "wcserr.h"
#include "wcsfix.h"
#include "wcshdr.h"
#include "wcsmath.h"
#include "wcsprintf.h"
#include "wcstrig.h"
#include "wcsunits.h"
#include "wcsutil.h"
#include "wtbarr.h"
```

19.43.1   Detailed Description

This header file is provided purely for convenience. Use it to include all of the separate WCSLIB headers.

19.44   wcslib.h

[Go to the documentation of this file.](#)
```
1  /*============================================================================
2    WCSLIB 7.11 - an implementation of the FITS WCS standard.
3    Copyright (C) 1995-2022, Mark Calabretta
4
5    This file is part of WCSLIB.
6
7    WCSLIB is free software: you can redistribute it and/or modify it under the
8    terms of the GNU Lesser General Public License as published by the Free
9    Software Foundation, either version 3 of the License, or (at your option)
10    any later version.
11
12    WCSLIB is distributed in the hope that it will be useful, but WITHOUT ANY
13    WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
14    FOR A PARTICULAR PURPOSE.  See the GNU Lesser General Public License for
15    more details.
16
17    You should have received a copy of the GNU Lesser General Public License
18    along with WCSLIB.  If not, see http://www.gnu.org/licenses.
19
20    Author: Mark Calabretta, Australia Telescope National Facility, CSIRO.
21    http://www.atnf.csiro.au/people/Mark.Calabretta
22    $Id: wcslib.h,v 7.11 2022/04/26 06:13:52 mcalabre Exp $
23  *=============================================================================
24  *
25  * WCSLIB 7.11 - C routines that implement the FITS World Coordinate System
26  * (WCS) standard.  Refer to the README file provided with WCSLIB for an
27  * overview of the library.
28  *
29  * Summary of wcslib.h
30  * -------------------
31  * This header file is provided purely for convenience.  Use it to include all
32  * of the separate WCSLIB headers.
33  *
34  *===========================================================================*/
35
36  #ifndef WCSLIB_WCSLIB
37  #define WCSLIB_WCSLIB
38
39  #include "cel.h"
40  #include "dis.h"
41  #include "fitshdr.h"
42  #include "lin.h"
43  #include "log.h"
44  #include "prj.h"
45  #include "spc.h"
46  #include "sph.h"
47  #include "spx.h"
48  #include "tab.h"
49  #include "wcs.h"
50  #include "wcserr.h"
51  #include "wcsfix.h"
52  #include "wcshdr.h"
53  #include "wcsmath.h"
54  #include "wcsprintf.h"
55  #include "wcstrig.h"
56  #include "wcsunits.h"
57  #include "wcsutil.h"
58  #include "wtbarr.h"
59
60  #endif // WCSLIB_WCSLIB
3582    wcserr_enable(1);
3583    wcsprintf_set(stderr);
3584
3585    ...
3586
3587    if (wcsset(&wcs) {
3588      wcsperr(&wcs);
3589      return wcs.err->status;
3590    }
3591  @endverbatim
3592  In this example, if an error was generated in one of the prjset() functions,
3593  wcsperr() would print an error traceback starting with wcsset(), then
3594  celset(), and finally the particular projection-setting function that
```

```
3595 generated the error.  For each of them it would print the status return value,
3596 function name, source file, line number, and an error message which may be
3597 more specific and informative than the general error messages reported in the
3598 first example.  For example, in response to a deliberately generated error,
3599 the @c twcs test program, which tests wcserr among other things, produces a
3600 traceback similar to this:
3601 @verbatim
3602 ERROR 5 in wcsset() at line 1564 of file wcs.c:
3603   Invalid parameter value.
3604 ERROR 2 in celset() at line 196 of file cel.c:
3605   Invalid projection parameters.
3606 ERROR 2 in bonset() at line 5727 of file prj.c:
3607   Invalid parameters for Bonne's projection.
3608 @endverbatim
3609
3610 Each of the @ref structs "structs" in @ref overview "WCSLIB" includes a
3611 pointer, called @a err, to a wcserr struct.  When an error occurs, a struct is
3612 allocated and error information stored in it.  The wcserr pointers and the
3613 @ref memory "memory" allocated for them are managed by the routines that
3614 manage the various structs such as wcsinit() and wcsfree().
3615
3616 wcserr messaging is an opt-in system enabled via wcserr_enable(), as in the
3617 example above.  If enabled, when an error occurs it is the user's
3618 responsibility to free the memory allocated for the error message using
3619 wcsfree(), celfree(), prjfree(), etc.  Failure to do so before the struct goes
3620 out of scope will result in memory leaks (if execution continues beyond the
3621 error).
3622 */
3623
3624
```

Index