WCSLIB

8.1

Generated on Thu Jul 6 2023 03:13:18 for WCSLIB by Doxygen 1.9.7

Thu Jul 6 2023 03:13:18

1 WCSLIB 8.1 and PGSBOX 8.1

1.1 Contents

1.2 Copyright


```
WCSLIB 8.1 - an implementation of the FITS WCS standard.
Copyright (C) 1995-2023, Mark Calabretta

WCSLIB is free software: you can redistribute it and/or modify it under the
terms of the GNU Lesser General Public License as published by the Free
Software Foundation, either version 3 of the License, or (at your option)
any later version.

WCSLIB is distributed in the hope that it will be useful, but WITHOUT ANY
WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
FOR A PARTICULAR PURPOSE.  See the GNU Lesser General Public License for
more details.

You should have received a copy of the GNU Lesser General Public License
along with WCSLIB.  If not, see http://www.gnu.org/licenses.

Direct correspondence concerning WCSLIB to mark@calabretta.id.au

Author: Mark Calabretta, Australia Telescope National Facility, CSIRO.
http://www.atnf.csiro.au/people/Mark.Calabretta
$Id: mainpage.dox,v 8.1 2023/07/05 17:12:07 mcalabre Exp $
```

1.3 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`.

1.4 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

1.5 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.

1.6 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.

1.7 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().

1.8 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).

1.9 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.

1.9.1 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.

1.9.2 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.

1.10 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, NPSMAX, 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 WCSLIB 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.

1.11 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.

1.12 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.

1.13 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.

1.14 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.

1.15 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
```

2 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.

3 Data Structure Index

3.1 Data Structures

Here are the data structures with brief descriptions:

4 File Index

4.1 File List

Here is a list of all files with brief descriptions:

5 Data Structure Documentation

5.1 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
- double a_radius
- double b_radius
- double c_radius
- double blon_obs
- double blat_obs
- double bdis_obs
- double dummy [2]

5.1.1 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.

5.1.2 Field Documentation

rsun_ref

```
double auxprm::rsun_ref
```

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

dsun_obs

`double auxprm::dsun_obs`

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

crln_obs

`double auxprm::crln_obs`

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

hgln_obs

`double auxprm::hgln_obs`

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

hglt_obs

`double auxprm::hglt_obs`

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

a_radius

`double auxprm::a_radius`

Length of the semi-major axis of a triaxial ellipsoid approximating the shape of a body (e.g. planet) in the solar system (m).

b_radius

`double auxprm::b_radius`

Length of the intermediate axis, normal to the semi-major and semi-minor axes, of a triaxial ellipsoid approximating the shape of a body (m).

c_radius

`double auxprm::c_radius`

Length of the semi-minor axis, normal to the semi-major axis, of a triaxial ellipsoid approximating the shape of a body (m).

blon_obs

```
double auxprm::blon_obs
```

Bodycentric longitude of the observer in the coordinate system fixed to the planet or other solar system body (deg, in range 0 to 360).

blat_obs

```
double auxprm::blat_obs
```

Bodycentric latitude of the observer in the coordinate system fixed to the planet or other solar system body (deg).

bdis_obs

```
double auxprm::bdis_obs
```

Bodycentric distance of the observer (m).

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

dummy

```
double auxprm::dummy[2]
```

5.2 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

5.2.1 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.

5.2.2 Field Documentation

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.

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).

phi0

`double celprm::phi0`

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

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.

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**ia keywords in FITS.

(Given and returned) The second pair of values are the native longitude, $\phi_{\rm p}$ [deg], and latitude, $\theta_{\rm p}$ [deg], of the celestial pole (the latter is the same as the celestial latitude of the native pole, $\delta_{\rm p}$) 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.

$\theta_{\rm p}$, the native latitude of the celestial pole (or equally the celestial latitude of the native pole, $\delta_{\rm p}$) 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 $\theta_{\rm p}$ 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.

prj

```
struct prjprm celprm::prj
```

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

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.

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.

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.

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().

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.

5.3 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 totdis
- double ∗ maxdis
- 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)
- int m_flag
- int m_naxis
- char(∗ m_dtype)[72]
- struct dpkey ∗ m_dp
- double ∗ m_maxdis

5.3.1 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.

5.3.2 Field Documentation

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.

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.

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.

ndp

`int disprm::ndp`

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

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().

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.

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.

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.

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).

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.

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.

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.

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.

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.

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.

i_naxis

```
int disprm::i_naxis
```

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

ndis

```
int disprm::ndis
```

(*Returned*) The number of distortion functions.

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().

disp2x

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

(For internal use only.)

disx2p

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

(For internal use only.)

m_flag

```
int disprm::m_flag
```

(For internal use only.)

m_naxis

```
int disprm::m_naxis
```

(For internal use only.)

m_dtype

```
disprm::m_dtype
```

(For internal use only.)

m_dp

```
double ** disprm::m_dp
```

(For internal use only.)

m_maxdis

```
double * disprm::m_maxdis
```

(For internal use only.)

5.4 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

5.4.1 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.

5.4.2 Field Documentation

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.

j

```
int dpkey::j
```

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

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).

i

```
int dpkey::i
```

f

```
double dpkey::f
```

value

```
union dpkey::value
```

(*Given*) A union comprised of

- dpkey::i,
- dpkey::f,

the record's value.

5.5 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]

5.5.1 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.

5.5.2 Field Documentation

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.

keyid

```
int fitskey::keyid
```

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

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.

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.

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.

padding

`int fitskey::padding`

(An unused variable inserted for alignment purposes only.)

i

`int fitskey::i`

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

k

`int64 fitskey::k`

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

l

`int fitskey::l`

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

f

`double fitskey::f`

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

c

`double fitskey::c`

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

s

`char fitskey::s`

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

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.

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.

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).

5.6 fitskeyid Struct Reference

Keyword indexing.

```
#include <fitshdr.h>
```

Data Fields

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

5.6.1 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().

5.6.2 Field Documentation

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.

count

```
int fitskeyid::count
```

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

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.

5.7 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

5.7.1 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*).

5.7.2 Field Documentation

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.

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.

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.

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.

cdelt

```
double * linprm::cdelt
```

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

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.

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.

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.

imgpix

```
double * linprm::imgpix
```

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

i_naxis

```
int linprm::i_naxis
```

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

unity

```
int linprm::unity
```

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

affine

```
int linprm::affine
```

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

simple

```
int linprm::simple
```

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

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().

tmpcrd

```
double * linprm::tmpcrd
```

(For internal use only.)

m_flag

```
int linprm::m_flag
```

(For internal use only.)

m_naxis

```
int linprm::m_naxis
```

(For internal use only.)

m_crpix

```
double * linprm::m_crpix
```

(For internal use only.)

m_pc

```
double * linprm::m_pc
```

(For internal use only.)

m_cdelt

```
double * linprm::m_cdelt
```

(For internal use only.)

m_dispre

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

(For internal use only.)

m_disseq

```
struct disprm * linprm::m_disseq
```

(For internal use only.)

5.8 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)

5.8.1 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.

5.8.2 Field Documentation

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.

code

```
char prjprm::code
```

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

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 $180^\circ/\pi$ (the value for FITS WCS).

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.

phi0

```
double prjprm::phi0
```

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

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.

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:

name

```
char prjprm::name
```

(*Returned*) Long name of the projection.

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

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.

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.

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.

equiareal

`int prjprm::equiareal`

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

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

conformal

`int prjprm::conformal`

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

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

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.

divergent

`int prjprm::divergent`

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

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

x0

`double prjprm::x0`

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

y0

`double prjprm::y0`

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

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().

padding

`void * prjprm::padding`

(An unused variable inserted for alignment purposes only.)

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.

m

`int prjprm::m`

n

`int prjprm::n`

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

prjx2s

`prjprm::prjx2s`

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

prjs2x

`prjprm::prjs2x`

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

5.9 pscard Struct Reference

Store for **PS**i_ma keyrecords.

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

Data Fields

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

5.9.1 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.

5.9.2 Field Documentation

i

```
int pscard::i
```

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

m

```
int pscard::m
```

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

value

```
char pscard::value
```

(*Given*) Parameter value.

5.10 pvcard Struct Reference

Store for **PV**i_ma keyrecords.

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

Data Fields

- int i
- int m
- double value

5.10.1 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.

5.10.2 Field Documentation

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.

m

`int pvcard::m`

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

value

`double pvcard::value`

(*Given*) Parameter value.

5.11 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)

5.11.1 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.

5.11.2 Field Documentation

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.

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.)

code

```
char spcprm::code
```

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

crval

```
double spcprm::crval
```

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

restfrq

```
double spcprm::restfrq
```

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

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.

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:

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.

isGrism

`int spcprm::isGrism`

(*Returned*) Grism coordinates?

- 0: no,
- 1: in vacuum,
- 2: in air.

padding1

`int spcprm::padding1`

(An unused variable inserted for alignment purposes only.)

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().

padding2

`void * spcprm::padding2`

(An unused variable inserted for alignment purposes only.)

spxX2P

`spcprm::spxX2P`

(*Returned*) The first and ...

spxP2S

`spcprm::spxP2S`

(*Returned*) ... the second of the pointers to the transformation functions in the two-step algorithm chain $X \leadsto P \rightarrow S$ 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.

spxS2P

`spcprm::spxS2P`

(*Returned*) The first and ...

spxP2X

`spcprm::spxP2X`

(*Returned*) ... the second of the pointers to the transformation functions in the two-step algorithm chain $S \rightarrow P \leadsto X$ 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.

5.12 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

5.12.1 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.

5.12.2 Field Documentation

restfrq

`double spxprm::restfrq`

(*Returned*) Rest frequency [Hz].

restwav

`double spxprm::restwav`

(*Returned*) Rest wavelength [m].

wavetype

`int spxprm::wavetype`

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

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.

freq

```
double spxprm::freq
```

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

afrq

```
double spxprm::afrq
```

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

ener

```
double spxprm::ener
```

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

wavn

```
double spxprm::wavn
```

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

vrad

```
double spxprm::vrad
```

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

wave

```
double spxprm::wave
```

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

vopt

```
double spxprm::vopt
```

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

zopt

```
double spxprm::zopt
```

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

awav

```
double spxprm::awav
```

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

velo

```
double spxprm::velo
```

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

beta

```
double spxprm::beta
```

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

dfreqafrq

```
double spxprm::dfreqafrq
```

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

dafrqfreq

```
double spxprm::dafrqfreq
```

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

dfreqener

```
double spxprm::dfreqener
```

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

denerfreq

```
double spxprm::denerfreq
```

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

dfreqwavn

```
double spxprm::dfreqwavn
```

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

dwavnfreq

```
double spxprm::dwavnfreq
```

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

dfreqvrad

```
double spxprm::dfreqvrad
```

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

dvradfreq

```
double spxprm::dvradfreq
```

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

dfreqwave

```
double spxprm::dfreqwave
```

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

dwavefreq

```
double spxprm::dwavefreq
```

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

dfreqawav

```
double spxprm::dfreqawav
```

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

dawavfreq

```
double spxprm::dawavfreq
```

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

dfreqvelo

```
double spxprm::dfreqvelo
```

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

dvelofreq

```
double spxprm::dvelofreq
```

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

dwavevopt

```
double spxprm::dwavevopt
```

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

dvoptwave

```
double spxprm::dvoptwave
```

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

dwavezopt

```
double spxprm::dwavezopt
```

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

dzoptwave

```
double spxprm::dzoptwave
```

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

dwaveawav

```
double spxprm::dwaveawav
```

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

dawavwave

```
double spxprm::dawavwave
```

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

dwavevelo

```
double spxprm::dwavevelo
```

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

dvelowave

```
double spxprm::dvelowave
```

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

dawavvelo

```
double spxprm::dawavvelo
```

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

dveloawav

```
double spxprm::dveloawav
```

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

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 ...

dbetavelo

```
double spxprm::dbetavelo
```

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

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().

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.

5.13 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

5.13.1 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.

5.13.2 Field Documentation

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.

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.

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.

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_**3**a - 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.

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.

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;
```

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.

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).

padding

```
int tabprm::padding
```

(An unused variable inserted for alignment purposes only.)

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).

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].

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].

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.

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().

m_flag

```
int tabprm::m_flag
```

(For internal use only.)

m_M

```
int tabprm::m_M
```

(For internal use only.)

m_N

```
int tabprm::m_N
```

(For internal use only.)

set_M

```
int tabprm::set_M
```

(For internal use only.)

m_K

```
int tabprm::m_K
```

(For internal use only.)

m_map

```
int tabprm::m_map
```

(For internal use only.)

m_crval

```
int tabprm::m_crval
```

(For internal use only.)

m_index

```
int tabprm::m_index
```

(For internal use only.)

m_indxs

```
int tabprm::m_indxs
```

(For internal use only.)

m_coord

```
int tabprm::m_coord
```

(For internal use only.)

5.14 wcserr Struct Reference

Error message handling.

```
#include <wcserr.h>
```

Data Fields

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

5.14.1 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.

5.14.2 Field Documentation

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.

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.

function

```
const char* wcserr::function
```

file

```
const char* wcserr::file
```

msg

```
char * wcserr::msg
```

Informative error message.

5.15 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 time
- int cubeface
- int dummy
- 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

5.15.1 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.

5.15.2 Field Documentation

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.

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.

crpix

`double * wcsprm::crpix`

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

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.

cdelt

`double * wcsprm::cdelt`

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

crval

`double * wcsprm::crval`

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

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.

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.

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 ...

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.

restfrq

`double wcsprm::restfrq`

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

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.

npv

`int wcsprm::npv`

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

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().

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.

nps

```
int wcsprm::nps
```

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

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().

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).

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 ...

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.

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.

  CROTAi may not formally co-exist with **PC**i_ja.

  CROTAi 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.

CDi_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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

jepoch

```
double wcsprm::jepoch
```

(Given, auxiliary) **JEPOCH** 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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

ntab

```
int wcsprm::ntab
```

(*Given*) See wcsprm::tab.

nwtb

```
int wcsprm::nwtb
```

(*Given*) See wcsprm::wtb.

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.

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.

lngtyp

```
char wcsprm::lngtyp
```

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

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.)

lng

```
int wcsprm::lng
```

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

lat

```
int wcsprm::lat
```

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

spec

```
int wcsprm::spec
```

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

time

```
int wcsprm::time
```

(*Returned*) ... index for the time 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.

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).

dummy

```
int wcsprm::dummy
```

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.

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

lin

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

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

cel

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

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

spc

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

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

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().

m_flag

```
int wcsprm::m_flag
```

(For internal use only.)

m_naxis

```
int wcsprm::m_naxis
```

(For internal use only.)

m_crpix

```
double * wcsprm::m_crpix
```

(For internal use only.)

m_pc

```
double * wcsprm::m_pc
```

(For internal use only.)

m_cdelt

```
double * wcsprm::m_cdelt
```

(For internal use only.)

m_crval

`double * wcsprm::m_crval`

(For internal use only.)

m_cunit

`wcsprm::m_cunit`

(For internal use only.)

m_ctype

`wcsprm::m_ctype`

(For internal use only.)

m_pv

`struct pvcard * wcsprm::m_pv`

(For internal use only.)

m_ps

`struct pscard * wcsprm::m_ps`

(For internal use only.)

m_cd

`double * wcsprm::m_cd`

(For internal use only.)

m_crota

`double * wcsprm::m_crota`

(For internal use only.)

m_colax

`int * wcsprm::m_colax`

(For internal use only.)

m_cname

```
wcsprm::m_cname
```

(For internal use only.)

m_crder

```
double * wcsprm::m_crder
```

(For internal use only.)

m_csyer

```
double * wcsprm::m_csyer
```

(For internal use only.)

m_czphs

```
double * wcsprm::m_czphs
```

(For internal use only.)

m_cperi

```
double * wcsprm::m_cperi
```

(For internal use only.)

m_aux

```
struct auxprm* wcsprm::m_aux
```

m_tab

```
struct tabprm * wcsprm::m_tab
```

(For internal use only.)

m_wtb

```
struct wtbarr * wcsprm::m_wtb
```

(For internal use only.)

5.16 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

5.16.1 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.

5.16.2 Field Documentation

i

```
int wtbarr::i
```

(*Given*) Image axis number.

m

```
int wtbarr::m
```

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

kind

`int wtbarr::kind`

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

- c: coordinate array,
- i: index vector.

extnam

`char wtbarr::extnam`

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

extver

`int wtbarr::extver`

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

extlev

`int wtbarr::extlev`

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

ttype

`char wtbarr::ttype`

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

row

`long wtbarr::row`

(*Given*) Table row number.

ndim

`int wtbarr::ndim`

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

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.

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.

6 File Documentation

6.1 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 [ ]

6.1.1 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.

celperr() 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.

6.1.2 Macro Definition Documentation

CELLEN

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

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

celini_errmsg

```
#define celini_errmsg cel_errmsg
```

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

celprt_errmsg

```
#define celprt_errmsg cel_errmsg
```

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

celset_errmsg

```
#define celset_errmsg cel_errmsg
```

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

celx2s_errmsg

```
#define celx2s_errmsg cel_errmsg
```

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

cels2x_errmsg

```
#define cels2x_errmsg cel_errmsg
```

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

6.1.3 Enumeration Type Documentation

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	

6.1.4 Function Documentation

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.

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.

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.

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.

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.

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().

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().

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().

6.1.5 Variable Documentation

cel_errmsg

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

6.2 cel.h

Go to the documentation of this file.

```
/*============================================================================
  WCSLIB 8.1 - an implementation of the FITS WCS standard.
  Copyright (C) 1995-2023, Mark Calabretta

  This file is part of WCSLIB.

  WCSLIB is free software: you can redistribute it and/or modify it under the
  terms of the GNU Lesser General Public License as published by the Free
  Software Foundation, either version 3 of the License, or (at your option)
  any later version.

  WCSLIB is distributed in the hope that it will be useful, but WITHOUT ANY
  WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
  FOR A PARTICULAR PURPOSE.  See the GNU Lesser General Public License for
  more details.

  You should have received a copy of the GNU Lesser General Public License
  along with WCSLIB.  If not, see http://www.gnu.org/licenses.

  Author: Mark Calabretta, Australia Telescope National Facility, CSIRO.
  http://www.atnf.csiro.au/people/Mark.Calabretta
  $Id: cel.h,v 8.1 2023/07/05 17:12:07 mcalabre Exp $
```

```
*=============================================================================
*
* WCSLIB 8.1 - C routines that implement the FITS World Coordinate System
* (WCS) standard.  Refer to the README file provided with WCSLIB for an
* overview of the library.
*
*
* Summary of the cel routines
* ---------------------------
* 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.
*
* celperr() 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.
*
*
* celini() - Default constructor for the celprm struct
* ----------------------------------------------------
* 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.
*
* Returned:
*   cel       struct celprm*
*                       Celestial transformation parameters.
*
* Function return value:
*             int       Status return value:
*                         0: Success.
*                         1: Null celprm pointer passed.
*
*
* celfree() - Destructor for the celprm struct
* --------------------------------------------
* celfree() frees any memory that may have been allocated to store an error
* message in the celprm struct.
*
* Given:
*   cel       struct celprm*
*                       Celestial transformation parameters.
*
* Function return value:
*             int       Status return value:
*                         0: Success.
*                         1: Null celprm pointer passed.
*
*
* celsize() - Compute the size of a celprm struct
* -----------------------------------------------
* celsize() computes the full size of a celprm struct, including allocated
* memory.
*
* Given:
*   cel       const struct celprm*
```

```
*                       Celestial transformation parameters.
*
*                       If NULL, the base size of the struct and the allocated
*                       size are both set to zero.
*
* Returned:
*   sizes     int[2]    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().
*
* Function return value:
*             int       Status return value:
*                         0: Success.
*
*
* celprt() - Print routine for the celprm struct
* ----------------------------------------------
* celprt() prints the contents of a celprm struct using wcsprintf().  Mainly
* intended for diagnostic purposes.
*
* Given:
*   cel       const struct celprm*
*                       Celestial transformation parameters.
*
* Function return value:
*             int       Status return value:
*                         0: Success.
*                         1: Null celprm pointer passed.
*
*
* celperr() - Print error messages from a celprm struct
* -----------------------------------------------------
* 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.
*
* Given:
*   cel       const struct celprm*
*                       Coordinate transformation parameters.
*
*   prefix    const char *
*                       If non-NULL, each output line will be prefixed with
*                       this string.
*
* Function return value:
*             int       Status return value:
*                         0: Success.
*                         1: Null celprm pointer passed.
*
*
* celset() - Setup routine for the celprm struct
* ----------------------------------------------
* 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.
*
* Given and returned:
*   cel       struct celprm*
*                       Celestial transformation parameters.
*
* Function return value:
*             int       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().
*
*
* celx2s() - Pixel-to-world celestial transformation
* --------------------------------------------------
* celx2s() transforms (x,y) coordinates in the plane of projection to
* celestial coordinates (lng,lat).
*
* Given and returned:
```

```
*   cel       struct celprm*
*                       Celestial transformation parameters.
*
* Given:
*   nx,ny     int       Vector lengths.
*
*   sxy,sll   int       Vector strides.
*
*   x,y       const double[]
*                       Projected coordinates in pseudo "degrees".
*
* Returned:
*   phi,theta double[]  Longitude and latitude (phi,theta) in the native
*                       coordinate system of the projection [deg].
*
*   lng,lat   double[]  Celestial longitude and latitude (lng,lat) of the
*                       projected point [deg].
*
*   stat      int[]     Status return value for each vector element:
*                         0: Success.
*                         1: Invalid value of (x,y).
*
* Function return value:
*             int       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().
*
*
* cels2x() - World-to-pixel celestial transformation
* --------------------------------------------------
* cels2x() transforms celestial coordinates (lng,lat) to (x,y) coordinates in
* the plane of projection.
*
* Given and returned:
*   cel       struct celprm*
*                       Celestial transformation parameters.
*
* Given:
*   nlng,nlat int       Vector lengths.
*
*   sll,sxy   int       Vector strides.
*
*   lng,lat   const double[]
*                       Celestial longitude and latitude (lng,lat) of the
*                       projected point [deg].
*
* Returned:
*   phi,theta double[]  Longitude and latitude (phi,theta) in the native
*                       coordinate system of the projection [deg].
*
*   x,y       double[]  Projected coordinates in pseudo "degrees".
*
*   stat      int[]     Status return value for each vector element:
*                         0: Success.
*                         1: Invalid value of (lng,lat).
*
* Function return value:
*             int       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 (lng,lat) 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().
*
*
* celprm struct - Celestial transformation parameters
* ---------------------------------------------------
* 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.
*
*   int 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.
*
*   int offset
*     (Given) If true (non-zero), an offset will be applied to (x,y) to
*     force (x,y) = (0,0) at the fiducial point, (phi_0,theta_0).
*     Default is 0 (false).
*
*   double phi0
*     (Given) The native longitude, phi_0 [deg], and ...
*
*   double theta0
*     (Given) ... the native latitude, theta_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.
*
*   double ref[4]
*     (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 CRVALia keywords in
*     FITS.
*
*     (Given and returned) The second pair of values are the native longitude,
*     phi_p [deg], and latitude, theta_p [deg], of the celestial pole (the
*     latter is the same as the celestial latitude of the native pole,
*     delta_p) and these are given by the FITS keywords LONPOLEa and LATPOLEa
*     (or by PVi_2a and PVi_3a attached to the longitude axis which take
*     precedence if defined).
*
*     LONPOLEa defaults to phi_0 (see above) if the celestial latitude of the
*     fiducial point of the projection is greater than or equal to the native
*     latitude, otherwise phi_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.
*
*     theta_p, the native latitude of the celestial pole (or equally the
*     celestial latitude of the native pole, delta_p) is often determined
*     uniquely by CRVALia and LONPOLEa in which case LATPOLEa is ignored.
*     However, in some circumstances there are two valid solutions for theta_p
*     and LATPOLEa is used to choose between them.  LATPOLEa 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 LATPOLEa 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 LONPOLEa = +/- 90.0 then the celestial latitude of the native
*     pole is not determined by the first three reference values and LATPOLEa
*     specifies it completely.
*
*     The returned value, celprm::latpreq, specifies how LATPOLEa was actually
*     used.
*
*   struct prjprm prj
*     (Given and returned) Projection parameters described in the prologue to
*     prj.h.
*
*   double euler[5]
*     (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.
*
*   int latpreq
*     (Returned) For informational purposes, this indicates how the LATPOLEa
```

```
*     keyword was used
*       - 0: Not required, theta_p (== delta_p) was determined uniquely by the
*            CRVALia and LONPOLEa keywords.
*       - 1: Required to select between two valid solutions of theta_p.
*       - 2: theta_p was specified solely by LATPOLEa.
*
*   int 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.
*
*   struct wcserr *err
*     (Returned) If enabled, when an error status is returned, this struct
*     contains detailed information about the error, see wcserr_enable().
*
*   void *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.
*
*===========================================================================*/

#ifndef WCSLIB_CEL
#define WCSLIB_CEL

#include "prj.h"

#ifdef __cplusplus
extern "C" {
#endif


extern const char *cel_errmsg[];

enum cel_errmsg_enum {
  CELERR_SUCCESS         = 0,      // Success.
  CELERR_NULL_POINTER    = 1,         // Null celprm pointer passed.
  CELERR_BAD_PARAM       = 2,    // Invalid projection parameters.
  CELERR_BAD_COORD_TRANS = 3,      // Invalid coordinate transformation
                                 // parameters.
  CELERR_ILL_COORD_TRANS = 4,      // Ill-conditioned coordinated transformation
                                 // parameters.
  CELERR_BAD_PIX         = 5,      // One or more of the (x,y) coordinates were
                                 // invalid.
  CELERR_BAD_WORLD       = 6     // One or more of the (lng,lat) coordinates
                                 // were invalid.
};

struct celprm {
  // Initialization flag (see the prologue above).
  //-------------------------------------------------------------------------
  int    flag;                         // Set to zero to force initialization.

  // Parameters to be provided (see the prologue above).
  //-------------------------------------------------------------------------
  int    offset;             // Force (x,y) = (0,0) at (phi_0,theta_0).
  double phi0, theta0;                   // Native coordinates of fiducial point.
  double ref[4];                // Celestial coordinates of fiducial
                                 // point and native coordinates of
                                 // celestial pole.

  struct prjprm prj;            // Projection parameters (see prj.h).

  // Information derived from the parameters supplied.
  //-------------------------------------------------------------------------
  double euler[5];           // Euler angles and functions thereof.
  int    latpreq;                  // LATPOLEa requirement.
  int    isolat;             // True if |latitude| is preserved.

  // Error handling
  //-------------------------------------------------------------------------
  struct wcserr *err;

  // Private
  //-------------------------------------------------------------------------
  void   *padding;                 // (Dummy inserted for alignment purposes.)
};

// Size of the celprm struct in int units, used by the Fortran wrappers.
#define CELLEN (sizeof(struct celprm)/sizeof(int))


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

```

int celfree(struct celprm *cel);

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

int celprt(const struct celprm *cel);

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

int celset(struct celprm *cel);

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[]);

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[]);


// Deprecated.
#define celini_errmsg cel_errmsg
#define celprt_errmsg cel_errmsg
#define celset_errmsg cel_errmsg
#define celx2s_errmsg cel_errmsg
#define cels2x_errmsg cel_errmsg

#ifdef __cplusplus
}
#endif

#endif // WCSLIB_CEL
```

6.3 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 ***DP**ja and **DQ**ia.*
- 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.

6.3.1 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.

DP`ja` 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:

6.3.2 Macro Definition Documentation

DISP2X_ARGS

```
#define DISP2X_ARGS
```

Value:

```
int inverse, const int iparm[], const double dparm[], \
int ncrd, const double rawcrd[], double *discrd
```

DISX2P_ARGS

```
#define DISX2P_ARGS
```

Value:

```
int inverse, const int iparm[], const double dparm[], \
int ncrd, const double discrd[], double *rawcrd
```

DPLEN

```
#define DPLEN (sizeof(struct dpkey)/sizeof(int))
```

DISLEN

```
#define DISLEN (sizeof(struct disprm)/sizeof(int))
```

6.3.3 Enumeration Type Documentation

dis_errmsg_enum

```
enum dis_errmsg_enum
```

Enumerator

DISERR_SUCCESS	
DISERR_NULL_POINTER	
DISERR_MEMORY	
DISERR_BAD_PARAM	
DISERR_DISTORT	
DISERR_DEDISTORT	

6.3.4 Function Documentation

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.

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.

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.

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.

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.

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().

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().

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.

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.

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.

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.

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.

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().

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.

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] ∼= 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().

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.

6.3.5 Variable Documentation

dis_errmsg

```
const char * dis_errmsg[]  [extern]
```

Error messages to match the status value returned from each function.

6.4 dis.h

Go to the documentation of this file.

```
/*============================================================================
  WCSLIB 8.1 - an implementation of the FITS WCS standard.
  Copyright (C) 1995-2023, Mark Calabretta

  This file is part of WCSLIB.

  WCSLIB is free software: you can redistribute it and/or modify it under the
  terms of the GNU Lesser General Public License as published by the Free
  Software Foundation, either version 3 of the License, or (at your option)
  any later version.

  WCSLIB is distributed in the hope that it will be useful, but WITHOUT ANY
  WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
  FOR A PARTICULAR PURPOSE.  See the GNU Lesser General Public License for
  more details.

  You should have received a copy of the GNU Lesser General Public License
  along with WCSLIB.  If not, see http://www.gnu.org/licenses.

  Author: Mark Calabretta, Australia Telescope National Facility, CSIRO.
  http://www.atnf.csiro.au/people/Mark.Calabretta
  $Id: dis.h,v 8.1 2023/07/05 17:12:07 mcalabre Exp $
*=============================================================================
*
* WCSLIB 8.1 - C routines that implement the FITS World Coordinate System
* (WCS) standard.  Refer to the README file provided with WCSLIB for an
* overview of the library.
*
*
* Summary of the dis routines
* ---------------------------
* 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 PCi_ja or CDi_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 CQDISia, DQia, and CQERRia.
* An additional floating-valued keyword, DVERRa, records the maximum value of
* the combined distortions.
*
* DPja and DQia 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 PVi_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 PVi_ma
* keywords to DQia and loads them into a disprm struct.  These DQia keyrecords
```

```
* have the form
*
=   DQia = 'TPV.m: <value>'
*
* where i, a, m, and the value for each DQia match each PVi_ma.  Consequently,
* WCSLIB would handle a FITS header containing these keywords, along with
* CQDISia = 'TPV' and the required DQia.NAXES and DQia.AXIS.ihat keywords.
*
* Note that, as defined, TPV assumes that CDi_ja is used to define the linear
* transformation.  The section on historical idiosyncrasies (below) cautions
* about translating CDi_ja to PCi_ja plus CDELTia 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 CPDISja = 'SIP' and
* the required DPja.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 (CRPIXja, PCi_ja, CRVALia, 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 CQDISia = 'DSS' and the required DQia.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 WATi_n keywords
* as a set of continued strings, thus providing the name for this distortion
* type.  WATi_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 CQDISia = 'WAT' and the required DPja.NAXES keywords.  For ZPX,
* the ZPN projection parameters are also encoded in WATi_n, and wcspih()
* translates these to standard PVi_ma.
*
* Note that, as defined, TNX and ZPX assume that CDi_ja is used to define the
```

```
* linear transformation.  The section on historical idiosyncrasies (below)
* cautions about translating CDi_ja to PCi_ja plus CDELTia 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 CPDISja or CQDISia), for use as both prior
* and sequent distortions.  Its DPja and DQia 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(xx + yy).  Note that even powers of r are excluded since they
* can be accomodated by powers of (xx + yy).
*
* 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 PVi_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 AMDXm (or AMDYm) 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 CRPIXja.
*
* Likewise, because TPV, TNX, and ZPX are defined in terms of CDi_ja, the
* independent variables of the polynomial are intermediate world coordinates
* rather than intermediate pixel coordinates.  Because sequent distortions
* are always applied before CDELTia, if CDi_ja is translated to PCi_ja plus
* CDELTia, then either CDELTia 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: Distortions are not yet handled by wcsbth(), or wcscompare().
*
*
* disndp() - Memory allocation for DPja and DQia
* ----------------------------------------------
* disndp() sets or gets the value of NDPMAX (default 256).  This global
* variable controls the maximum number of dpkey structs, for holding DPja or
* DQia 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.
*
* Given:
*   n         int       Value of NDPMAX; ignored if < 0.  Use a value less
*                       than zero to get the current value.
*
* Function return value:
*             int       Current value of NDPMAX.
*
*
* dpfill() - Fill the contents of a dpkey struct
* ----------------------------------------------
* 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 DPja and DQia record values, such as those of DPja.NAXES and
* DPja.AXIS.j, are intrinsically integer-valued.  While FITS header parsers
* are not expected to know in advance which of DPja and DQia 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.
*
* Given and returned:
*   dp        struct dpkey*
*                       Store for DPja and DQia keyvalues.
*
* Given:
*   keyword   const char *
*   field     const char *
*                       These arguments are concatenated with an intervening
*                       "." to construct the full record field name, i.e.
*                       including the keyword name, DPja or DQia (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.
*
*   j         int       Axis number (1-relative), i.e. the j in DPja or
*                       i in DQia.  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.
*
*   type      int       Data type of the record's value
*                         0: Integer,
*                         1: Floating point.
*
*   i         int       For type == 0, the integer value of the record.
*
*   f         double    For type == 1, the floating point value of the record.
*
* Function return value:
*             int       Status return value:
*                         0: Success.
```

```
*
*
* dpkeyi() - Get the data value in a dpkey struct as int
* -----------------------------------------------------
* dpkeyi() returns the data value in a dpkey struct as an integer value.
*
* Given and returned:
*   dp        const struct dpkey *
*                       Parsed contents of a DPja or DQia keyrecord.
*
* Function return value:
*             int       The record's value as int.
*
*
* dpkeyd() - Get the data value in a dpkey struct as double
* --------------------------------------------------------
* dpkeyd() returns the data value in a dpkey struct as a floating point
* value.
*
* Given and returned:
*   dp        const struct dpkey *
*                       Parsed contents of a DPja or DQia keyrecord.
*
* Function return value:
*             double    The record's value as double.
*
*
* disini() - Default constructor for the disprm struct
* ---------------------------------------------------
* 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.
*
*
* disinit() - Default constructor for the disprm struct
* ---------------------------------------------------
* 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.
*
* Given:
*   alloc     int       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.)
*
*   naxis     int       The number of world coordinate axes, used to determine
*                       array sizes.
*
* Given and returned:
*   dis       struct disprm*
*                       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).
*
* Given:
*   ndpmax    int       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.
*
* Function return value:
*             int       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().
*
*
* discpy() - Copy routine for the disprm struct
* --------------------------------------------
* 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.
*
* Given:
*   alloc     int       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.
*
*   dissrc    const struct disprm*
*                       Struct to copy from.
*
* Given and returned:
*   disdst    struct disprm*
*                       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).
*
* Function return value:
*             int       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().
*
*
* disfree() - Destructor for the disprm struct
* --------------------------------------------
* 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().
*
* Given:
*   dis       struct disprm*
*                       Distortion function parameters.
*
* Function return value:
*             int       Status return value:
*                         0: Success.
*                         1: Null disprm pointer passed.
*
*
* dissize() - Compute the size of a disprm struct
* -----------------------------------------------
* dissize() computes the full size of a disprm struct, including allocated
* memory.
*
* Given:
*   dis       const struct disprm*
*                       Distortion function parameters.
*
*                       If NULL, the base size of the struct and the allocated
*                       size are both set to zero.
*
* Returned:
*   sizes     int[2]    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.
*
* Function return value:
*             int       Status return value:
*                         0: Success.
*
*
* disprt() - Print routine for the disprm struct
* ----------------------------------------------
* disprt() prints the contents of a disprm struct using wcsprintf().  Mainly
* intended for diagnostic purposes.
*
* Given:
*   dis       const struct disprm*
*                       Distortion function parameters.
*
* Function return value:
```

```
*             int       Status return value:
*                         0: Success.
*                         1: Null disprm pointer passed.
*
*
* disperr() - Print error messages from a disprm struct
* -----------------------------------------------------
* 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.
*
* Given:
*   dis       const struct disprm*
*                       Distortion function parameters.
*
*   prefix    const char *
*                       If non-NULL, each output line will be prefixed with
*                       this string.
*
* Function return value:
*             int       Status return value:
*                         0: Success.
*                         1: Null disprm pointer passed.
*
*
* dishdo() - write FITS headers using TPD
* ---------------------------------------
* 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.
*
* Given and returned:
*   dis       struct disprm*
*                       Distortion function parameters.
*
* Function return value:
*             int       Status return value:
*                         0: Success.
*                         1: Null disprm pointer passed.
*                         3: No TPD translation.
*
*
* disset() - Setup routine for the disprm struct
* ----------------------------------------------
* 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.
*
* Given and returned:
*   dis       struct disprm*
*                       Distortion function parameters.
*
* Function return value:
*             int       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().
*
*
* disp2x() - Apply distortion function
* ------------------------------------
* 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.
*
*
* Given and returned:
*   dis       struct disprm*
*                       Distortion function parameters.
*
* Given:
*   rawcrd    const double[naxis]
*                       Array of coordinates.
*
* Returned:
*   discrd    double[naxis]
*                       Array of coordinates to which the distortion functions
```

```
*                       have been applied.
*
* Function return value:
*             int       Status return value:
*                         0: Success.
*                         1: Null disprm pointer passed.
*                         2: Memory allocation failed.
*                         3: Invalid parameter.
*                         4: Distort error.
*
*                       For returns > 1, a detailed error message is set in
*                       disprm::err if enabled, see wcserr_enable().
*
*
* disx2p() - Apply de-distortion function
* ---------------------------------------
* 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] ~= 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.
*
* Given and returned:
*   dis       struct disprm*
*                       Distortion function parameters.
*
* Given:
*   discrd    const double[naxis]
*                       Array of coordinates.
*
* Returned:
*   rawcrd    double[naxis]
*                       Array of coordinates to which the inverse distortion
*                       functions have been applied.
*
* Function return value:
*             int       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().
*
*
* diswarp() - Compute measures of distortion
* ------------------------------------------
* 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 (PCi_ja or CDi_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().
*
* Given and returned:
*   dis       struct disprm*
*                       Distortion function parameters.
*
* Given:
*   pixblc    const double[naxis]
*                       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,...).
*
*   pixtrc    const double[naxis]
*                       End of the range of pixel coordinates (prior) or
*                       intermediate pixel coordinates (sequent).
*
*   pixsamp   const double[naxis]
*                       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.
*
* Returned:
*   nsamp     int*      The number of pixel coordinates sampled.
*
*                       Can be specified as a NULL pointer if not required.
*
*   maxdis    double[naxis]
*                       For each individual distortion function, the
*                       maximum absolute value of the distortion.
*
*                       Can be specified as a NULL pointer if not required.
*
*   maxtot    double*   For the combination of all distortion functions, the
*                       maximum absolute value of the distortion.
*
*                       Can be specified as a NULL pointer if not required.
*
*   avgdis    double[naxis]
*                       For each individual distortion function, the
*                       mean value of the distortion.
*
*                       Can be specified as a NULL pointer if not required.
*
*   avgtot    double*   For the combination of all distortion functions, the
*                       mean value of the distortion.
*
*                       Can be specified as a NULL pointer if not required.
*
*   rmsdis    double[naxis]
*                       For each individual distortion function, the
*                       root mean square deviation of the distortion.
*
*                       Can be specified as a NULL pointer if not required.
*
*   rmstot    double*   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.
*
* Function return value:
*             int       Status return value:
*                         0: Success.
*                         1: Null disprm pointer passed.
*                         2: Memory allocation failed.
*                         3: Invalid parameter.
*                         4: Distort error.
*
*
* disprm struct - Distortion parameters
* -------------------------------------
* 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.
*
*   int 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.
*
*   int 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.
*
*   char (*dtype)[72]
*     (Given) Pointer to the first element of an array of char[72] containing
*     the name of the distortion function for each axis.
*
*   int ndp
*     (Given) The number of entries in the disprm::dp[] array.
*
*   int 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().
*
*   struct dpkey dp
*     (Given) Address of the first element of an array of length ndpmax of
*     dpkey structs.
*
*     As a FITS header parser encounters each DPja or DQia 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 DPja or DQia keyvalues,
*     not both.  disset() interprets them as required by the particular
*     distortion function.
*
*   double *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.
*
*   double 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.
*
*   int *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).
*
*   int *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.
*
*   int **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.
*
*   double **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.
*
*   double **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.
*
*   int **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.
*
*   double **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.
*
*   int i_naxis
*     (Returned) Dimension of the internal arrays (normally equal to naxis).
*
*   int ndis
*     (Returned) The number of distortion functions.
*
*   struct wcserr *err
*     (Returned) If enabled, when an error status is returned, this struct
*     contains detailed information about the error, see wcserr_enable().
*
*   int (**disp2x)(DISP2X_ARGS)
*     (For internal use only.)
*   int (**disx2p)(DISX2P_ARGS)
*     (For internal use only.)
*   double *dummy
*     (For internal use only.)
*   int m_flag
*     (For internal use only.)
*   int m_naxis
*     (For internal use only.)
*   char (*m_dtype)[72]
*     (For internal use only.)
*   double **m_dp
*     (For internal use only.)
*   double *m_maxdis
*     (For internal use only.)
*
*
* dpkey struct - Store for DPja and DQia keyvalues
* ------------------------------------------------
* The dpkey struct is used to pass the parsed contents of DPja or DQia
* keyrecords to disset() via the disprm struct.  A disprm struct must hold
* only DPja or DQia keyvalues, not both.
*
* All members of this struct are to be set by the user.
*
*   char field[72]
*     (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.
*
*   int j
*     (Given) Axis number (1-relative), i.e. the j in DPja or i in DQia.
*
*   int type
*     (Given) The data type of the record's value
*       - 0: Integer (stored as an int),
*       - 1: Floating point (stored as a double).
*
*   union value
*     (Given) A union comprised of
*       - dpkey::i,
*       - dpkey::f,
*
```

```
*     the record's value.
*
*
* Global variable: const char *dis_errmsg[] - Status return messages
* ---------------------------------------------------------------------
* Error messages to match the status value returned from each function.
*
*===========================================================================*/

#ifndef WCSLIB_DIS
#define WCSLIB_DIS

#ifdef __cplusplus
extern "C" {
#endif


extern const char *dis_errmsg[];

enum dis_errmsg_enum {
  DISERR_SUCCESS      = 0,        // Success.
  DISERR_NULL_POINTER = 1,    // Null disprm pointer passed.
  DISERR_MEMORY       = 2, // Memory allocation failed.
  DISERR_BAD_PARAM    = 3,      // Invalid parameter value.
  DISERR_DISTORT      = 4,        // Distortion error.
  DISERR_DEDISTORT    = 5       // De-distortion error.
};

// For use in declaring distortion function prototypes (= DISX2P_ARGS).
#define DISP2X_ARGS int inverse, const int iparm[], const double dparm[], \
int ncrd, const double rawcrd[], double *discrd

// For use in declaring de-distortion function prototypes (= DISP2X_ARGS).
#define DISX2P_ARGS int inverse, const int iparm[], const double dparm[], \
int ncrd, const double discrd[], double *rawcrd


// Struct used for storing DPja and DQia keyvalues.
struct dpkey {
  char field[72];          // Full record field name (no colon).
  int j;                       // Axis number, as in DPja (1-relative).
  int type;                 // Data type of value.
  union {
    int    i;                  // Integer record value.
    double f;                  // Floating point record value.
  } value;                 // Record value.
};

// Size of the dpkey struct in int units, used by the Fortran wrappers.
#define DPLEN (sizeof(struct dpkey)/sizeof(int))


struct disprm {
  // Initialization flag (see the prologue above).
  //-----------------------------------------------------------------------
  int flag;                 // Set to zero to force initialization.

  // Parameters to be provided (see the prologue above).
  //-----------------------------------------------------------------------
  int naxis;                       // The number of pixel coordinate elements,
                                // given by NAXIS.
  char   (*dtype)[72];             // For each axis, the distortion type.
  int    ndp;                        // Number of DPja or DQia keywords, and the
  int    ndpmax;          // number for which space was allocated.
  struct dpkey *dp;               // DPja or DQia keyvalues (not both).
  double totdis;          // The maximum combined distortion.
  double *maxdis;         // For each axis, the maximum distortion.

  // Information derived from the parameters supplied.
  //-----------------------------------------------------------------------
  int    *docorr;         // For each axis, the mode of correction.
  int    *Nhat;                     // For each axis, the number of coordinate
                                // axes that form the independent variables
                                // of the distortion function.
  int    **axmap;          // For each axis, the axis mapping array.
  double **offset;                // For each axis, renormalization offsets.
  double **scale;          // For each axis, renormalization scales.
  int    **iparm;          // For each axis, the array of integer
                                // distortion parameters.
  double **dparm;          // For each axis, the array of floating
                                // point distortion parameters.
  int    i_naxis;                // Dimension of the internal arrays.
  int    ndis;                      // The number of distortion functions.

  // Error handling, if enabled.
  //-----------------------------------------------------------------------
  struct wcserr *err;
```

```

  // Private - the remainder are for internal use.
  //-----------------------------------------------------------------------------
  int (**disp2x)(DISP2X_ARGS); // For each axis, pointers to the
  int (**disx2p)(DISX2P_ARGS); // distortion function and its inverse.

  int    m_flag, m_naxis;  // The remainder are for memory management.
  char   (*m_dtype)[72];
  struct dpkey *m_dp;
  double *m_maxdis;
};

// Size of the disprm struct in int units, used by the Fortran wrappers.
#define DISLEN (sizeof(struct disprm)/sizeof(int))


int disndp(int n);

int dpfill(struct dpkey *dp, const char *keyword, const char *field, int j,
           int type, int i, double f);

int    dpkeyi(const struct dpkey *dp);

double dpkeyd(const struct dpkey *dp);

int disini(int alloc, int naxis, struct disprm *dis);

int disinit(int alloc, int naxis, struct disprm *dis, int ndpmax);

int discpy(int alloc, const struct disprm *dissrc, struct disprm *disdst);

int disfree(struct disprm *dis);

int dissize(const struct disprm *dis, int sizes[2]);

int disprt(const struct disprm *dis);

int disperr(const struct disprm *dis, const char *prefix);

int dishdo(struct disprm *dis);

int disset(struct disprm *dis);

int disp2x(struct disprm *dis, const double rawcrd[], double discrd[]);

int disx2p(struct disprm *dis, const double discrd[], double rawcrd[]);

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);

#ifdef __cplusplus
}
#endif

#endif // WCSLIB_DIS
```

6.5 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.

6.5.1 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.

6.5.2 Macro Definition Documentation

FITSHDR_KEYWORD

```
#define FITSHDR_KEYWORD 0x01
```

Bit mask for the status flag bit-vector returned by fitshdr() indicating illegal keyword syntax.

FITSHDR_KEYVALUE

```
#define FITSHDR_KEYVALUE 0x02
```

Bit mask for the status flag bit-vector returned by fitshdr() indicating illegal keyvalue syntax.

FITSHDR_COMMENT

```
#define FITSHDR_COMMENT 0x04
```

Bit mask for the status flag bit-vector returned by fitshdr() indicating illegal keycomment syntax.

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.

FITSHDR_CARD

```
#define FITSHDR_CARD 0x08
```

Deprecated Added for backwards compatibility, use *FITSHDR_KEYREC* instead.

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.

KEYIDLEN

```
#define KEYIDLEN (sizeof(struct fitskeyid)/sizeof(int))
```

KEYLEN

```
#define KEYLEN (sizeof(struct fitskey)/sizeof(int))
```

6.5.3 Typedef Documentation

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.

6.5.4 Enumeration Type Documentation

fitshdr_errmsg_enum

```
enum fitshdr_errmsg_enum
```

Enumerator

FITSHDRERR_SUCCESS	
FITSHDRERR_NULL_POINTER	
FITSHDRERR_MEMORY	
FITSHDRERR_FLEX_PARSER	
FITSHDRERR_DATA_TYPE	

6.5.5 Function Documentation

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 keycomment, 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.

6.5.6 Variable Documentation

fitshdr_errmsg

```
const char * fitshdr_errmsg[] [extern]
```

Error messages to match the status value returned from each function.

6.6 fitshdr.h

Go to the documentation of this file.

```
/*============================================================================
  WCSLIB 8.1 - an implementation of the FITS WCS standard.
  Copyright (C) 1995-2023, Mark Calabretta

  This file is part of WCSLIB.

  WCSLIB is free software: you can redistribute it and/or modify it under the
  terms of the GNU Lesser General Public License as published by the Free
  Software Foundation, either version 3 of the License, or (at your option)
  any later version.

  WCSLIB is distributed in the hope that it will be useful, but WITHOUT ANY
  WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
  FOR A PARTICULAR PURPOSE.  See the GNU Lesser General Public License for
  more details.

  You should have received a copy of the GNU Lesser General Public License
  along with WCSLIB.  If not, see http://www.gnu.org/licenses.

  Author: Mark Calabretta, Australia Telescope National Facility, CSIRO.
  http://www.atnf.csiro.au/people/Mark.Calabretta
  $Id: fitshdr.h,v 8.1 2023/07/05 17:12:07 mcalabre Exp $
*=============================================================================
*
```

```
* WCSLIB 8.1 - C routines that implement the FITS World Coordinate System
* (WCS) standard.  Refer to the README file provided with WCSLIB for an
* overview of the library.
*
*
* Summary of the fitshdr routines
* -------------------------------
* 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://fits.gsfc.nasa.gov
*
* 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.
*
*
* fitshdr() - FITS header parser routine
* --------------------------------------
* fitshdr() parses a character array containing a FITS header, extracting
* all keywords and their values into an array of fitskey structs.
*
* Given:
*   header    const char []
*                       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.
*
*   nkeyrec   int       Number of keyrecords in header[].
*
*   nkeyids   int       Number of entries in keyids[].
*
* Given and returned:
*   keyids    struct fitskeyid []
*                       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.
*
* Returned:
*   nreject   int*      Number of header keyrecords rejected for syntax
*                       errors.
*
*   keys      struct fitskey**
*                       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().
*
* Function return value:
*             int       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 keycomment, 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.
*
*
* fitskeyid struct - Keyword indexing
* -----------------------------------
* 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().
*
*   char name[12]:
*     (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.
*
*   int count:
*     (Returned) The number of matches found for the keyword.
*
*   int idx[2]:
*     (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.
*
*
* fitskey struct - Keyword/value information
* ------------------------------------------
* 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.
*
*   int 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.
*
*   int keyid
*     (Returned) Index into the first entry in keyids[] with which the
*     keyrecord matches, else -1.
*
*   int 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.
*
*   char keyword[12]
*     (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.
*
*   int 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.
*
*   int padding
*     (An unused variable inserted for alignment purposes only.)
*
*   union 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.
*
*   int i
*     (Returned) Logical (fitskey::type == 1) and 32-bit signed integer
*     (fitskey::type == 2) data types in the fitskey::keyvalue union.
*
*   int64 k
*     (Returned) 64-bit signed integer (fitskey::type == 3) data type in the
*     fitskey::keyvalue union.
*
*   int l[8]
*     (Returned) Very long integer (fitskey::type == 4) data type in the
*     fitskey::keyvalue union.
*
*   double f
*     (Returned) Floating point (fitskey::type == 5) data type in the
*     fitskey::keyvalue union.
*
*   double c[2]
*     (Returned) Integer and floating point complex (fitskey::type == 6 || 7)
*     data types in the fitskey::keyvalue union.
*
*   char s[72]
*     (Returned) Null-terminated string (fitskey::type == 8) data type in the
*     fitskey::keyvalue union.
*
*   int 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.
*
*   char comment[84]
*     (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).
*
*
* Global variable: const char *fitshdr_errmsg[] - Status return messages
* ---------------------------------------------------------------------
* Error messages to match the status value returned from each function.
*
*===========================================================================*/

#ifndef WCSLIB_FITSHDR
```

```
#define WCSLIB_FITSHDR

#include "wcsconfig.h"

#ifdef __cplusplus
extern "C" {
#endif

#define FITSHDR_KEYWORD  0x01
#define FITSHDR_KEYVALUE 0x02
#define FITSHDR_COMMENT  0x04
#define FITSHDR_KEYREC   0x08
#define FITSHDR_CARD     0x08   // Alias for backwards compatibility.
#define FITSHDR_TRAILER  0x10


extern const char *fitshdr_errmsg[];

enum fitshdr_errmsg_enum {
  FITSHDRERR_SUCCESS      = 0,        // Success.
  FITSHDRERR_NULL_POINTER = 1,   // Null fitskey pointer passed.
  FITSHDRERR_MEMORY       = 2, // Memory allocation failed.
  FITSHDRERR_FLEX_PARSER  = 3,    // Fatal error returned by Flex parser.
  FITSHDRERR_DATA_TYPE    = 4       // Unrecognised data type.
};

#ifdef WCSLIB_INT64
  typedef WCSLIB_INT64 int64;
#else
  typedef int int64[3];
#endif


// Struct used for indexing the keywords.
struct fitskeyid {
  char name[12];            // Keyword name, null-terminated.
  int  count;                      // Number of occurrences of keyword.
  int  idx[2];                       // Indices into fitskey array.
};

// Size of the fitskeyid struct in int units, used by the Fortran wrappers.
#define KEYIDLEN (sizeof(struct fitskeyid)/sizeof(int))


// Struct used for storing FITS keywords.
struct fitskey {
  int  keyno;                      // Header keyrecord sequence number (1-rel).
  int  keyid;                      // Index into fitskeyid[].
  int  status;                    // Header keyrecord status bit flags.
  char keyword[12];              // Keyword name, null-filled.
  int  type;                          // Keyvalue type (see above).
  int  padding;                  // (Dummy inserted for alignment purposes.)
  union {
    int    i;                  // 32-bit integer and logical values.
    int64  k;                     // 64-bit integer values.
    int    l[8];               // Very long signed integer values.
    double f;                  // Floating point values.
    double c[2];               // Complex values.
    char   s[72];              // String values, null-terminated.
  } keyvalue;                   // Keyvalue.
  int  ulen;                          // Length of units string.
  char comment[84];              // Comment (or keyrecord), null-terminated.
};

// Size of the fitskey struct in int units, used by the Fortran wrappers.
#define KEYLEN (sizeof(struct fitskey)/sizeof(int))


int fitshdr(const char header[], int nkeyrec, int nkeyids,
            struct fitskeyid keyids[], int *nreject, struct fitskey **keys);


#ifdef __cplusplus
}
#endif

#endif // WCSLIB_FITSHDR
```

6.7 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.*

6.7.1 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.

6.7.2 Function Documentation

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.

6.8 getwcstab.h

Go to the documentation of this file.

```
/*============================================================================
  WCSLIB 8.1 - an implementation of the FITS WCS standard.
  Copyright (C) 1995-2023, Mark Calabretta

  This file is part of WCSLIB.

  WCSLIB is free software: you can redistribute it and/or modify it under the
  terms of the GNU Lesser General Public License as published by the Free
  Software Foundation, either version 3 of the License, or (at your option)
  any later version.

  WCSLIB is distributed in the hope that it will be useful, but WITHOUT ANY
  WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
  FOR A PARTICULAR PURPOSE.  See the GNU Lesser General Public License for
  more details.

  You should have received a copy of the GNU Lesser General Public License
  along with WCSLIB.  If not, see http://www.gnu.org/licenses.

  Author: Mark Calabretta, Australia Telescope National Facility, CSIRO.
  http://www.atnf.csiro.au/people/Mark.Calabretta
  $Id: getwcstab.h,v 8.1 2023/07/05 17:12:07 mcalabre Exp $
*=============================================================================
*
* WCSLIB 8.1 - C routines that implement the FITS World Coordinate System
* (WCS) standard.  Refer to the README file provided with WCSLIB for an
* overview of the library.
*
* Summary of the getwcstab routines
* ---------------------------------
* 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.
```

```
*
*
* fits_read_wcstab() - FITS 'TAB' table reading routine
* ----------------------------------------------------
* fits_read_wcstab() extracts arrays from a binary table required in
* constructing 'TAB' coordinates.
*
* Given:
*   fptr      fitsfile *
*                       Pointer to the file handle returned, for example, by
*                       the fits_open_file() routine in CFITSIO.
*
*   nwtb      int       Number of arrays to be read from the binary table(s).
*
* Given and returned:
*   wtb       wtbarr *  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.
*
* Returned:
*   status    int *     CFITSIO status value.
*
* Function return value:
*             int       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.
*
*
* wtbarr typedef
* --------------
* The wtbarr typedef is defined as a struct containing the following members:
*
*   int i
*     Image axis number.
*
*   int m
*     Array axis number for index vectors.
*
*   int kind
*     Character identifying the array type:
*       - c: coordinate array,
```

```
*       - i: index vector.
*
*   char extnam[72]
*     EXTNAME identifying the binary table extension.
*
*   int extver
*     EXTVER identifying the binary table extension.
*
*   int extlev
*     EXTLEV identifying the binary table extension.
*
*   char ttype[72]
*     TTYPEn identifying the column of the binary table that contains the
*     array.
*
*   long row
*     Table row number.
*
*   int ndim
*     Expected dimensionality of the array.
*
*   int *dimlen
*     Address of the first element of an array of int of length ndim into
*     which the array axis lengths are to be written.
*
*   double **arrayp
*     Pointer to an array of double which is to be allocated by the user
*     and into which the array is to be written.
*
*===========================================================================*/

#ifndef WCSLIB_GETWCSTAB
#define WCSLIB_GETWCSTAB

#ifdef __cplusplus
extern "C" {
#endif

#include <fitsio.h>

typedef struct {
  int  i;                       // Image axis number.
  int  m;                       // Array axis number for index vectors.
  int  kind;                    // Array type, 'c' (coord) or 'i' (index).
  char extnam[72];              // EXTNAME of binary table extension.
  int  extver;                  // EXTVER  of binary table extension.
  int  extlev;                  // EXTLEV  of binary table extension.
  char ttype[72];               // TTYPEn of column containing the array.
  long row;                     // Table row number.
  int  ndim;                    // Expected array dimensionality.
  int  *dimlen;                 // Where to write the array axis lengths.
  double **arrayp;              // Where to write the address of the array
                                // allocated to store the array.
} wtbarr;


int fits_read_wcstab(fitsfile *fptr, int nwtb, wtbarr *wtb, int *status);


#ifdef __cplusplus
}
#endif

#endif // WCSLIB_GETWCSTAB
```

6.9 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.

6.9.1 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.

6.9.2 Macro Definition Documentation

LINLEN

```
#define LINLEN (sizeof(struct linprm)/sizeof(int))
```

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

linini_errmsg

```
#define linini_errmsg lin_errmsg
```

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

lincpy_errmsg

```
#define lincpy_errmsg lin_errmsg
```

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

linfree_errmsg

```
#define linfree_errmsg lin_errmsg
```

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

linprt_errmsg

```
#define linprt_errmsg lin_errmsg
```

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

linset_errmsg

```
#define linset_errmsg lin_errmsg
```

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

linp2x_errmsg

```
#define linp2x_errmsg lin_errmsg
```

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

linx2p_errmsg

```
#define linx2p_errmsg lin_errmsg
```

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

6.9.3 Enumeration Type Documentation

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	

6.9.4 Function Documentation

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.

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().

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.

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.

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().

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.

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.

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.

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.

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.
- 4: Failed to initialise distortions.

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

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: `PCi_ja` matrix is singular.
- 4: Failed to initialise distortions.
- 5: Distort error.

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

Notes:
1. Historically, the API to **linp2x**() did not have a stat[] vector because a valid linear transformation should always succeed. However, now that it invokes disp2x() if distortions are present, it does have the potential to fail. Consequently, when distortions are present and a status return (stat[]) is required for each coordinate, then **linp2x**() should be invoked separately for each of them.

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.

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. • 4: Failed to initialise distortions. • 6: De-distort error. For returns > 1, a detailed error message is set in linprm::err if enabled, see wcserr_enable().

Notes:
1. Historically, the API to **linx2p**() did not have a stat[] vector because a valid linear transformation should always succeed. However, now that it invokes disx2p() if distortions are present, it does have the potential to fail. Consequently, when distortions are present and a status return (stat[]) is required for each coordinate, then **linx2p**() should be invoked separately for each of them.

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,...).

Parameters

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.

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.

6.9.5 Variable Documentation

lin_errmsg

```
const char * lin_errmsg[] [extern]
```

Error messages to match the status value returned from each function.

6.10 lin.h

Go to the documentation of this file.

```
/*============================================================================
  WCSLIB 8.1 - an implementation of the FITS WCS standard.
  Copyright (C) 1995-2023, Mark Calabretta

  This file is part of WCSLIB.

  WCSLIB is free software: you can redistribute it and/or modify it under the
  terms of the GNU Lesser General Public License as published by the Free
  Software Foundation, either version 3 of the License, or (at your option)
  any later version.

  WCSLIB is distributed in the hope that it will be useful, but WITHOUT ANY
  WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
  FOR A PARTICULAR PURPOSE.  See the GNU Lesser General Public License for
  more details.

  You should have received a copy of the GNU Lesser General Public License
  along with WCSLIB.  If not, see http://www.gnu.org/licenses.

  Author: Mark Calabretta, Australia Telescope National Facility, CSIRO.
  http://www.atnf.csiro.au/people/Mark.Calabretta
  $Id: lin.h,v 8.1 2023/07/05 17:12:07 mcalabre Exp $
*=============================================================================
*
* WCSLIB 8.1 - C routines that implement the FITS World Coordinate System
* (WCS) standard.  Refer to the README file provided with WCSLIB for an
* overview of the library.
*
*
* Summary of the lin routines
* ---------------------------
* 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.
*
*
* linini() - Default constructor for the linprm struct
* ----------------------------------------------------
* 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.
*
*
* lininit() - Default constructor for the linprm struct
* -----------------------------------------------------
* 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.
*
* Given:
*   alloc     int       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.)
*
*   naxis     int       The number of world coordinate axes, used to determine
*                       array sizes.
*
* Given and returned:
*   lin       struct linprm*
*                       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).
*
* Given:
*   ndpmax    int       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.
*
* Function return value:
*             int       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().
*
*
* lindis() - Assign a distortion to a linprm struct
* -------------------------------------------------
* 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.
*
*
* lindist() - Assign a distortion to a linprm struct
* --------------------------------------------------
* 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.
*
* Given:
*   sequence  int       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.
*
* Given and returned:
*   lin       struct linprm*
*                       Linear transformation parameters.
*
*   dis       struct disprm*
*                       Distortion function parameters.
*
* Given:
*   ndpmax    int       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.
*
* Function return value:
*             int       Status return value:
*                         0: Success.
*                         1: Null linprm pointer passed.
*                         4: Invalid sequence.
*
*
* lincpy() - Copy routine for the linprm struct
* ---------------------------------------------
* 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.
*
* Given:
*   alloc     int       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.
*
*   linsrc    const struct linprm*
*                       Struct to copy from.
*
* Given and returned:
*   lindst    struct linprm*
*                       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).
*
* Function return value:
*             int       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().
*
*
* linfree() - Destructor for the linprm struct
* --------------------------------------------
* 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().
*
* Given:
*   lin       struct linprm*
*                       Linear transformation parameters.
*
* Function return value:
*             int       Status return value:
*                         0: Success.
*                         1: Null linprm pointer passed.
*
*
* linsize() - Compute the size of a linprm struct
* -----------------------------------------------
* linsize() computes the full size of a linprm struct, including allocated
* memory.
```

```
*
* Given:
*   lin       const struct linprm*
*                       Linear transformation parameters.
*
*                       If NULL, the base size of the struct and the allocated
*                       size are both set to zero.
*
* Returned:
*   sizes     int[2]    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.
*
* Function return value:
*             int       Status return value:
*                         0: Success.
*
*
* linprt() - Print routine for the linprm struct
* ----------------------------------------------
* linprt() prints the contents of a linprm struct using wcsprintf().  Mainly
* intended for diagnostic purposes.
*
* Given:
*   lin       const struct linprm*
*                       Linear transformation parameters.
*
* Function return value:
*             int       Status return value:
*                         0: Success.
*                         1: Null linprm pointer passed.
*
*
* linperr() - Print error messages from a linprm struct
* -----------------------------------------------------
* 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.
*
* Given:
*   lin       const struct linprm*
*                       Coordinate transformation parameters.
*
*   prefix    const char *
*                       If non-NULL, each output line will be prefixed with
*                       this string.
*
* Function return value:
*             int       Status return value:
*                         0: Success.
*                         1: Null linprm pointer passed.
*
*
* linset() - Setup routine for the linprm struct
* ----------------------------------------------
* 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.
*
* Given and returned:
*   lin       struct linprm*
*                       Linear transformation parameters.
*
* Function return value:
*             int       Status return value:
*                         0: Success.
*                         1: Null linprm pointer passed.
*                         2: Memory allocation failed.
*                         3: PCi_ja matrix is singular.
*                         4: Failed to initialise distortions.
*
*                       For returns > 1, a detailed error message is set in
*                       linprm::err if enabled, see wcserr_enable().
*
```

```
*
* linp2x() - Pixel-to-world linear transformation
* -----------------------------------------------
* linp2x() transforms pixel coordinates to intermediate world coordinates.
*
* Given and returned:
*   lin       struct linprm*
*                       Linear transformation parameters.
*
* Given:
*   ncoord,
*   nelem     int       The number of coordinates, each of vector length nelem
*                       but containing lin.naxis coordinate elements.
*
*   pixcrd    const double[ncoord][nelem]
*                       Array of pixel coordinates.
*
* Returned:
*   imgcrd    double[ncoord][nelem]
*                       Array of intermediate world coordinates.
*
* Function return value:
*             int       Status return value:
*                         0: Success.
*                         1: Null linprm pointer passed.
*                         2: Memory allocation failed.
*                         3: PCi_ja matrix is singular.
*                         4: Failed to initialise distortions.
*                         5: Distort error.
*
*                       For returns > 1, a detailed error message is set in
*                       linprm::err if enabled, see wcserr_enable().
*
* Notes:
*   1. Historically, the API to linp2x() did not have a stat[] vector because
*      a valid linear transformation should always succeed.  However, now that
*      it invokes disp2x() if distortions are present, it does have the
*      potential to fail.  Consequently, when distortions are present and a
*      status return (stat[]) is required for each coordinate, then linp2x()
*      should be invoked separately for each of them.
*
*
* linx2p() - World-to-pixel linear transformation
* -----------------------------------------------
* linx2p() transforms intermediate world coordinates to pixel coordinates.
*
* Given and returned:
*   lin       struct linprm*
*                       Linear transformation parameters.
*
* Given:
*   ncoord,
*   nelem     int       The number of coordinates, each of vector length nelem
*                       but containing lin.naxis coordinate elements.
*
*   imgcrd   const double[ncoord][nelem]
*                       Array of intermediate world coordinates.
*
* Returned:
*   pixcrd    double[ncoord][nelem]
*                       Array of pixel coordinates.
*
*             int       Status return value:
*                         0: Success.
*                         1: Null linprm pointer passed.
*                         2: Memory allocation failed.
*                         3: PCi_ja matrix is singular.
*                         4: Failed to initialise distortions.
*                         6: De-distort error.
*
*                       For returns > 1, a detailed error message is set in
*                       linprm::err if enabled, see wcserr_enable().
*
* Notes:
*   1. Historically, the API to linx2p() did not have a stat[] vector because
*      a valid linear transformation should always succeed.  However, now that
*      it invokes disx2p() if distortions are present, it does have the
*      potential to fail.  Consequently, when distortions are present and a
*      status return (stat[]) is required for each coordinate, then linx2p()
*      should be invoked separately for each of them.
*
*
* linwarp() - Compute measures of distortion
* ------------------------------------------
* 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
* (PCi_ja or CDi_ja).  The difference may be significant if the matrix
* introduced a scaling.
*
* If all distortions are prior, then linwarp() uses diswarp(), q.v.
*
* Given and returned:
*   lin       struct linprm*
*                       Linear transformation parameters plus distortions.
*
* Given:
*   pixblc    const double[naxis]
*                       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,...).
*
*   pixtrc    const double[naxis]
*                       End of the range of pixel coordinates (i.e. "top
*                       right-hand corner" in the conventional FITS image
*                       display orientation).
*
*   pixsamp   const double[naxis]
*                       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.
*
* Returned:
*   nsamp     int*      The number of pixel coordinates sampled.
*
*                       Can be specified as a NULL pointer if not required.
*
*   maxdis    double[naxis]
*                       For each individual distortion function, the
*                       maximum absolute value of the distortion.
*
*                       Can be specified as a NULL pointer if not required.
*
*   maxtot    double*   For the combination of all distortion functions, the
*                       maximum absolute value of the distortion.
*
*                       Can be specified as a NULL pointer if not required.
*
*   avgdis    double[naxis]
*                       For each individual distortion function, the
*                       mean value of the distortion.
*
*                       Can be specified as a NULL pointer if not required.
*
*   avgtot    double*   For the combination of all distortion functions, the
*                       mean value of the distortion.
*
*                       Can be specified as a NULL pointer if not required.
*
*   rmsdis    double[naxis]
*                       For each individual distortion function, the
*                       root mean square deviation of the distortion.
*
*                       Can be specified as a NULL pointer if not required.
*
*   rmstot    double*   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.
*
* Function return value:
*             int       Status return value:
*                         0: Success.
*                         1: Null linprm pointer passed.
*                         2: Memory allocation failed.
*                         3: Invalid parameter.
*                         4: Distort error.
*
*
* linprm struct - Linear transformation parameters
```

```
* -----------------------------------------------
* 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").
*
*   int 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.
*
*   int 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.
*
*   double *crpix
*     (Given) Pointer to the first element of an array of double containing
*     the coordinate reference pixel, CRPIXja.
*
*     It is not necessary to reset the linprm struct (via linset()) when
*     linprm::crpix is changed.
*
*   double *pc
*     (Given) Pointer to the first element of the PCi_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.
*
*   double *cdelt
*     (Given) Pointer to the first element of an array of double containing
*     the coordinate increments, CDELTia.
*
*   struct disprm *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.
*
*   struct disprm *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.
*
*   double *piximg
*     (Returned) Pointer to the first element of the matrix containing the
*     product of the CDELTia diagonal matrix and the PCi_ja matrix.
*
*   double *imgpix
*     (Returned) Pointer to the first element of the inverse of the
*     linprm::piximg matrix.
*
*   int i_naxis
*     (Returned) The dimension of linprm::piximg and linprm::imgpix (normally
*     equal to naxis).
*
*   int unity
*     (Returned) True if the linear transformation matrix is unity.
*
*   int affine
*     (Returned) True if there are no distortions.
*
*   int simple
*     (Returned) True if unity and no distortions.
*
*   struct wcserr *err
*     (Returned) If enabled, when an error status is returned, this struct
*     contains detailed information about the error, see wcserr_enable().
*
*   double *tmpcrd
*     (For internal use only.)
*   int m_flag
*     (For internal use only.)
*   int m_naxis
*     (For internal use only.)
*   double *m_crpix
*     (For internal use only.)
*   double *m_pc
*     (For internal use only.)
*   double *m_cdelt
*     (For internal use only.)
*   struct disprm *m_dispre
*     (For internal use only.)
*   struct disprm *m_disseq
*     (For internal use only.)
*
*
* Global variable: const char *lin_errmsg[] - Status return messages
* --------------------------------------------------------------------
* Error messages to match the status value returned from each function.
*
*===========================================================================*/

#ifndef WCSLIB_LIN
#define WCSLIB_LIN

#ifdef __cplusplus
```

```
extern "C" {
#endif


extern const char *lin_errmsg[];

enum lin_errmsg_enum {
  LINERR_SUCCESS      = 0,        // Success.
  LINERR_NULL_POINTER = 1,    // Null linprm pointer passed.
  LINERR_MEMORY       = 2,  // Memory allocation failed.
  LINERR_SINGULAR_MTX = 3,    // PCi_ja matrix is singular.
  LINERR_DISTORT_INIT = 4,    // Failed to initialise distortions.
  LINERR_DISTORT      = 5,        // Distort error.
  LINERR_DEDISTORT    = 6       // De-distort error.
};

struct linprm {
  // Initialization flag (see the prologue above).
  //-----------------------------------------------------------------------
  int flag;                  // Set to zero to force initialization.

  // Parameters to be provided (see the prologue above).
  //-----------------------------------------------------------------------
  int naxis;                         // The number of axes, given by NAXIS.
  double *crpix;            // CRPIXja keywords for each pixel axis.
  double *pc;                  // PCi_ja  linear transformation matrix.
  double *cdelt;            // CDELTia keywords for each coord axis.
  struct disprm *dispre;     // Prior   distortion parameters, if any.
  struct disprm *disseq;     // Sequent distortion parameters, if any.

  // Information derived from the parameters supplied.
  //-----------------------------------------------------------------------
  double *piximg;          // Product of CDELTia and PCi_ja matrices.
  double *imgpix;          // Inverse of the piximg matrix.
  int    i_naxis;                 // Dimension of piximg and imgpix.
  int    unity;                     // True if the PCi_ja matrix is unity.
  int    affine;           // True if there are no distortions.
  int    simple;           // True if unity and no distortions.

  // Error handling, if enabled.
  //-----------------------------------------------------------------------
  struct wcserr *err;

  // Private - the remainder are for internal use.
  //-----------------------------------------------------------------------
  double *tmpcrd;

  int    m_flag, m_naxis;
  double *m_crpix, *m_pc, *m_cdelt;
  struct disprm *m_dispre, *m_disseq;
};

// Size of the linprm struct in int units, used by the Fortran wrappers.
#define LINLEN (sizeof(struct linprm)/sizeof(int))


int linini(int alloc, int naxis, struct linprm *lin);

int lininit(int alloc, int naxis, struct linprm *lin, int ndpmax);

int lindis(int sequence, struct linprm *lin, struct disprm *dis);

int lindist(int sequence, struct linprm *lin, struct disprm *dis, int ndpmax);

int lincpy(int alloc, const struct linprm *linsrc, struct linprm *lindst);

int linfree(struct linprm *lin);

int linsize(const struct linprm *lin, int sizes[2]);

int linprt(const struct linprm *lin);

int linperr(const struct linprm *lin, const char *prefix);

int linset(struct linprm *lin);

int linp2x(struct linprm *lin, int ncoord, int nelem, const double pixcrd[],
           double imgcrd[]);

int linx2p(struct linprm *lin, int ncoord, int nelem, const double imgcrd[],
           double pixcrd[]);

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);
```

```
int matinv(int n, const double mat[], double inv[]);


// Deprecated.
#define linini_errmsg lin_errmsg
#define lincpy_errmsg lin_errmsg
#define linfree_errmsg lin_errmsg
#define linprt_errmsg lin_errmsg
#define linset_errmsg lin_errmsg
#define linp2x_errmsg lin_errmsg
#define linx2p_errmsg lin_errmsg

#ifdef __cplusplus
}
#endif

#endif // WCSLIB_LIN
```

6.11 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.

6.11.1 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.

6.11.2 Enumeration Type Documentation

log_errmsg_enum

```
enum log_errmsg_enum
```

Enumerator

LOGERR_SUCCESS	
LOGERR_NULL_POINTER	
LOGERR_BAD_LOG_REF_VAL	
LOGERR_BAD_X	
LOGERR_BAD_WORLD	

6.11.3 Function Documentation

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.

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.

6.11.4 Variable Documentation

log_errmsg

```
const char * log_errmsg[] [extern]
```

Error messages to match the status value returned from each function.

6.12 log.h

Go to the documentation of this file.

```
/*============================================================================
  WCSLIB 8.1 - an implementation of the FITS WCS standard.
  Copyright (C) 1995-2023, Mark Calabretta

  This file is part of WCSLIB.

```

```
  WCSLIB is free software: you can redistribute it and/or modify it under the
  terms of the GNU Lesser General Public License as published by the Free
  Software Foundation, either version 3 of the License, or (at your option)
  any later version.

  WCSLIB is distributed in the hope that it will be useful, but WITHOUT ANY
  WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
  FOR A PARTICULAR PURPOSE.  See the GNU Lesser General Public License for
  more details.

  You should have received a copy of the GNU Lesser General Public License
  along with WCSLIB.  If not, see http://www.gnu.org/licenses.

  Author: Mark Calabretta, Australia Telescope National Facility, CSIRO.
  http://www.atnf.csiro.au/people/Mark.Calabretta
  $Id: log.h,v 8.1 2023/07/05 17:12:07 mcalabre Exp $
*=============================================================================
*
* WCSLIB 8.1 - C routines that implement the FITS World Coordinate System
* (WCS) standard.  Refer to the README file provided with WCSLIB for an
* overview of the library.
*
*
* Summary of the log routines
* ---------------------------
* 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.
*
*
* logx2s() - Transform to logarithmic coordinates
* -----------------------------------------------
* logx2s() transforms intermediate world coordinates to logarithmic
* coordinates.
*
* Given and returned:
*   crval     double    Log-coordinate reference value (CRVALia).
*
* Given:
*   nx        int       Vector length.
*
*   sx        int       Vector stride.
*
*   slogc     int       Vector stride.
*
*   x         const double[]
*                       Intermediate world coordinates, in SI units.
*
* Returned:
*   logc      double[]  Logarithmic coordinates, in SI units.
*
*   stat      int[]     Status return value status for each vector element:
*                         0: Success.
*
* Function return value:
*             int       Status return value:
*                         0: Success.
*                         2: Invalid log-coordinate reference value.
*
```

```
*
* logs2x() - Transform logarithmic coordinates
* ---------------------------------------------
* logs2x() transforms logarithmic world coordinates to intermediate world
* coordinates.
*
* Given and returned:
*   crval     double    Log-coordinate reference value (CRVALia).
*
* Given:
*   nlogc     int       Vector length.
*
*   slogc     int       Vector stride.
*
*   sx        int       Vector stride.
*
*   logc      const double[]
*                       Logarithmic coordinates, in SI units.
*
* Returned:
*   x         double[]  Intermediate world coordinates, in SI units.
*
*   stat      int[]     Status return value status for each vector element:
*                         0: Success.
*                         1: Invalid value of logc.
*
* Function return value:
*             int       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.
*
*
* Global variable: const char *log_errmsg[] - Status return messages
* ------------------------------------------------------------------
* Error messages to match the status value returned from each function.
*
*===========================================================================*/

#ifndef WCSLIB_LOG
#define WCSLIB_LOG

#ifdef __cplusplus
extern "C" {
#endif

extern const char *log_errmsg[];

enum log_errmsg_enum {
  LOGERR_SUCCESS          = 0,     // Success.
  LOGERR_NULL_POINTER     = 1,        // Null pointer passed.
  LOGERR_BAD_LOG_REF_VAL  = 2,     // Invalid log-coordinate reference value.
  LOGERR_BAD_X            = 3,       // One or more of the x coordinates were
                                 // invalid.
  LOGERR_BAD_WORLD        = 4    // One or more of the world coordinates were
                                 // invalid.
};

int logx2s(double crval, int nx, int sx, int slogc, const double x[],
           double logc[], int stat[]);

int logs2x(double crval, int nlogc, int slogc, int sx, const double logc[],
           double x[], int stat[]);


#ifdef __cplusplus
}
#endif

#endif // WCSLIB_LOG
```

6.13 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.

6.13.1 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 $[-180^\circ, 180^\circ]$ for ϕ, and $[-90^\circ, 90^\circ]$ 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 $[-90^\circ, 90^\circ]$. 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 $[-180^\circ, 180^\circ]$ for ϕ, and $[-90^\circ, 90^\circ]$ 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 $0^\circ.0000000001$ 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.

6.13.2 Macro Definition Documentation

PVN

```
#define PVN 30
```

The total number of projection parameters numbered 0 to **PVN**-1.

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.

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.

PRJLEN

```
#define PRJLEN (sizeof(struct prjprm)/sizeof(int))
```

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

prjini_errmsg

```
#define prjini_errmsg prj_errmsg
```

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

prjprt_errmsg

```
#define prjprt_errmsg prj_errmsg
```

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

prjset_errmsg

```
#define prjset_errmsg prj_errmsg
```

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

prjx2s_errmsg

```
#define prjx2s_errmsg prj_errmsg
```

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

prjs2x_errmsg

```
#define prjs2x_errmsg prj_errmsg
```

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

6.13.3 Enumeration Type Documentation

prj_errmsg_enum

```
enum prj_errmsg_enum
```

Enumerator

PRJERR_SUCCESS	
PRJERR_NULL_POINTER	
PRJERR_BAD_PARAM	
PRJERR_BAD_PIX	
PRJERR_BAD_WORLD	

6.13.4 Function Documentation

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.

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.

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.

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.

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.

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.

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().

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().

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().

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

xphset()

```
int xphset (
            struct prjprm * prj )
```

xphx2s()

```
int xphx2s (
            PRJX2S_ARGS  )
```

xphs2x()

```
int xphs2x (
            PRJS2X_ARGS  )
```

6.13.5 Variable Documentation

prj_errmsg

```
const char * prj_errmsg[ ]  [extern]
```

Error messages to match the status value returned from each function.

CONIC

```
const int CONIC  [extern]
```

Identifier for conic projections, see prjprm::category.

CONVENTIONAL

```
const int CONVENTIONAL
```

Identifier for conventional projections, see prjprm::category.

CYLINDRICAL

```
const int CYLINDRICAL
```

Identifier for cylindrical projections, see prjprm::category.

POLYCONIC

```
const int POLYCONIC
```

Identifier for polyconic projections, see prjprm::category.

PSEUDOCYLINDRICAL

```
const int PSEUDOCYLINDRICAL
```

Identifier for pseudocylindrical projections, see prjprm::category.

QUADCUBE

```
const int QUADCUBE
```

Identifier for quadcube projections, see prjprm::category.

ZENITHAL

```
const int ZENITHAL
```

Identifier for zenithal/azimuthal projections, see prjprm::category.

HEALPIX

```
const int HEALPIX
```

Identifier for the HEALPix projection, see prjprm::category.

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.

prj_ncode

```
const int prj_ncode  [extern]
```

The number of recognized three-letter projection codes (currently 27), see prj_codes.

prj_codes

```
const char prj_codes[27][4]  [extern]
```

List of all recognized three-letter projection codes (currently 27), e.g. **SIN**, **TAN**, etc.

6.14 prj.h

Go to the documentation of this file.

```
/*============================================================================
  WCSLIB 8.1 - an implementation of the FITS WCS standard.
  Copyright (C) 1995-2023, Mark Calabretta

  This file is part of WCSLIB.

  WCSLIB is free software: you can redistribute it and/or modify it under the
  terms of the GNU Lesser General Public License as published by the Free
  Software Foundation, either version 3 of the License, or (at your option)
  any later version.

  WCSLIB is distributed in the hope that it will be useful, but WITHOUT ANY
  WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
  FOR A PARTICULAR PURPOSE.  See the GNU Lesser General Public License for
  more details.

  You should have received a copy of the GNU Lesser General Public License
  along with WCSLIB.  If not, see http://www.gnu.org/licenses.

  Author: Mark Calabretta, Australia Telescope National Facility, CSIRO.
  http://www.atnf.csiro.au/people/Mark.Calabretta
  $Id: prj.h,v 8.1 2023/07/05 17:12:07 mcalabre Exp $
*=============================================================================
*
* WCSLIB 8.1 - C routines that implement the FITS World Coordinate System
* (WCS) standard.  Refer to the README file provided with WCSLIB for an
* overview of the library.
*
*
* Summary of the prj routines
* ---------------------------
* 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 carree)
*   - 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 phi and theta (the native longitude and latitude) normally lie
* in the range [-180,180] for phi, and [-90,90] for theta.  However, all
* projection routines will accept any value of phi and will not normalize it.
*
* The projection routines do not explicitly check that theta lies within the
* range [-90,90].  They do check for any value of theta 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 (phi,theta) 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 phi and theta obtained
* lie within the ranges [-180,180] for phi, and [-90,90] for theta.  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 1E-10 degree of longitude and latitude has been verified for typical
* projection parameters on the 1 degree graticule of native longitude and
* latitude (to within 5 degrees of any latitude where the projection may
* diverge).  Refer to the tprj1.c and tprj2.c test routines that accompany
* this software.
*
*
* prjini() - Default constructor for the prjprm struct
* ----------------------------------------------------
* 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.
*
* Returned:
*   prj       struct prjprm*
*                       Projection parameters.
*
* Function return value:
*             int       Status return value:
*                         0: Success.
*                         1: Null prjprm pointer passed.
*
*
* prjfree() - Destructor for the prjprm struct
* --------------------------------------------
* prjfree() frees any memory that may have been allocated to store an error
* message in the prjprm struct.
*
* Given:
*   prj       struct prjprm*
*                       Projection parameters.
*
* Function return value:
*             int       Status return value:
*                         0: Success.
*                         1: Null prjprm pointer passed.
*
*
* prjsize() - Compute the size of a prjprm struct
* -----------------------------------------------
* prjsize() computes the full size of a prjprm struct, including allocated
* memory.
*
* Given:
*   prj       const struct prjprm*
*                       Projection parameters.
*
*                       If NULL, the base size of the struct and the allocated
*                       size are both set to zero.
*
* Returned:
*   sizes     int[2]    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().
*
* Function return value:
*             int       Status return value:
*                         0: Success.
*
*
* prjprt() - Print routine for the prjprm struct
* ----------------------------------------------
* prjprt() prints the contents of a prjprm struct using wcsprintf().  Mainly
* intended for diagnostic purposes.
*
* Given:
```

```
*   prj       const struct prjprm*
*                       Projection parameters.
*
* Function return value:
*             int       Status return value:
*                         0: Success.
*                         1: Null prjprm pointer passed.
*
*
* prjperr() - Print error messages from a prjprm struct
* -----------------------------------------------------
* 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.
*
* Given:
*   prj       const struct prjprm*
*                       Projection parameters.
*
*   prefix    const char *
*                       If non-NULL, each output line will be prefixed with
*                       this string.
*
* Function return value:
*             int       Status return value:
*                         0: Success.
*                         1: Null prjprm pointer passed.
*
*
* prjbchk() - Bounds checking on native coordinates
* -------------------------------------------------
* 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.
*
* Given:
*   tol       double    Tolerance for the bounds check [deg].
*
*   nphi,
*   ntheta    int       Vector lengths.
*
*   spt       int       Vector stride.
*
* Given and returned:
*   phi,theta double[]  Native longitude and latitude (phi,theta) [deg].
*
* Returned:
*   stat      int[]     Status value for each vector element:
*                         0: Valid value of (phi,theta).
*                         1: Invalid value.
*
* Function return value:
*             int       Status return value:
*                         0: Success.
*                         1: One or more of the (phi,theta) coordinates
*                            were, invalid, as indicated by the stat vector.
*
*
* prjset() - Generic setup routine for the prjprm struct
* ------------------------------------------------------
* 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.
*
* Given and returned:
*   prj       struct prjprm*
*                       Projection parameters.
*
* Function return value:
*             int       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().
*
*
* prjx2s() - Generic Cartesian-to-spherical deprojection
* ------------------------------------------------------
* Deproject Cartesian (x,y) coordinates in the plane of projection to native
* spherical coordinates (phi,theta).
*
* The projection is that specified by prjprm::code.
*
* Given and returned:
*   prj       struct prjprm*
*                      Projection parameters.
*
* Given:
*   nx,ny     int       Vector lengths.
*
*   sxy,spt   int       Vector strides.
*
*   x,y       const double[]
*                      Projected coordinates.
*
* Returned:
*   phi,theta double[]  Longitude and latitude (phi,theta) of the projected
*                      point in native spherical coordinates [deg].
*
*   stat      int[]     Status value for each vector element:
*                        0: Success.
*                        1: Invalid value of (x,y).
*
* Function return value:
*             int       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().
*
*
* prjs2x() - Generic spherical-to-Cartesian projection
* ----------------------------------------------------
* Project native spherical coordinates (phi,theta) to Cartesian (x,y)
* coordinates in the plane of projection.
*
* The projection is that specified by prjprm::code.
*
* Given and returned:
*   prj       struct prjprm*
*                      Projection parameters.
*
* Given:
*   nphi,
*   ntheta    int       Vector lengths.
*
*   spt,sxy   int       Vector strides.
*
*   phi,theta const double[]
*                      Longitude and latitude (phi,theta) of the projected
*                      point in native spherical coordinates [deg].
*
* Returned:
*   x,y       double[]  Projected coordinates.
*
*   stat      int[]     Status value for each vector element:
*                        0: Success.
*                        1: Invalid value of (phi,theta).
*
* Function return value:
*             int       Status return value:
*                        0: Success.
*                        1: Null prjprm pointer passed.
*                        2: Invalid projection parameters.
*                        4: One or more of the (phi,theta) 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().
*
*
```

```
* ???set() - Specific setup routines for the prjprm struct
* -----------------------------------------------------------
* Set up a prjprm struct for a particular projection according to information
* supplied within it.
*
* Given and returned:
*   prj       struct prjprm*
*                       Projection parameters.
*
* Function return value:
*             int       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().
*
*
* ???x2s() - Specific Cartesian-to-spherical deprojection routines
* -------------------------------------------------------------------
* Transform (x,y) coordinates in the plane of projection to native spherical
* coordinates (phi,theta).
*
* Given and returned:
*   prj       struct prjprm*
*                       Projection parameters.
*
* Given:
*   nx,ny     int       Vector lengths.
*
*   sxy,spt   int       Vector strides.
*
*   x,y       const double[]
*                       Projected coordinates.
*
* Returned:
*   phi,theta double[]  Longitude and latitude of the projected point in
*                       native spherical coordinates [deg].
*
*   stat      int[]     Status value for each vector element:
*                         0: Success.
*                         1: Invalid value of (x,y).
*
* Function return value:
*             int       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().
*
*
* ???s2x() - Specific spherical-to-Cartesian projection routines
*---------------------------------------------------------------
* Transform native spherical coordinates (phi,theta) to (x,y) coordinates in
* the plane of projection.
*
* Given and returned:
*   prj       struct prjprm*
*                       Projection parameters.
*
* Given:
*   nphi,
*   ntheta    int       Vector lengths.
*
*   spt,sxy   int       Vector strides.
*
*   phi,theta const double[]
*                       Longitude and latitude of the projected point in
*                       native spherical coordinates [deg].
*
* Returned:
*   x,y       double[]  Projected coordinates.
*
*   stat      int[]     Status value for each vector element:
*                         0: Success.
*                         1: Invalid value of (phi,theta).
*
* Function return value:
*             int       Status return value:
*                         0: Success.
*                         1: Null prjprm pointer passed.
*                         2: Invalid projection parameters.
```

```
*                         4: One or more of the (phi,theta) 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().
*
*
* prjprm struct - Projection parameters
* -------------------------------------
* 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.
*
*   int 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.
*
*   char code[4]
*     (Given) Three-letter projection code defined by the FITS standard.
*
*   double 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).
*
*   double pv[30]
*     (Given) Projection parameters.  These correspond to the PVi_ma keywords
*     in FITS, so pv[0] is PVi_0a, pv[1] is PVi_1a, etc., where i denotes the
*     latitude-like axis.  Many projections use pv[1] (PVi_1a), some also use
*     pv[2] (PVi_2a) and SZP uses pv[3] (PVi_3a).  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.
*
*   double phi0
*     (Given) The native longitude, phi_0 [deg], and ...
*   double theta0
*     (Given) ... the native latitude, theta_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.
*
*   int 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:
*
*   char name[40]
*     (Returned) Long name of the projection.
*
*     Provided for information only, not used by the projection routines.
*
*   int  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.
*
*   int  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.
*
*   int  simplezen
*     (Returned) True if the projection is a radially-symmetric zenithal
*     projection.
*
*     Provided for information only, not used by the projection routines.
*
*   int  equiareal
*     (Returned) True if the projection is equal area.
*
*     Provided for information only, not used by the projection routines.
*
*   int  conformal
*     (Returned) True if the projection is conformal.
*
*     Provided for information only, not used by the projection routines.
*
*   int  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.
*
*   int  divergent
*     (Returned) True if the projection diverges in latitude.
*
*     Provided for information only, not used by the projection routines.
*
*   double x0
*     (Returned) The offset in x, and ...
*   double y0
*     (Returned) ... the offset in y used to force (x,y) = (0,0) at
*     (phi_0,theta_0).
*
*   struct wcserr *err
*     (Returned) If enabled, when an error status is returned, this struct
*     contains detailed information about the error, see wcserr_enable().
*
*   void *padding
*     (An unused variable inserted for alignment purposes only.)
*
*   double w[10]
*     (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.
*
*   int n
*     (Returned) Intermediate integer value (used only for the ZPN and HPX
*     projections).
*
*   int (*prjx2s)(PRJX2S_ARGS)
*     (Returned) Pointer to the spherical projection ...
*   int (*prjs2x)(PRJ_ARGS)
*     (Returned) ... and deprojection routines.
*
*
* Global variable: const char *prj_errmsg[] - Status return messages
* -------------------------------------------------------------------
* Error messages to match the status value returned from each function.
*
*===========================================================================*/

#ifndef WCSLIB_PROJ
#define WCSLIB_PROJ

```

```
#ifdef __cplusplus
extern "C" {
#endif


// Total number of projection parameters; 0 to PVN-1.
#define PVN 30

extern const char *prj_errmsg[];

enum prj_errmsg_enum {
  PRJERR_SUCCESS      = 0,        // Success.
  PRJERR_NULL_POINTER = 1,   // Null prjprm pointer passed.
  PRJERR_BAD_PARAM    = 2,      // Invalid projection parameters.
  PRJERR_BAD_PIX      = 3,        // One or more of the (x, y) coordinates were
                              // invalid.
  PRJERR_BAD_WORLD    = 4       // One or more of the (phi, theta) coordinates
                              // were invalid.
};

extern const int CONIC, CONVENTIONAL, CYLINDRICAL, POLYCONIC,
                 PSEUDOCYLINDRICAL, QUADCUBE, ZENITHAL, HEALPIX;
extern const char prj_categories[9][32];

extern const int  prj_ncode;
extern const char prj_codes[28][4];

#ifdef PRJX2S_ARGS
#undef PRJX2S_ARGS
#endif

#ifdef PRJS2X_ARGS
#undef PRJS2X_ARGS
#endif

// For use in declaring deprojection function prototypes.
#define PRJX2S_ARGS struct prjprm *prj, int nx, int ny, int sxy, int spt, \
const double x[], const double y[], double phi[], double theta[], int stat[]

// For use in declaring projection function prototypes.
#define PRJS2X_ARGS struct prjprm *prj, int nx, int ny, int sxy, int spt, \
const double phi[], const double theta[], double x[], double y[], int stat[]


struct prjprm {
  // Initialization flag (see the prologue above).
  //-----------------------------------------------------------------------------
  int    flag;                       // Set to zero to force initialization.

  // Parameters to be provided (see the prologue above).
  //-----------------------------------------------------------------------------
  char   code[4];            // Three-letter projection code.
  double r0;                   // Radius of the generating sphere.
  double pv[PVN];           // Projection parameters.
  double phi0, theta0;                   // Fiducial native coordinates.
  int    bounds;           // Controls bounds checking.

  // Information derived from the parameters supplied.
  //-----------------------------------------------------------------------------
  char   name[40];           // Projection name.
  int    category;               // Projection category.
  int    pvrange;                 // Range of projection parameter indices.
  int    simplezen;             // Is it a simple zenithal projection?
  int    equiareal;             // Is it an equal area projection?
  int    conformal;             // Is it a conformal projection?
  int    global;           // Can it map the whole sphere?
  int    divergent;             // Does the projection diverge in latitude?
  double x0, y0;             // Fiducial offsets.

  // Error handling
  //-----------------------------------------------------------------------------
  struct wcserr *err;

  // Private
  //-----------------------------------------------------------------------------
  void   *padding;                // (Dummy inserted for alignment purposes.)
  double w[10];                         // Intermediate values.
  int    m, n;                         // Intermediate values.

  int (*prjx2s)(PRJX2S_ARGS);  // Pointers to the spherical projection and
  int (*prjs2x)(PRJS2X_ARGS);  // deprojection functions.
};

// Size of the prjprm struct in int units, used by the Fortran wrappers.
#define PRJLEN (sizeof(struct prjprm)/sizeof(int))


```

```
int prjini(struct prjprm *prj);

int prjfree(struct prjprm *prj);

int prjsize(const struct prjprm *prj, int sizes[2]);

int prjprt(const struct prjprm *prj);

int prjperr(const struct prjprm *prj, const char *prefix);

int prjbchk(double tol, int nphi, int ntheta, int spt, double phi[],
            double theta[], int stat[]);

// Use the preprocessor to help declare function prototypes (see above).
int prjset(struct prjprm *prj);
int prjx2s(PRJX2S_ARGS);
int prjs2x(PRJS2X_ARGS);

int azpset(struct prjprm *prj);
int azpx2s(PRJX2S_ARGS);
int azps2x(PRJS2X_ARGS);

int szpset(struct prjprm *prj);
int szpx2s(PRJX2S_ARGS);
int szps2x(PRJS2X_ARGS);

int tanset(struct prjprm *prj);
int tanx2s(PRJX2S_ARGS);
int tans2x(PRJS2X_ARGS);

int stgset(struct prjprm *prj);
int stgx2s(PRJX2S_ARGS);
int stgs2x(PRJS2X_ARGS);

int sinset(struct prjprm *prj);
int sinx2s(PRJX2S_ARGS);
int sins2x(PRJS2X_ARGS);

int arcset(struct prjprm *prj);
int arcx2s(PRJX2S_ARGS);
int arcs2x(PRJS2X_ARGS);

int zpnset(struct prjprm *prj);
int zpnx2s(PRJX2S_ARGS);
int zpns2x(PRJS2X_ARGS);

int zeaset(struct prjprm *prj);
int zeax2s(PRJX2S_ARGS);
int zeas2x(PRJS2X_ARGS);

int airset(struct prjprm *prj);
int airx2s(PRJX2S_ARGS);
int airs2x(PRJS2X_ARGS);

int cypset(struct prjprm *prj);
int cypx2s(PRJX2S_ARGS);
int cyps2x(PRJS2X_ARGS);

int ceaset(struct prjprm *prj);
int ceax2s(PRJX2S_ARGS);
int ceas2x(PRJS2X_ARGS);

int carset(struct prjprm *prj);
int carx2s(PRJX2S_ARGS);
int cars2x(PRJS2X_ARGS);

int merset(struct prjprm *prj);
int merx2s(PRJX2S_ARGS);
int mers2x(PRJS2X_ARGS);

int sflset(struct prjprm *prj);
int sflx2s(PRJX2S_ARGS);
int sfls2x(PRJS2X_ARGS);

int parset(struct prjprm *prj);
int parx2s(PRJX2S_ARGS);
int pars2x(PRJS2X_ARGS);

int molset(struct prjprm *prj);
int molx2s(PRJX2S_ARGS);
int mols2x(PRJS2X_ARGS);

int aitset(struct prjprm *prj);
int aitx2s(PRJX2S_ARGS);
int aits2x(PRJS2X_ARGS);

int copset(struct prjprm *prj);
```

```
int copx2s(PRJX2S_ARGS);
int cops2x(PRJS2X_ARGS);

int coeset(struct prjprm *prj);
int coex2s(PRJX2S_ARGS);
int coes2x(PRJS2X_ARGS);

int codset(struct prjprm *prj);
int codx2s(PRJX2S_ARGS);
int cods2x(PRJS2X_ARGS);

int cooset(struct prjprm *prj);
int coox2s(PRJX2S_ARGS);
int coos2x(PRJS2X_ARGS);

int bonset(struct prjprm *prj);
int bonx2s(PRJX2S_ARGS);
int bons2x(PRJS2X_ARGS);

int pcoset(struct prjprm *prj);
int pcox2s(PRJX2S_ARGS);
int pcos2x(PRJS2X_ARGS);

int tscset(struct prjprm *prj);
int tscx2s(PRJX2S_ARGS);
int tscs2x(PRJS2X_ARGS);

int cscset(struct prjprm *prj);
int cscx2s(PRJX2S_ARGS);
int cscs2x(PRJS2X_ARGS);

int qscset(struct prjprm *prj);
int qscx2s(PRJX2S_ARGS);
int qscs2x(PRJS2X_ARGS);

int hpxset(struct prjprm *prj);
int hpxx2s(PRJX2S_ARGS);
int hpxs2x(PRJS2X_ARGS);

int xphset(struct prjprm *prj);
int xphx2s(PRJX2S_ARGS);
int xphs2x(PRJS2X_ARGS);


// Deprecated.
#define prjini_errmsg prj_errmsg
#define prjprt_errmsg prj_errmsg
#define prjset_errmsg prj_errmsg
#define prjx2s_errmsg prj_errmsg
#define prjs2x_errmsg prj_errmsg

#ifdef __cplusplus
}
#endif

#endif // WCSLIB_PROJ
```

6.15 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.

6.15.1 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.

6.15.2 Macro Definition Documentation

SPCLEN

```
#define SPCLEN (sizeof(struct spcprm)/sizeof(int))
```

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

spcini_errmsg

```
#define spcini_errmsg spc_errmsg
```

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

spcprt_errmsg

```
#define spcprt_errmsg spc_errmsg
```

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

spcset_errmsg

```
#define spcset_errmsg spc_errmsg
```

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

spcx2s_errmsg

```
#define spcx2s_errmsg spc_errmsg
```

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

spcs2x_errmsg

```
#define spcs2x_errmsg spc_errmsg
```

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

6.15.3 Enumeration Type Documentation

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	

6.15.4 Function Documentation

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.

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.

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. It is not an error for the struct not to have been set up via spcset().

Returns

Status return value:

- 0: Success.

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.

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.

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().

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().

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().

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**ia 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**ia 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).

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.

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.

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.

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

<table>
<tr><td>in</td><td>ctypeA</td><td>CTYPEia keyvalue possibly containing an AIPS-convention spectral code (eight characters, need not be null-terminated).</td></tr>
<tr><td>in</td><td>velref</td><td>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 CTYPEia in defining the Doppler frame, e.g.
<code>ctypeA = 'VELO-HEL'   velref = 1</code>
returns ctype = 'VOPT' with specsys set to 'LSRK'.
If omitted from the header, the default value of VELREF is 0.</td></tr>
<tr><td>out</td><td>ctype</td><td>Translated CTYPEia keyvalue, or a copy of ctypeA if no translation was performed (in which case any trailing blanks in ctypeA will be replaced with nulls).</td></tr>
<tr><td>out</td><td>specsys</td><td>Doppler reference frame indicated by VELREF or else by CTYPEia 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.</td></tr>
</table>

Returns

Status return value:

- -1: No translation required (not an error).
- 0: Success.
- 2: Invalid value of **VELREF**.

spctyp()

```
int spctyp (
            const char ctype[9],
            char stype[],
            char scode[],
            char sname[],
            char units[],
            char * ptype,
            char * xtype,
            int * restreq )
```

spcspx()

```
int spcspx (
            const char ctypeS[9],
            double crvalS,
            double restfrq,
            double restwav,
            char * ptype,
            char * xtype,
            int * restreq,
            double * crvalX,
            double * dXdS )
```

spcxps()

```
int spcxps (
            const char ctypeS[9],
            double crvalX,
            double restfrq,
            double restwav,
            char * ptype,
            char * xtype,
            int * restreq,
            double * crvalS,
            double * dSdX )
```

spctrn()

```
int spctrn (
            const char ctypeS1[9],
            double crvalS1,
            double cdeltS1,
            double restfrq,
            double restwav,
            char ctypeS2[9],
            double * crvalS2,
            double * cdeltS2 )
```

6.15.5 Variable Documentation

spc_errmsg

`const char * spc_errmsg[] [extern]`

Error messages to match the status value returned from each function.

6.16 spc.h

Go to the documentation of this file.

```
/*============================================================================
  WCSLIB 8.1 - an implementation of the FITS WCS standard.
  Copyright (C) 1995-2023, Mark Calabretta

  This file is part of WCSLIB.

  WCSLIB is free software: you can redistribute it and/or modify it under the
  terms of the GNU Lesser General Public License as published by the Free
  Software Foundation, either version 3 of the License, or (at your option)
  any later version.

  WCSLIB is distributed in the hope that it will be useful, but WITHOUT ANY
  WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
  FOR A PARTICULAR PURPOSE.  See the GNU Lesser General Public License for
  more details.

  You should have received a copy of the GNU Lesser General Public License
  along with WCSLIB.  If not, see http://www.gnu.org/licenses.

  Author: Mark Calabretta, Australia Telescope National Facility, CSIRO.
  http://www.atnf.csiro.au/people/Mark.Calabretta
  $Id: spc.h,v 8.1 2023/07/05 17:12:07 mcalabre Exp $
*=============================================================================
*
* WCSLIB 8.1 - C routines that implement the FITS World Coordinate System
* (WCS) standard.  Refer to the README file provided with WCSLIB for an
* overview of the library.
*
*
* Summary of the spc routines
* ---------------------------
* 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 CTYPEia 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 CTYPEia.
*
*   - spcaips() translates AIPS-convention spectral CTYPEia 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 CTYPEia keyvalue,
*     and CRVALia and CDELTia 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
*     CTYPEia.
*
*   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
*     CTYPEia.
*
*   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 CTYPEia, 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 CTYPEia
* 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 CTYPEia 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.
*
*
* spcini() - Default constructor for the spcprm struct
* ----------------------------------------------------
* 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.
*
* Given and returned:
*   spc       struct spcprm*
*                       Spectral transformation parameters.
*
* Function return value:
*             int       Status return value:
*                         0: Success.
*                         1: Null spcprm pointer passed.
*
*
* spcfree() - Destructor for the spcprm struct
* --------------------------------------------
* spcfree() frees any memory that may have been allocated to store an error
* message in the spcprm struct.
*
* Given:
*   spc       struct spcprm*
*                       Spectral transformation parameters.
*
* Function return value:
*             int       Status return value:
*                         0: Success.
*                         1: Null spcprm pointer passed.
*
*
* spcsize() - Compute the size of a spcprm struct
* -----------------------------------------------
* spcsize() computes the full size of a spcprm struct, including allocated
* memory.
*
* Given:
*   spc       const struct spcprm*
*                       Spectral transformation parameters.
*
*                       If NULL, the base size of the struct and the allocated
*                       size are both set to zero.
*
* Returned:
*   sizes     int[2]    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.
*
*                       It is not an error for the struct not to have been set
*                       up via spcset().
*
* Function return value:
*             int       Status return value:
*                         0: Success.
*
*
* spcprt() - Print routine for the spcprm struct
```

```
* ---------------------------------------------
* spcprt() prints the contents of a spcprm struct using wcsprintf().  Mainly
* intended for diagnostic purposes.
*
* Given:
*   spc       const struct spcprm*
*                       Spectral transformation parameters.
*
* Function return value:
*             int       Status return value:
*                         0: Success.
*                         1: Null spcprm pointer passed.
*
*
* spcperr() - Print error messages from a spcprm struct
* -----------------------------------------------------
* 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.
*
* Given:
*   spc       const struct spcprm*
*                       Spectral transformation parameters.
*
*   prefix    const char *
*                       If non-NULL, each output line will be prefixed with
*                       this string.
*
* Function return value:
*             int       Status return value:
*                         0: Success.
*                         1: Null spcprm pointer passed.
*
*
* spcset() - Setup routine for the spcprm struct
* ----------------------------------------------
* 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.
*
* Given and returned:
*   spc       struct spcprm*
*                       Spectral transformation parameters.
*
* Function return value:
*             int       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().
*
*
* spcx2s() - Transform to spectral coordinates
* --------------------------------------------
* spcx2s() transforms intermediate world coordinates to spectral coordinates.
*
* Given and returned:
*   spc       struct spcprm*
*                       Spectral transformation parameters.
*
* Given:
*   nx        int       Vector length.
*
*   sx        int       Vector stride.
*
*   sspec     int       Vector stride.
*
*   x         const double[]
*                       Intermediate world coordinates, in SI units.
*
* Returned:
*   spec      double[]  Spectral coordinates, in SI units.
*
*   stat      int[]     Status return value status for each vector element:
*                         0: Success.
*                         1: Invalid value of x.
*
* Function return value:
*             int       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().
*
*
* spcs2x() - Transform spectral coordinates
* -----------------------------------------
* spcs2x() transforms spectral world coordinates to intermediate world
* coordinates.
*
* Given and returned:
*   spc       struct spcprm*
*                       Spectral transformation parameters.
*
* Given:
*   nspec     int       Vector length.
*
*   sspec     int       Vector stride.
*
*   sx        int       Vector stride.
*
*   spec      const double[]
*                       Spectral coordinates, in SI units.
*
* Returned:
*   x         double[]  Intermediate world coordinates, in SI units.
*
*   stat      int[]     Status return value status for each vector element:
*                         0: Success.
*                         1: Invalid value of spec.
*
* Function return value:
*             int       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().
*
*
* spctype() - Spectral CTYPEia keyword analysis
* ---------------------------------------------
* spctype() checks whether a CTYPEia 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 CTYPEia 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.
*
* Given:
*   ctype     const char[9]
*                       The CTYPEia keyvalue, (eight characters with null
*                       termination).
*
* Returned:
*   stype     char[]    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.
*
*   scode     char[]    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.
*
*   sname     char[]    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.
*
*   units     char[]    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.
*
*   ptype     char*     Character code for the P-type spectral variable
*                       derived from ctype, one of 'F', 'W', 'A', or 'V'.
*
*   xtype     char*     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.
*
*   restreq   int*      Multivalued flag that indicates whether rest
*                       frequency or wavelength is required to compute
*                       spectral variables for this CTYPEia:
*                         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.
*
*   err       struct wcserr **
*                       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.
*
* Function return value:
*             int       Status return value:
*                         0: Success.
*                         2: Invalid spectral parameters (not a spectral
*                            CTYPEia).
*
*
* spcspxe() - Spectral keyword analysis
* ------------------------------------
* spcspxe() analyses the CTYPEia and CRVALia 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.
*
* Given:
*   ctypeS    const char[9]
*                       Spectral axis type, i.e. the CTYPEia 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 CTYPEia) may be set to '?' (it will not
*                       be reset).
*
*   crvalS    double    Value of the S-type spectral variable at the reference
*                       point, i.e. the CRVALia keyvalue, SI units.
*
*   restfrq,
*   restwav   double    Rest frequency [Hz] and rest wavelength in vacuo [m],
*                       only one of which need be given, the other should be
*                       set to zero.
*
* Returned:
*   ptype     char*     Character code for the P-type spectral variable
*                       derived from ctypeS, one of 'F', 'W', 'A', or 'V'.
*
*   xtype     char*     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.
*
*   restreq   int*      Multivalued flag that indicates whether rest frequency
*                       or wavelength is required to compute spectral
*                       variables for this CTYPEia, as for spctype().
*
*   crvalX    double*   Value of the X-type spectral variable at the reference
*                       point, SI units.
*
*   dXdS      double*   The derivative, dX/dS, evaluated at the reference
*                       point, SI units.  Multiply the CDELTia keyvalue by
*                       this to get the pixel spacing in the X-type spectral
*                       coordinate.
*
*   err       struct wcserr **
*                       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.
*
* Function return value:
```

```
 *              int       Status return value:
 *                          0: Success.
 *                          2: Invalid spectral parameters.
 *
 *
 * spcxpse() - Spectral keyword synthesis
 * ---------------------------------------
 * 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 CRVALia and also the derivative dS/dX which
 * may be used to compute CDELTia.  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.
 *
 * Given:
 *   ctypeS    const char[9]
 *                        The required spectral axis type, i.e. the CTYPEia
 *                        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 CTYPEia) may be set to '?' (it
 *                        will not be reset).
 *
 *   crvalX    double     Value of the X-type spectral variable at the reference
 *                        point (N.B. NOT the CRVALia keyvalue), SI units.
 *
 *   restfrq,
 *   restwav   double     Rest frequency [Hz] and rest wavelength in vacuo [m],
 *                        only one of which need be given, the other should be
 *                        set to zero.
 *
 * Returned:
 *   ptype     char*      Character code for the P-type spectral variable
 *                        derived from ctypeS, one of 'F', 'W', 'A', or 'V'.
 *
 *   xtype     char*      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.
 *
 *   restreq   int*       Multivalued flag that indicates whether rest frequency
 *                        or wavelength is required to compute spectral
 *                        variables for this CTYPEia, as for spctype().
 *
 *   crvalS    double*    Value of the S-type spectral variable at the reference
 *                        point (i.e. the appropriate CRVALia keyvalue), SI
 *                        units.
 *
 *   dSdX      double*    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 CDELTia
 *                        keyvalue.
 *
 *   err       struct wcserr **
 *                        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.
 *
 * Function return value:
 *              int       Status return value:
 *                          0: Success.
 *                          2: Invalid spectral parameters.
 *
 *
 * spctrne() - Spectral keyword translation
 * ----------------------------------------
 * 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.
 *
 * Given:
 *   ctypeS1   const char[9]
 *                        Spectral axis type, i.e. the CTYPEia 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 CTYPEia) may be set to '?' (it will not
 *                        be reset).
 *
 *   crvalS1   double     Value of the S-type spectral variable at the reference
 *                        point, i.e. the CRVALia keyvalue, SI units.
 *
```

```
*   cdeltS1   double    Increment of the S-type spectral variable at the
*                       reference point, SI units.
*
*   restfrq,
*   restwav   double    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'.
*
* Given and returned:
*   ctypeS2   char[9]   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).
*
* Returned:
*   crvalS2   double*   Value of the new S-type spectral variable at the
*                       reference point, i.e. the new CRVALia keyvalue, SI
*                       units.
*
*   cdeltS2   double*   Increment of the new S-type spectral variable at the
*                       reference point, i.e. the new CDELTia keyvalue, SI
*                       units.
*
*   err       struct wcserr **
*                       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.
*
* Function return value:
*             int       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.
*
*
* spcaips() - Translate AIPS-convention spectral keywords
* -------------------------------------------------------
* spcaips() translates AIPS-convention spectral CTYPEia and VELREF keyvalues.
*
* Given:
*   ctypeA    const char[9]
*                       CTYPEia keyvalue possibly containing an
*                       AIPS-convention spectral code (eight characters, need
*                       not be null-terminated).
*
*   velref    int       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 CTYPEia 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.
*
* Returned:
*   ctype     char[9]   Translated CTYPEia keyvalue, or a copy of ctypeA if no
*                       translation was performed (in which case any trailing
*                       blanks in ctypeA will be replaced with nulls).
*
*   specsys   char[9]   Doppler reference frame indicated by VELREF or else
*                       by CTYPEia 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.
*
* Function return value:
*             int       Status return value:
*                        -1: No translation required (not an error).
*                         0: Success.
*                         2: Invalid value of VELREF.
*
*
* spcprm struct - Spectral transformation parameters
* --------------------------------------------------
* 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.
*
*   int 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.
*
*   char type[8]
*     (Given) Four-letter spectral variable type, e.g "ZOPT" for
*     CTYPEia = 'ZOPT-F2W'.  (Declared as char[8] for alignment reasons.)
*
*   char code[4]
*     (Given) Three-letter spectral algorithm code, e.g "F2W" for
*     CTYPEia = 'ZOPT-F2W'.
*
*   double crval
*     (Given) Reference value (CRVALia), SI units.
*
*   double restfrq
*     (Given) The rest frequency [Hz], and ...
*
*   double 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.
*
*   double pv[7]
*     (Given) Grism parameters for 'GRI' and 'GRA' algorithm codes:
*       - 0: G, grating ruling density.
*       - 1: m, interference order.
*       - 2: alpha, angle of incidence [deg].
*       - 3: n_r, refractive index at the reference wavelength, lambda_r.
*       - 4: n'_r, dn/dlambda at the reference wavelength, lambda_r (/m).
*       - 5: epsilon, grating tilt angle [deg].
*       - 6: theta, detector tilt angle [deg].
*
* The remaining members of the spcprm struct are maintained by spcset() and
* must not be modified elsewhere:
*
*   double w[6]
```

```
*     (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.
*
*   int isGrism
*     (Returned) Grism coordinates?
*       - 0: no,
*       - 1: in vacuum,
*       - 2: in air.
*
*   int padding1
*     (An unused variable inserted for alignment purposes only.)
*
*   struct wcserr *err
*     (Returned) If enabled, when an error status is returned, this struct
*     contains detailed information about the error, see wcserr_enable().
*
*   void *padding2
*     (An unused variable inserted for alignment purposes only.)
*   int (*spxX2P)(SPX_ARGS)
*     (Returned) The first and ...
*   int (*spxP2S)(SPX_ARGS)
*     (Returned) ... the second of the pointers to the transformation
*     functions in the two-step algorithm chain X -> P -> 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.
*
*   int (*spxS2P)(SPX_ARGS)
*     (Returned) The first and ...
*   int (*spxP2X)(SPX_ARGS)
*     (Returned) ... the second of the pointers to the transformation
*     functions in the two-step algorithm chain S -> P -> 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.
*
*
* Global variable: const char *spc_errmsg[] - Status return messages
* -------------------------------------------------------------------
* Error messages to match the status value returned from each function.
*
*===========================================================================*/

#ifndef WCSLIB_SPC
#define WCSLIB_SPC

#include "spx.h"

#ifdef __cplusplus
extern "C" {
#endif


extern const char *spc_errmsg[];

enum spc_errmsg_enum {
  SPCERR_NO_CHANGE       = -1,  // No change.
  SPCERR_SUCCESS         =  0,    // Success.
  SPCERR_NULL_POINTER    =  1,       // Null spcprm pointer passed.
  SPCERR_BAD_SPEC_PARAMS =  2,    // Invalid spectral parameters.
  SPCERR_BAD_X           =  3,      // One or more of x coordinates were
                                // invalid.
  SPCERR_BAD_SPEC        =  4    // One or more of the spec coordinates were
                                // invalid.
};

struct spcprm {
  // Initialization flag (see the prologue above).
  //--------------------------------------------------------------------------
  int    flag;                    // Set to zero to force initialization.

  // Parameters to be provided (see the prologue above).
  //--------------------------------------------------------------------------
  char   type[8];           // Four-letter spectral variable type.
  char   code[4];           // Three-letter spectral algorithm code.

  double crval;                    // Reference value (CRVALia), SI units.
  double restfrq;                 // Rest frequency, Hz.
  double restwav;                 // Rest wavelength, m.

  double pv[7];                       // Grism parameters:
                                //   0: G, grating ruling density.
                                //   1: m, interference order.
                                //   2: alpha, angle of incidence.
                                //   3: n_r, refractive index at lambda_r.
```

```
                                //   4: n'_r, dn/dlambda at lambda_r.
                                //   5: epsilon, grating tilt angle.
                                //   6: theta, detector tilt angle.

  // Information derived from the parameters supplied.
  //----------------------------------------------------------------------------
  double w[6];                 // Intermediate values.
                                //   0: Rest frequency or wavelength (SI).
                                //   1: CRVALX (SI units).
                                //   2: CDELTX/CDELTia = dX/dS (SI units).
                                // The remainder are grism intermediates.

  int    isGrism;                // Grism coordinates?  1: vacuum, 2: air.
  int    padding1;              // (Dummy inserted for alignment purposes.)

  // Error handling
  //----------------------------------------------------------------------------
  struct wcserr *err;

  // Private
  //----------------------------------------------------------------------------
  void   *padding2;             // (Dummy inserted for alignment purposes.)
  int (*spxX2P)(SPX_ARGS);        // Pointers to the transformation functions
  int (*spxP2S)(SPX_ARGS);        // in the two-step algorithm chain in the
                                // pixel-to-spectral direction.

  int (*spxS2P)(SPX_ARGS);        // Pointers to the transformation functions
  int (*spxP2X)(SPX_ARGS);        // in the two-step algorithm chain in the
                                // spectral-to-pixel direction.
};

// Size of the spcprm struct in int units, used by the Fortran wrappers.
#define SPCLEN (sizeof(struct spcprm)/sizeof(int))


int spcini(struct spcprm *spc);

int spcfree(struct spcprm *spc);

int spcsize(const struct spcprm *spc, int sizes[2]);

int spcprt(const struct spcprm *spc);

int spcperr(const struct spcprm *spc, const char *prefix);

int spcset(struct spcprm *spc);

int spcx2s(struct spcprm *spc, int nx, int sx, int sspec,
           const double x[], double spec[], int stat[]);

int spcs2x(struct spcprm *spc, int nspec, int sspec, int sx,
           const double spec[], double x[], int stat[]);

int spctype(const char ctype[9], char stype[], char scode[], char sname[],
            char units[], char *ptype, char *xtype, int *restreq,
            struct wcserr **err);

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);

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);

int spctrne(const char ctypeS1[9], double crvalS1, double cdeltS1,
            double restfrq, double restwav, char ctypeS2[9], double *crvalS2,
            double *cdeltS2, struct wcserr **err);

int spcaips(const char ctypeA[9], int velref, char ctype[9], char specsys[9]);


// Deprecated.
#define spcini_errmsg spc_errmsg
#define spcprt_errmsg spc_errmsg
#define spcset_errmsg spc_errmsg
#define spcx2s_errmsg spc_errmsg
#define spcs2x_errmsg spc_errmsg

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 ctypeS2[9], double *crvalS2,
           double *cdeltS2);

#ifdef __cplusplus
}
#endif

#endif // WCSLIB_SPC
```

6.17 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.

6.17.1 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.

6.17.2 Function Documentation

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.

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.

Parameters

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.

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 $(\alpha_0, +90^\circ - \epsilon)$ or $(\alpha_0, -90^\circ + \epsilon)$, 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.

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 $(\alpha_0, +90^\circ - \epsilon)$ or $(\alpha_0, -90^\circ + \epsilon)$, 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.

6.18 sph.h

Go to the documentation of this file.

```
/*============================================================================
  WCSLIB 8.1 - an implementation of the FITS WCS standard.
  Copyright (C) 1995-2023, Mark Calabretta

  This file is part of WCSLIB.

  WCSLIB is free software: you can redistribute it and/or modify it under the
  terms of the GNU Lesser General Public License as published by the Free
  Software Foundation, either version 3 of the License, or (at your option)
  any later version.

  WCSLIB is distributed in the hope that it will be useful, but WITHOUT ANY
  WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
  FOR A PARTICULAR PURPOSE.  See the GNU Lesser General Public License for
  more details.

  You should have received a copy of the GNU Lesser General Public License
  along with WCSLIB.  If not, see http://www.gnu.org/licenses.

  Author: Mark Calabretta, Australia Telescope National Facility, CSIRO.
  http://www.atnf.csiro.au/people/Mark.Calabretta
  $Id: sph.h,v 8.1 2023/07/05 17:12:07 mcalabre Exp $
*=============================================================================
*
* WCSLIB 8.1 - C routines that implement the FITS World Coordinate System
* (WCS) standard.  Refer to the README file provided with WCSLIB for an
* overview of the library.
*
*
* Summary of the sph routines
* ---------------------------
* 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.
*
*
* sphx2s() - Rotation in the pixel-to-world direction
* ---------------------------------------------------
* sphx2s() transforms native coordinates of a projection to celestial
* coordinates.
*
* Given:
*   eul       const double[5]
*                       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])
*
*   nphi,
*   ntheta    int       Vector lengths.
*
*   spt,sxy   int       Vector strides.
*
*   phi,theta const double[]
*                       Longitude and latitude in the native coordinate
*                       system of the projection [deg].
*
* Returned:
*   lng,lat   double[]  Celestial longitude and latitude [deg].  These may
```

```
*                       refer to the same storage as phi and theta
*                       respectively.
*
* Function return value:
*             int       Status return value:
*                         0: Success.
*
*
* sphs2x() - Rotation in the world-to-pixel direction
* ---------------------------------------------------
* sphs2x() transforms celestial coordinates to the native coordinates of a
* projection.
*
* Given:
*   eul       const double[5]
*                       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])
*
*   nlng,nlat int       Vector lengths.
*
*   sll,spt   int       Vector strides.
*
*   lng,lat   const double[]
*                       Celestial longitude and latitude [deg].
*
* Returned:
*   phi,theta double[]  Longitude and latitude in the native coordinate system
*                       of the projection [deg].  These may refer to the same
*                       storage as lng and lat respectively.
*
* Function return value:
*             int       Status return value:
*                         0: Success.
*
*
* sphdpa() - Compute angular distance and position angle
* ------------------------------------------------------
* 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().
*
* Given:
*   nfield    int       The number of field points.
*
*   lng0,lat0 double    Spherical coordinates of the reference point [deg].
*
*   lng,lat   const double[]
*                       Spherical coordinates of the field points [deg].
*
* Returned:
*   dist,pa   double[]  Angular distances and position angles [deg].  These
*                       may refer to the same storage as lng and lat
*                       respectively.
*
* Function return value:
*             int       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 (lng0,+90-epsilon) or
*      (lng0,-90+epsilon), in the limit as epsilon 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 degrees).
*
*      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 degrees (within rounding tolerance).  sphdpa()
*      will return an arbitrary position angle in such circumstances.
*
*
* sphpad() - Compute field points offset from a given point
* ---------------------------------------------------------
* 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().
*
* Given:
*   nfield    int       The number of field points.
*
*   lng0,lat0 double    Spherical coordinates of the reference point [deg].
*
*   dist,pa   const double[]
*                       Angular distances and position angles [deg].
*
* Returned:
*   lng,lat   double[]  Spherical coordinates of the field points [deg].
*                       These may refer to the same storage as dist and pa
*                       respectively.
*
* Function return value:
*             int       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 (lng0,+90-epsilon) or (lng0,-90+epsilon), in
*      the limit as epsilon goes to zero.
*
*   Applying sphpad() with the distances and position angles computed by
*   sphdpa() should return the original field points.
*
*===========================================================================*/

#ifndef WCSLIB_SPH
#define WCSLIB_SPH

#ifdef __cplusplus
extern "C" {
#endif


int sphx2s(const double eul[5], int nphi, int ntheta, int spt, int sxy,
           const double phi[], const double theta[],
           double lng[], double lat[]);

int sphs2x(const double eul[5], int nlng, int nlat, int sll , int spt,
           const double lng[], const double lat[],
           double phi[], double theta[]);

int sphdpa(int nfield, double lng0, double lat0,
           const double lng[], const double lat[],
           double dist[], double pa[]);

int sphpad(int nfield, double lng0, double lat0,
           const double dist[], const double pa[],
           double lng[], double lat[]);


#ifdef __cplusplus
}
#endif

#endif // WCSLIB_SPH
```

6.19 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 [ ]

6.19.1 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.

6.19.2 Macro Definition Documentation

SPXLEN

```
#define SPXLEN (sizeof(struct spxprm)/sizeof(int))
```

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

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.

6.19.3 Enumeration Type Documentation

spx_errmsg

```
enum spx_errmsg
```

Enumerator

SPXERR_SUCCESS	
SPXERR_NULL_POINTER	
SPXERR_BAD_SPEC_PARAMS	
SPXERR_BAD_SPEC_VAR	
SPXERR_BAD_INSPEC_COORD	

6.19.4 Function Documentation

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.

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.

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.

afrqfreq()

```
int afrqfreq (
            SPX_ARGS )
```

afrqfreq() converts angular frequency to frequency.

See freqafrq() for a description of the API.

freqener()

```
int freqener (
            SPX_ARGS )
```

freqener() converts frequency to photon energy.

See freqafrq() for a description of the API.

enerfreq()

```
int enerfreq (
            SPX_ARGS )
```

enerfreq() converts photon energy to frequency.

See freqafrq() for a description of the API.

freqwavn()

```
int freqwavn (
            SPX_ARGS )
```

freqwavn() converts frequency to wave number.

See freqafrq() for a description of the API.

wavnfreq()

```
int wavnfreq (
            SPX_ARGS )
```

wavnfreq() converts wave number to frequency.

See freqafrq() for a description of the API.

freqwave()

```
int freqwave (
            SPX_ARGS )
```

freqwave() converts frequency to vacuum wavelength.

See freqafrq() for a description of the API.

wavefreq()

```
int wavefreq (
            SPX_ARGS )
```

wavefreq() converts vacuum wavelength to frequency.

See freqafrq() for a description of the API.

freqawav()

```
int freqawav (
            SPX_ARGS )
```

freqawav() converts frequency to air wavelength.

See freqafrq() for a description of the API.

awavfreq()

```
int awavfreq (
            SPX_ARGS )
```

awavfreq() converts air wavelength to frequency.

See freqafrq() for a description of the API.

waveawav()

```
int waveawav (
            SPX_ARGS )
```

waveawav() converts vacuum wavelength to air wavelength.

See freqafrq() for a description of the API.

awavwave()

```
int awavwave (
            SPX_ARGS )
```

awavwave() converts air wavelength to vacuum wavelength.

See freqafrq() for a description of the API.

velobeta()

```
int velobeta (
            SPX_ARGS )
```

velobeta() converts relativistic velocity to relativistic beta.

See freqafrq() for a description of the API.

betavelo()

```
int betavelo (
            SPX_ARGS )
```

betavelo() converts relativistic beta to relativistic velocity.

See freqafrq() for a description of the API.

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.

velofreq()

```
int velofreq (
            SPX_ARGS )
```

velofreq() converts relativistic velocity to frequency.

See freqvelo() for a description of the API.

freqvrad()

```
int freqvrad (
            SPX_ARGS  )
```

freqvrad() converts frequency to radio velocity.

See freqvelo() for a description of the API.

vradfreq()

```
int vradfreq (
            SPX_ARGS  )
```

vradfreq() converts radio velocity to frequency.

See freqvelo() for a description of the API.

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.

velowave()

```
int velowave (
            SPX_ARGS  )
```

velowave() converts relativistic velocity to vacuum wavelength.

See freqvelo() for a description of the API.

awavvelo()

```
int awavvelo (
              SPX_ARGS   )
```

awavvelo() converts air wavelength to relativistic velocity.

See freqvelo() for a description of the API.

veloawav()

```
int veloawav (
              SPX_ARGS   )
```

veloawav() converts relativistic velocity to air wavelength.

See freqvelo() for a description of the API.

wavevopt()

```
int wavevopt (
              SPX_ARGS   )
```

wavevopt() converts vacuum wavelength to optical velocity.

See freqvelo() for a description of the API.

voptwave()

```
int voptwave (
              SPX_ARGS   )
```

voptwave() converts optical velocity to vacuum wavelength.

See freqvelo() for a description of the API.

wavezopt()

```
int wavezopt (
              SPX_ARGS   )
```

wavevopt() converts vacuum wavelength to redshift.

See freqvelo() for a description of the API.

zoptwave()

```
int zoptwave (
            SPX_ARGS )
```

zoptwave() converts redshift to vacuum wavelength.

See freqvelo() for a description of the API.

6.19.5 Variable Documentation

spx_errmsg

```
const char* spx_errmsg[]  [extern]
```

6.20 spx.h

Go to the documentation of this file.

```
/*============================================================================
  WCSLIB 8.1 - an implementation of the FITS WCS standard.
  Copyright (C) 1995-2023, Mark Calabretta

  This file is part of WCSLIB.

  WCSLIB is free software: you can redistribute it and/or modify it under the
  terms of the GNU Lesser General Public License as published by the Free
  Software Foundation, either version 3 of the License, or (at your option)
  any later version.

  WCSLIB is distributed in the hope that it will be useful, but WITHOUT ANY
  WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
  FOR A PARTICULAR PURPOSE.  See the GNU Lesser General Public License for
  more details.

  You should have received a copy of the GNU Lesser General Public License
  along with WCSLIB.  If not, see http://www.gnu.org/licenses.

  Author: Mark Calabretta, Australia Telescope National Facility, CSIRO.
  http://www.atnf.csiro.au/people/Mark.Calabretta
  $Id: spx.h,v 8.1 2023/07/05 17:12:07 mcalabre Exp $
*=============================================================================
*
* WCSLIB 8.1 - C routines that implement the FITS World Coordinate System
* (WCS) standard.  Refer to the README file provided with WCSLIB for an
* overview of the library.
*
*
* Summary of the spx routines
* ---------------------------
* 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 (= v/c)
*   - betavelo()    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.
*
*
* specx() - Spectral cross conversions (scalar)
* ---------------------------------------------
* Given one spectral variable specx() computes all the others, plus the
* required derivatives of each with respect to the others.
*
* Given:
*   type      const char*
*                       The type of spectral variable given by spec, FREQ,
*                       AFRQ, ENER, WAVN, VRAD, WAVE, VOPT, ZOPT, AWAV, VELO,
*                       or BETA (case sensitive).
*
*   spec      double    The spectral variable given, in SI units.
*
*   restfrq,
*   restwav   double    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.
*
* Given and returned:
*   specs     struct spxprm*
*                       Data structure containing all spectral variables and
*                       their derivatives, in SI units.
*
* Function return value:
*             int       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.
*
*
* spxperr() - Print error messages from a spxprm struct
* -----------------------------------------------------
* 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.
*
* Given:
*   spx       const struct spxprm*
*                       Spectral variables and their derivatives.
*
*   prefix    const char *
*                       If non-NULL, each output line will be prefixed with
*                       this string.
*
* Function return value:
*             int       Status return value:
*                         0: Success.
*                         1: Null spxprm pointer passed.
*
*
* freqafrq() - Convert frequency to angular frequency (vector)
* ------------------------------------------------------------
* freqafrq() converts frequency to angular frequency.
*
* Given:
*   param     double    Ignored.
*
*   nspec     int       Vector length.
*
*   instep,
*   outstep   int       Vector strides.
*
*   inspec    const double[]
*                       Input spectral variables, in SI units.
*
* Returned:
*   outspec   double[]  Output spectral variables, in SI units.
```

```
*
*   stat      int[]     Status return value for each vector element:
*                         0: Success.
*                         1: Invalid value of inspec.
*
* Function return value:
*             int       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.
*
*
* freqvelo(), velofreq(), freqvrad(), and vradfreq() implement vector
* conversions between frequency and velocity spectral types.  They all have
* the same API.
*
*
* freqvelo() - Convert frequency to relativistic velocity (vector)
* ----------------------------------------------------------------
* freqvelo() converts frequency to relativistic velocity.
*
* Given:
*   param     double    Rest frequency [Hz].
*
*   nspec     int       Vector length.
*
*   instep,
*   outstep   int       Vector strides.
*
*   inspec    const double[]
*                       Input spectral variables, in SI units.
*
* Returned:
*   outspec   double[]  Output spectral variables, in SI units.
*
*   stat      int[]     Status return value for each vector element:
*                         0: Success.
*                         1: Invalid value of inspec.
*
* Function return value:
*             int       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.
*
*
* wavevelo(), velowave(), awavvelo(), veloawav(), wavevopt(), voptwave(),
* wavezopt(), and zoptwave() implement vector conversions between wavelength
* and velocity spectral types.  They all have the same API.
*
*
* wavevelo() - Conversions between wavelength and velocity types (vector)
* -----------------------------------------------------------------------
* wavevelo() converts vacuum wavelength to relativistic velocity.
*
* Given:
*   param     double    Rest wavelength in vacuo [m].
*
*   nspec     int       Vector length.
*
*   instep,
*   outstep   int       Vector strides.
*
*   inspec    const double[]
*                       Input spectral variables, in SI units.
*
* Returned:
*   outspec   double[]  Output spectral variables, in SI units.
*
*   stat      int[]     Status return value for each vector element:
*                         0: Success.
*                         1: Invalid value of inspec.
*
* Function return value:
*             int       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.
*
*
* spxprm struct - Spectral variables and their derivatives
* --------------------------------------------------------
* 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.
*
*   double restfrq
*     (Returned) Rest frequency [Hz].
*
*   double restwav
*     (Returned) Rest wavelength [m].
*
*   int wavetype
*     (Returned) True if wave types have been computed, and ...
*
*   int 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.
*
*   double freq
*     (Returned) Frequency [Hz] (wavetype).
*
*   double afrq
*     (Returned) Angular frequency [rad/s] (wavetype).
*
*   double ener
*     (Returned) Photon energy [J] (wavetype).
*
*   double wavn
*     (Returned) Wave number [/m] (wavetype).
*
*   double vrad
*     (Returned) Radio velocity [m/s] (velotype).
*
*   double wave
*     (Returned) Vacuum wavelength [m] (wavetype).
*
*   double vopt
*     (Returned) Optical velocity [m/s] (velotype).
*
*   double zopt
*     (Returned) Redshift [dimensionless] (velotype).
*
*   double awav
*     (Returned) Air wavelength [m] (wavetype).
*
*   double velo
*     (Returned) Relativistic velocity [m/s] (velotype).
*
*   double beta
*     (Returned) Relativistic beta [dimensionless] (velotype).
*
*   double dfreqafrq
*     (Returned) Derivative of frequency with respect to angular frequency
*     [/rad] (constant, = 1 / 2*pi), and ...
*   double dafrqfreq
*     (Returned) ... vice versa [rad] (constant, = 2*pi, always available).
*
*   double dfreqener
*     (Returned) Derivative of frequency with respect to photon energy
*     [/J/s] (constant, = 1/h), and ...
*   double denerfreq
*     (Returned) ... vice versa [Js] (constant, = h, Planck's constant,
*     always available).
*
*   double dfreqwavn
*     (Returned) Derivative of frequency with respect to wave number [m/s]
*     (constant, = c, the speed of light in vacuo), and ...
*   double dwavnfreq
*     (Returned) ... vice versa [s/m] (constant, = 1/c, always available).
*
*   double dfreqvrad
*     (Returned) Derivative of frequency with respect to radio velocity [/m],
*     and ...
*   double dvradfreq
*     (Returned) ... vice versa [m] (wavetype && velotype).
*
*   double dfreqwave
*     (Returned) Derivative of frequency with respect to vacuum wavelength
*     [/m/s], and ...
*   double dwavefreq
```

```
*     (Returned) ... vice versa [m s] (wavetype).
*
*   double dfreqawav
*     (Returned) Derivative of frequency with respect to air wavelength,
*     [/m/s], and ...
*   double dawavfreq
*     (Returned) ... vice versa [m s] (wavetype).
*
*   double dfreqvelo
*     (Returned) Derivative of frequency with respect to relativistic
*     velocity [/m], and ...
*   double dvelofreq
*     (Returned) ... vice versa [m] (wavetype && velotype).
*
*   double dwavevopt
*     (Returned) Derivative of vacuum wavelength with respect to optical
*     velocity [s], and ...
*   double dvoptwave
*     (Returned) ... vice versa [/s] (wavetype && velotype).
*
*   double dwavezopt
*     (Returned) Derivative of vacuum wavelength with respect to redshift [m],
*     and ...
*   double dzoptwave
*     (Returned) ... vice versa [/m] (wavetype && velotype).
*
*   double dwaveawav
*     (Returned) Derivative of vacuum wavelength with respect to air
*     wavelength [dimensionless], and ...
*   double dawavwave
*     (Returned) ... vice versa [dimensionless] (wavetype).
*
*   double dwavevelo
*     (Returned) Derivative of vacuum wavelength with respect to relativistic
*     velocity [s], and ...
*   double dvelowave
*     (Returned) ... vice versa [/s] (wavetype && velotype).
*
*   double dawavvelo
*     (Returned) Derivative of air wavelength with respect to relativistic
*     velocity [s], and ...
*   double dveloawav
*     (Returned) ... vice versa [/s] (wavetype && velotype).
*
*   double dvelobeta
*     (Returned) Derivative of relativistic velocity with respect to
*     relativistic beta [m/s] (constant, = c, the speed of light in vacuo),
*     and ...
*   double dbetavelo
*     (Returned) ... vice versa [s/m] (constant, = 1/c, always available).
*
*   struct wcserr *err
*     (Returned) If enabled, when an error status is returned, this struct
*     contains detailed information about the error, see wcserr_enable().
*
*   void *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.
*
*===========================================================================*/

#ifndef WCSLIB_SPEC
#define WCSLIB_SPEC

#ifdef __cplusplus
extern "C" {
#endif

extern const char *spx_errmsg[];

enum spx_errmsg {
  SPXERR_SUCCESS          = 0,    // Success.
  SPXERR_NULL_POINTER     = 1,       // Null spxprm pointer passed.
  SPXERR_BAD_SPEC_PARAMS  = 2,    // Invalid spectral parameters.
  SPXERR_BAD_SPEC_VAR     = 3,       // Invalid spectral variable.
  SPXERR_BAD_INSPEC_COORD = 4    // One or more of the inspec coordinates were
                                // invalid.
};

struct spxprm {
  double restfrq, restwav;        // Rest frequency [Hz] and wavelength [m].

  int wavetype, velotype;       // True if wave/velocity types have been
                                // computed; types are defined below.
```

```
  // Spectral variables computed by specx().
  //------------------------------------------------------------------------
  double freq,                  // wavetype: Frequency [Hz].
         afrq,                  // wavetype: Angular frequency [rad/s].
         ener,                  // wavetype: Photon energy [J].
         wavn,                  // wavetype: Wave number [/m].
         vrad,                  // velotype: Radio velocity [m/s].
         wave,                  // wavetype: Vacuum wavelength [m].
         vopt,                  // velotype: Optical velocity [m/s].
         zopt,                  // velotype: Redshift.
         awav,                  // wavetype: Air wavelength [m].
         velo,                  // velotype: Relativistic velocity [m/s].
         beta;                  // velotype: Relativistic beta.

  // Derivatives of spectral variables computed by specx().
  //------------------------------------------------------------------------
  double dfreqafrq, dafrqfreq,    // Constant, always available.
         dfreqener, denerfreq,    // Constant, always available.
         dfreqwavn, dwavnfreq,    // Constant, always available.
         dfreqvrad, dvradfreq,    // wavetype && velotype.
         dfreqwave, dwavefreq,    // wavetype.
         dfreqawav, dawavfreq,    // wavetype.
         dfreqvelo, dvelofreq,    // wavetype && velotype.
         dwavevopt, dvoptwave,    // wavetype && velotype.
         dwavezopt, dzoptwave,    // wavetype && velotype.
         dwaveawav, dawavwave,    // wavetype.
         dwavevelo, dvelowave,    // wavetype && velotype.
         dawavvelo, dveloawav,    // wavetype && velotype.
         dvelobeta, dbetavelo;    // Constant, always available.

  // Error handling
  //------------------------------------------------------------------------
  struct wcserr *err;

  // Private
  //------------------------------------------------------------------------
  void   *padding;              // (Dummy inserted for alignment purposes.)
};

// Size of the spxprm struct in int units, used by the Fortran wrappers.
#define SPXLEN (sizeof(struct spxprm)/sizeof(int))


int specx(const char *type, double spec, double restfrq, double restwav,
          struct spxprm *specs);

int spxperr(const struct spxprm *spx, const char *prefix);

// For use in declaring function prototypes, e.g. in spcprm.
#define SPX_ARGS double param, int nspec, int instep, int outstep, \
    const double inspec[], double outspec[], int stat[]

int freqafrq(SPX_ARGS);
int afrqfreq(SPX_ARGS);

int freqener(SPX_ARGS);
int enerfreq(SPX_ARGS);

int freqwavn(SPX_ARGS);
int wavnfreq(SPX_ARGS);

int freqwave(SPX_ARGS);
int wavefreq(SPX_ARGS);

int freqawav(SPX_ARGS);
int awavfreq(SPX_ARGS);

int waveawav(SPX_ARGS);
int awavwave(SPX_ARGS);

int velobeta(SPX_ARGS);
int betavelo(SPX_ARGS);


int freqvelo(SPX_ARGS);
int velofreq(SPX_ARGS);

int freqvrad(SPX_ARGS);
int vradfreq(SPX_ARGS);


int wavevelo(SPX_ARGS);
int velowave(SPX_ARGS);

int awavvelo(SPX_ARGS);
int veloawav(SPX_ARGS);
```

```
int wavevopt(SPX_ARGS);
int voptwave(SPX_ARGS);

int wavezopt(SPX_ARGS);
int zoptwave(SPX_ARGS);


#ifdef __cplusplus
}
#endif

#endif // WCSLIB_SPEC
```

6.21 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.

6.21.1 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.

6.21.2 Macro Definition Documentation

TABLEN

```
#define TABLEN (sizeof(struct tabprm)/sizeof(int))
```

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

tabini_errmsg

```
#define tabini_errmsg tab_errmsg
```

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

tabcpy_errmsg

```
#define tabcpy_errmsg tab_errmsg
```

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

tabfree_errmsg

```
#define tabfree_errmsg tab_errmsg
```

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

tabprt_errmsg

```
#define tabprt_errmsg tab_errmsg
```

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

tabset_errmsg

```
#define tabset_errmsg tab_errmsg
```

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

tabx2s_errmsg

```
#define tabx2s_errmsg tab_errmsg
```

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

tabs2x_errmsg

```
#define tabs2x_errmsg tab_errmsg
```

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

6.21.3 Enumeration Type Documentation

tab_errmsg_enum

```
enum tab_errmsg_enum
```

Enumerator

TABERR_SUCCESS	
TABERR_NULL_POINTER	
TABERR_MEMORY	
TABERR_BAD_PARAMS	
TABERR_BAD_X	
TABERR_BAD_WORLD	

6.21.4 Function Documentation

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().

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().

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().

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.

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.

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.

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.

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.

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().

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().

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().

6.21.5 Variable Documentation

tab_errmsg

```
const char * tab_errmsg[ ]  [extern]
```

Error messages to match the status value returned from each function.

6.22 tab.h

Go to the documentation of this file.

```
/*============================================================================
  WCSLIB 8.1 - an implementation of the FITS WCS standard.
  Copyright (C) 1995-2023, Mark Calabretta

  This file is part of WCSLIB.

  WCSLIB is free software: you can redistribute it and/or modify it under the
  terms of the GNU Lesser General Public License as published by the Free
```

```
  Software Foundation, either version 3 of the License, or (at your option)
  any later version.

  WCSLIB is distributed in the hope that it will be useful, but WITHOUT ANY
  WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
  FOR A PARTICULAR PURPOSE.  See the GNU Lesser General Public License for
  more details.

  You should have received a copy of the GNU Lesser General Public License
  along with WCSLIB.  If not, see http://www.gnu.org/licenses.

  Author: Mark Calabretta, Australia Telescope National Facility, CSIRO.
  http://www.atnf.csiro.au/people/Mark.Calabretta
  $Id: tab.h,v 8.1 2023/07/05 17:12:07 mcalabre Exp $
*=============================================================================
*
* WCSLIB 8.1 - C routines that implement the FITS World Coordinate System
* (WCS) standard.  Refer to the README file provided with WCSLIB for an
* overview of the library.
*
*
* Summary of the tab routines
* ---------------------------
* 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.
*
*
* tabini() - Default constructor for the tabprm struct
* ----------------------------------------------------
* 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.
*
* Given:
*   alloc     int       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.)
*
*   M         int       The number of tabular coordinate axes.
*
```

```
*   K         const int[]
*                       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().
*
* Given and returned:
*   tab       struct tabprm*
*                       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).
*
* Function return value:
*             int       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().
*
*
* tabmem() - Acquire tabular memory
* ---------------------------------
* tabmem() takes control of memory allocated by the user for arrays in the
* tabprm struct.
*
* Given and returned:
*   tab       struct tabprm*
*                       Tabular transformation parameters.
*
* Function return value:
*             int       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().
*
*
* tabcpy() - Copy routine for the tabprm struct
* ---------------------------------------------
* 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.
*
* Given:
*   alloc     int       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.)
*
*   tabsrc    const struct tabprm*
*                       Struct to copy from.
*
* Given and returned:
*   tabdst    struct tabprm*
*                       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).
*
* Function return value:
*             int       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().
 *
 *
 * tabcmp() - Compare two tabprm structs for equality
 * --------------------------------------------------
 * tabcmp() compares two tabprm structs for equality.
 *
 * Given:
 *   cmp       int        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.
 *
 *   tol       double     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.
 *
 *   tab1      const struct tabprm*
 *                        The first tabprm struct to compare.
 *
 *   tab2      const struct tabprm*
 *                        The second tabprm struct to compare.
 *
 * Returned:
 *   equal     int*       Non-zero when the given structs are equal.
 *
 * Function return value:
 *             int        Status return value:
 *                          0: Success.
 *                          1: Null pointer passed.
 *
 *
 * tabfree() - Destructor for the tabprm struct
 * --------------------------------------------
 * 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().
 *
 * Returned:
 *   tab       struct tabprm*
 *                        Coordinate transformation parameters.
 *
 * Function return value:
 *             int        Status return value:
 *                          0: Success.
 *                          1: Null tabprm pointer passed.
 *
 *
 * tabsize() - Compute the size of a tabprm struct
 * -----------------------------------------------
 * tabsize() computes the full size of a tabprm struct, including allocated
 * memory.
 *
 * Given:
 *   tab       const struct tabprm*
 *                        Tabular transformation parameters.
 *
 *                        If NULL, the base size of the struct and the allocated
 *                        size are both set to zero.
 *
 * Returned:
 *   sizes     int[2]     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.
 *
 * Function return value:
 *             int        Status return value:
 *                          0: Success.
 *
 *
 * tabprt() - Print routine for the tabprm struct
 * ----------------------------------------------
 * tabprt() prints the contents of a tabprm struct using wcsprintf().  Mainly
 * intended for diagnostic purposes.
 *
```

```
* Given:
*   tab       const struct tabprm*
*                       Tabular transformation parameters.
*
* Function return value:
*             int       Status return value:
*                         0: Success.
*                         1: Null tabprm pointer passed.
*
*
* tabperr() - Print error messages from a tabprm struct
* -----------------------------------------------------
* 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.
*
* Given:
*   tab       const struct tabprm*
*                       Tabular transformation parameters.
*
*   prefix    const char *
*                       If non-NULL, each output line will be prefixed with
*                       this string.
*
* Function return value:
*             int       Status return value:
*                         0: Success.
*                         1: Null tabprm pointer passed.
*
*
* tabset() - Setup routine for the tabprm struct
* ----------------------------------------------
* 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.
*
* Given and returned:
*   tab       struct tabprm*
*                       Tabular transformation parameters.
*
* Function return value:
*             int       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().
*
*
* tabx2s() - Pixel-to-world transformation
* ----------------------------------------
* tabx2s() transforms intermediate world coordinates to world coordinates
* using coordinate lookup.
*
* Given and returned:
*   tab       struct tabprm*
*                       Tabular transformation parameters.
*
* Given:
*   ncoord,
*   nelem     int       The number of coordinates, each of vector length
*                       nelem.
*
*   x         const double[ncoord][nelem]
*                       Array of intermediate world coordinates, SI units.
*
* Returned:
*   world     double[ncoord][nelem]
*                       Array of world coordinates, in SI units.
*
*   stat      int[ncoord]
*                       Status return value status for each coordinate:
*                         0: Success.
*                         1: Invalid intermediate world coordinate.
*
* Function return value:
*             int       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().
*
*
* tabs2x() - World-to-pixel transformation
* ----------------------------------------
* tabs2x() transforms world coordinates to intermediate world coordinates.
*
* Given and returned:
*   tab       struct tabprm*
*                      Tabular transformation parameters.
*
* Given:
*   ncoord,
*   nelem     int       The number of coordinates, each of vector length
*                      nelem.
*   world     const double[ncoord][nelem]
*                      Array of world coordinates, in SI units.
*
* Returned:
*   x         double[ncoord][nelem]
*                      Array of intermediate world coordinates, SI units.
*   stat      int[ncoord]
*                      Status return value status for each vector element:
*                        0: Success.
*                        1: Invalid world coordinate.
*
* Function return value:
*             int       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().
*
*
* tabprm struct - Tabular transformation parameters
* -------------------------------------------------
* 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.
*
*   int 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.
*
*   int 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.
*
*   int *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.
*
*   int *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 <= m <= 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 <= i <= N).  In terms of the FITS
*     keywords
*
*       map[PVi_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.
*
*   double *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.
*
*   double **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;
*
*   double *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.
*
*   int nc
*     (Returned) Total number of coordinate vectors in the coordinate array
*     being the product K_1 * K_2 * ... * K_M (see tabprm::K).
*
*   int padding
*     (An unused variable inserted for alignment purposes only.)
*
*   int *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).
*
*   int *p0
*     (Returned) Pointer to the first element of a vector of length tabprm::M
*     of interpolated indices into the coordinate array such that Upsilon_m,
*     as defined in Paper III, is equal to (p0[m] + 1) + tabprm::delta[m].
*
*   double *delta
*     (Returned) Pointer to the first element of a vector of length tabprm::M
*     of interpolated indices into the coordinate array such that Upsilon_m,
*     as defined in Paper III, is equal to (tabprm::p0[m] + 1) + delta[m].
*
*   double *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.
*
*   struct wcserr *err
*     (Returned) If enabled, when an error status is returned, this struct
*     contains detailed information about the error, see wcserr_enable().
*
*   int m_flag
*     (For internal use only.)
*   int m_M
*     (For internal use only.)
*   int m_N
*     (For internal use only.)
*   int set_M
```

```
*     (For internal use only.)
*   int m_K
*     (For internal use only.)
*   int m_map
*     (For internal use only.)
*   int m_crval
*     (For internal use only.)
*   int m_index
*     (For internal use only.)
*   int m_indxs
*     (For internal use only.)
*   int m_coord
*     (For internal use only.)
*
*
* Global variable: const char *tab_errmsg[] - Status return messages
* ------------------------------------------------------------------
* Error messages to match the status value returned from each function.
*
*===========================================================================*/

#ifndef WCSLIB_TAB
#define WCSLIB_TAB

#ifdef __cplusplus
extern "C" {
#endif


extern const char *tab_errmsg[];

enum tab_errmsg_enum {
  TABERR_SUCCESS      = 0,        // Success.
  TABERR_NULL_POINTER = 1,   // Null tabprm pointer passed.
  TABERR_MEMORY       = 2, // Memory allocation failed.
  TABERR_BAD_PARAMS   = 3,     // Invalid tabular parameters.
  TABERR_BAD_X        = 4,  // One or more of the x coordinates were
                              // invalid.
  TABERR_BAD_WORLD    = 5       // One or more of the world coordinates were
                              // invalid.
};

struct tabprm {
  // Initialization flag (see the prologue above).
  //-------------------------------------------------------------------------
  int    flag;                      // Set to zero to force initialization.

  // Parameters to be provided (see the prologue above).
  //-------------------------------------------------------------------------
  int    M;                      // Number of tabular coordinate axes.
  int    *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.
  int    *map;                        // Vector of length M usually such that
                                  // map[m-1] == i-1 for coordinate array
                                  // axis m and image axis i (see above).
  double *crval;             // Vector of length M containing the index
                                  // value for the reference pixel for each
                                  // of the tabular coordinate axes.
  double **index;            // Vector of pointers to M indexing vectors
                                  // of lengths (K_1, K_2,... K_M).
  double *coord;             // (1+M)-dimensional tabular coordinate
                                  // array (see above).

  // Information derived from the parameters supplied.
  //-------------------------------------------------------------------------
  int    nc;                    // Number of coordinate vectors (of length
                                  // M) in the coordinate array.
  int    padding;                  // (Dummy inserted for alignment purposes.)
  int    *sense;             // Vector of M flags that indicate whether
                                  // the Mth indexing vector is monotonic
                                  // increasing, or else decreasing.
  int    *p0;                   // Vector of M indices.
  double *delta;             // Vector of M increments.
  double *extrema;                 // (1+M)-dimensional array of coordinate
                                  // extrema.

  // Error handling
  //-------------------------------------------------------------------------
  struct wcserr *err;

  // Private - the remainder are for memory management.
  //-------------------------------------------------------------------------
  int    m_flag, m_M, m_N;
  int    set_M;
  int    *m_K, *m_map;
```

```
  double *m_crval, **m_index, **m_indxs, *m_coord;
};

// Size of the tabprm struct in int units, used by the Fortran wrappers.
#define TABLEN (sizeof(struct tabprm)/sizeof(int))


int tabini(int alloc, int M, const int K[], struct tabprm *tab);

int tabmem(struct tabprm *tab);

int tabcpy(int alloc, const struct tabprm *tabsrc, struct tabprm *tabdst);

int tabcmp(int cmp, double tol, const struct tabprm *tab1,
           const struct tabprm *tab2, int *equal);

int tabfree(struct tabprm *tab);

int tabsize(const struct tabprm *tab, int size[2]);

int tabprt(const struct tabprm *tab);

int tabperr(const struct tabprm *tab, const char *prefix);

int tabset(struct tabprm *tab);

int tabx2s(struct tabprm *tab, int ncoord, int nelem, const double x[],
           double world[], int stat[]);

int tabs2x(struct tabprm *tab, int ncoord, int nelem, const double world[],
           double x[], int stat[]);


// Deprecated.
#define tabini_errmsg tab_errmsg
#define tabcpy_errmsg tab_errmsg
#define tabfree_errmsg tab_errmsg
#define tabprt_errmsg tab_errmsg
#define tabset_errmsg tab_errmsg
#define tabx2s_errmsg tab_errmsg
#define tabs2x_errmsg tab_errmsg

#ifdef __cplusplus
}
#endif

#endif // WCSLIB_TAB
```

6.23 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 [ ]

6.23.1 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.

6.23.2 Macro Definition Documentation

WCSSUB_LONGITUDE

```
#define WCSSUB_LONGITUDE 0x1001
```

Mask to use for extracting the longitude axis when sub-imaging, refer to the *axes* argument of wcssub().

WCSSUB_LATITUDE

```
#define WCSSUB_LATITUDE 0x1002
```

Mask to use for extracting the latitude axis when sub-imaging, refer to the *axes* argument of wcssub().

WCSSUB_CUBEFACE

```
#define WCSSUB_CUBEFACE 0x1004
```

Mask to use for extracting the **CUBEFACE** axis when sub-imaging, refer to the *axes* argument of wcssub().

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().

WCSSUB_SPECTRAL

```
#define WCSSUB_SPECTRAL 0x1008
```

Mask to use for extracting the spectral axis when sub-imaging, refer to the *axes* argument of wcssub().

WCSSUB_STOKES

```
#define WCSSUB_STOKES 0x1010
```

Mask to use for extracting the **STOKES** axis when sub-imaging, refer to the *axes* argument of wcssub().

WCSSUB_TIME

```
#define WCSSUB_TIME 0x1020
```

WCSCOMPARE_ANCILLARY

```
#define WCSCOMPARE_ANCILLARY 0x0001
```

WCSCOMPARE_TILING

```
#define WCSCOMPARE_TILING 0x0002
```

WCSCOMPARE_CRPIX

```
#define WCSCOMPARE_CRPIX 0x0004
```

PVLEN

```
#define PVLEN (sizeof(struct pvcard)/sizeof(int))
```

PSLEN

```
#define PSLEN (sizeof(struct pscard)/sizeof(int))
```

AUXLEN

```
#define AUXLEN (sizeof(struct auxprm)/sizeof(int))
```

WCSLEN

```
#define WCSLEN (sizeof(struct wcsprm)/sizeof(int))
```

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

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.

wcsini_errmsg

```
#define wcsini_errmsg wcs_errmsg
```

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

wcssub_errmsg

```
#define wcssub_errmsg wcs_errmsg
```

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

wcscopy_errmsg

```
#define wcscopy_errmsg wcs_errmsg
```

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

wcsfree_errmsg

```
#define wcsfree_errmsg wcs_errmsg
```

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

wcsprt_errmsg

```
#define wcsprt_errmsg wcs_errmsg
```

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

wcsset_errmsg

```
#define wcsset_errmsg wcs_errmsg
```

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

wcsp2s_errmsg

```
#define wcsp2s_errmsg wcs_errmsg
```

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

wcss2p_errmsg

```
#define wcss2p_errmsg wcs_errmsg
```

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

wcsmix_errmsg

```
#define wcsmix_errmsg wcs_errmsg
```

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

6.23.3 Enumeration Type Documentation

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	

6.23.4 Function Documentation

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 `PVi_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.

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 `PSi_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.

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.

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().

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.

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).

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: • WCSCOMPARE_ANCILLARY: Ignore ancillary keywords that don't change the WCS transformation, such as **DATE-OBS** or EQUINOX. • WCSCOMPARE_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. • WCSCOMPARE_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 WCSCOMPARE_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.

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.

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.

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.

auxsize()

```
int auxsize (
            const struct auxprm * aux,
            int sizes[2] )
```

auxsize() computes the full size of an 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.

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.

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.

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.
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.

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.

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. For time axes, imgcrd[][wcs.time] is the intermediate time coordinate.
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, world[][wcs.spec] is the spectral coordinate, in SI units. For time axes, world[][wcs.time] is the time coordinate.
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().

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. For time axes, world[][wcs.time] is the time coordinate.
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. For time axes, imgcrd[][wcs.time] is the intermediate time coordinate.
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().

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.

Parameters

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.

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.
```

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().

wcslib_version()

```
const char * wcslib_version (
            int vers[3] )
```

6.23.5 Variable Documentation

wcs_errmsg

```
const char* wcs_errmsg[] [extern]
```

6.24 wcs.h

Go to the documentation of this file.

```
/*============================================================================
  WCSLIB 8.1 - an implementation of the FITS WCS standard.
  Copyright (C) 1995-2023, Mark Calabretta

  This file is part of WCSLIB.

  WCSLIB is free software: you can redistribute it and/or modify it under the
  terms of the GNU Lesser General Public License as published by the Free
  Software Foundation, either version 3 of the License, or (at your option)
  any later version.

  WCSLIB is distributed in the hope that it will be useful, but WITHOUT ANY
  WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
```

```
  FOR A PARTICULAR PURPOSE.  See the GNU Lesser General Public License for
  more details.

  You should have received a copy of the GNU Lesser General Public License
  along with WCSLIB.  If not, see http://www.gnu.org/licenses.

  Author: Mark Calabretta, Australia Telescope National Facility, CSIRO.
  http://www.atnf.csiro.au/people/Mark.Calabretta
  $Id: wcs.h,v 8.1 2023/07/05 17:12:07 mcalabre Exp $
*=============================================================================
*
* WCSLIB 8.1 - C routines that implement the FITS World Coordinate System
* (WCS) standard.  Refer to the README file provided with WCSLIB for an
* overview of the library.
*
*
* Summary of the wcs routines
* ---------------------------
* 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.
*
*
* wcsnpv() - Memory allocation for PVi_ma
* ---------------------------------------
* wcsnpv() sets or gets the value of NPVMAX (default 64).  This global
* variable controls the number of pvcard structs, for holding PVi_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.
*
* Given:
*   n         int       Value of NPVMAX; ignored if < 0.  Use a value less
*                       than zero to get the current value.
*
* Function return value:
*             int       Current value of NPVMAX.
*
*
* wcsnps() - Memory allocation for PSi_ma
* ---------------------------------------
* wcsnps() sets or gets the value of NPSMAX (default 8).  This global variable
* controls the number of pscard structs, for holding PSi_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.
*
* Given:
*   n         int       Value of NPSMAX; ignored if < 0.  Use a value less
*                       than zero to get the current value.
*
* Function return value:
*             int       Current value of NPSMAX.
*
*
* wcsini() - Default constructor for the wcsprm struct
* ----------------------------------------------------
* 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.
*
*
* wcsinit() - Default constructor for the wcsprm struct
* -----------------------------------------------------
* 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.
*
* Given:
*   alloc     int       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.)
*
*   naxis     int         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.
*
* Given and returned:
*   wcs       struct wcsprm*
*                         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).
*
* Given:
*   npvmax    int         The number of PVi_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.
*
*   npsmax    int         The number of PSi_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.
*
*   ndpmax    int         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.
*
* Function return value:
*             int         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().
*
*
* wcsauxi() - Default constructor for the auxprm struct
* -----------------------------------------------------
* wcsauxi() optionally allocates memory for an auxprm struct, attaches it to
* wcsprm, and sets all members of the struct to default values.
*
* Given:
*   alloc     int         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.)
*
* Given and returned:
*   wcs       struct wcsprm*
*                         Coordinate transformation parameters.
*
* Function return value:
*             int         Status return value:
*                           0: Success.
*                           1: Null wcsprm pointer passed.
*                           2: Memory allocation failed.
*
*
* wcssub() - Subimage extraction routine for the wcsprm struct
* ------------------------------------------------------------
* 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().
*
* Given:
*   alloc     int       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.
*
*   wcssrc    const struct wcsprm*
*                       Struct to extract from.
*
* Given and returned:
*   nsub      int*
*   axes      int[]     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().
*
*   wcsdst    struct wcsprm*
*                       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).
*
* Function return value:
*             int       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).
*
*
* wcscompare() - Compare two wcsprm structs for equality
* ------------------------------------------------------
* wcscompare() compares two wcsprm structs for equality.
*
* Given:
*   cmp       int       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:
*                         WCSCOMPARE_ANCILLARY: Ignore ancillary keywords
*                           that don't change the WCS transformation, such
*                           as DATE-OBS or EQUINOX.
*                         WCSCOMPARE_TILING: Ignore integral differences in
*                           CRPIXja.  This is the 'tiling' condition, where
*                           two WCSes cover different regions of the same
*                           map projection and align on the same map grid.
*                         WCSCOMPARE_CRPIX: Ignore any differences at all in
*                           CRPIXja.  The two WCSes cover different regions
*                           of the same map projection but may not align on
*                           the same map grid.  Overrides WCSCOMPARE_TILING.
*
*   tol       double    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.
*
*   wcs1      const struct wcsprm*
*                       The first wcsprm struct to compare.
*
*   wcs2      const struct wcsprm*
*                       The second wcsprm struct to compare.
*
* Returned:
*   equal     int*      Non-zero when the given structs are equal.
*
* Function return value:
*             int       Status return value:
*                         0: Success.
*                         1: Null pointer passed.
*
*
* wcscopy() macro - Copy routine for the wcsprm struct
* ----------------------------------------------------
* 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.
*
*
* wcsfree() - Destructor for the wcsprm struct
* --------------------------------------------
* 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().
*
* Given and returned:
*   wcs       struct wcsprm*
*                       Coordinate transformation parameters.
*
* Function return value:
*             int       Status return value:
*                         0: Success.
*                         1: Null wcsprm pointer passed.
*
*
* wcstrim() - Free unused arrays in the wcsprm struct
* ---------------------------------------------------
* 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.
*
* Given and returned:
*   wcs       struct wcsprm*
*                       Coordinate transformation parameters.
*
* Function return value:
*             int       Status return value:
*                         0: Success.
*                         1: Null wcsprm pointer passed.
*                        14: wcsprm struct is unset.
*
*
* wcssize() - Compute the size of a wcsprm struct
* -----------------------------------------------
* wcssize() computes the full size of a wcsprm struct, including allocated
* memory.
*
* Given:
*   wcs       const struct wcsprm*
*                       Coordinate transformation parameters.
*
*                       If NULL, the base size of the struct and the allocated
*                       size are both set to zero.
*
* Returned:
*   sizes     int[2]    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.
*
* Function return value:
*             int       Status return value:
*                         0: Success.
*
*
* auxsize() - Compute the size of a auxprm struct
* -----------------------------------------------
* auxsize() computes the full size of an auxprm struct, including allocated
* memory.
*
* Given:
*   aux       const struct auxprm*
*                       Auxiliary coordinate information.
```

```
*
*                        If NULL, the base size of the struct and the allocated
*                        size are both set to zero.
*
* Returned:
*   sizes     int[2]     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.
*
* Function return value:
*             int        Status return value:
*                          0: Success.
*
*
* wcsprt() - Print routine for the wcsprm struct
* ----------------------------------------------
* wcsprt() prints the contents of a wcsprm struct using wcsprintf().  Mainly
* intended for diagnostic purposes.
*
* Given:
*   wcs       const struct wcsprm*
*                        Coordinate transformation parameters.
*
* Function return value:
*             int        Status return value:
*                          0: Success.
*                          1: Null wcsprm pointer passed.
*
*
* wcsperr() - Print error messages from a wcsprm struct
* -----------------------------------------------------
* 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.
*
* Given:
*   wcs       const struct wcsprm*
*                        Coordinate transformation parameters.
*
*   prefix    const char *
*                        If non-NULL, each output line will be prefixed with
*                        this string.
*
* Function return value:
*             int        Status return value:
*                          0: Success.
*                          1: Null wcsprm pointer passed.
*
*
* wcsbchk() - Enable/disable bounds checking
* ------------------------------------------
* 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.
*
* Given and returned:
*   wcs       struct wcsprm*
*                        Coordinate transformation parameters.
*
* Given:
*   bounds    int        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.
*
* Function return value:
*             int        Status return value:
*                          0: Success.
*                          1: Null wcsprm pointer passed.
*
*
* wcsset() - Setup routine for the wcsprm struct
* ----------------------------------------------
* 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.
*
* Given and returned:
*   wcs       struct wcsprm*
*                       Coordinate transformation parameters.
*
* Function return value:
*             int       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.
*
*
* wcsp2s() - Pixel-to-world transformation
* ----------------------------------------
* wcsp2s() transforms pixel coordinates to world coordinates.
*
* Given and returned:
*   wcs       struct wcsprm*
*                       Coordinate transformation parameters.
*
* Given:
*   ncoord,
*   nelem     int       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.
*
*   pixcrd    const double[ncoord][nelem]
*                       Array of pixel coordinates.
*
* Returned:
*   imgcrd    double[ncoord][nelem]
*                       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.  For time axes,
*                       imgcrd[][wcs.time] is the intermediate time
*                       coordinate.
*
*   phi,theta double[ncoord]
*                       Longitude and latitude in the native coordinate system
*                       of the projection [deg].
*
*   world     double[ncoord][nelem]
*                       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.  For time axes, world[][wcs.time] is the
*                       time coordinate.
*
*   stat      int[ncoord]
*                       Status return value for each coordinate:
*                         0: Success.
*                        1+: A bit mask indicating invalid pixel coordinate
*                            element(s).
*
* Function return value:
*             int       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().
 *
 *
 * wcss2p() - World-to-pixel transformation
 * ----------------------------------------
 * wcss2p() transforms world coordinates to pixel coordinates.
 *
 * Given and returned:
 *   wcs       struct wcsprm*
 *                       Coordinate transformation parameters.
 *
 * Given:
 *   ncoord,
 *   nelem     int       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.
 *
 *   world     const double[ncoord][nelem]
 *                       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.  For time axes, world[][wcs.time] is the
 *                       time coordinate.
 *
 * Returned:
 *   phi,theta double[ncoord]
 *                       Longitude and latitude in the native coordinate
 *                       system of the projection [deg].
 *
 *   imgcrd    double[ncoord][nelem]
 *                       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.  For time axes,
 *                       imgcrd[][wcs.time] is the intermediate time
 *                       coordinate.
 *
 *   pixcrd    double[ncoord][nelem]
 *                       Array of pixel coordinates.
 *
 *   stat      int[ncoord]
 *                       Status return value for each coordinate:
 *                         0: Success.
 *                        1+: A bit mask indicating invalid world coordinate
 *                            element(s).
 *
 * Function return value:
 *             int       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().
 *
 *
 * wcsmix() - Hybrid coordinate transformation
 * -------------------------------------------
 * 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.
*
* Given and returned:
*   wcs       struct wcsprm*
*                       Indices for the celestial coordinates obtained
*                       by parsing the wcsprm::ctype[].
*
* Given:
*   mixpix    int       Which element of the pixel coordinate is given.
*
*   mixcel    int       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].
*
*   vspan     const double[2]
*                       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.
*
*   vstep     const double
*                       Step size for solution search [deg].  If zero, a
*                       sensible, although perhaps non-optimal default will be
*                       used.
*
*   viter     int       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.
*
* Given and returned:
*   world     double[naxis]
*                       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.
*
* Returned:
*   phi,theta double[naxis]
*                       Longitude and latitude in the native coordinate
*                       system of the projection [deg].
*
*   imgcrd    double[naxis]
*                       Image coordinate elements.  imgcrd[wcs.lng] and
*                       imgcrd[wcs.lat] are the projected x-, and
*                       y-coordinates in pseudo "degrees".
*
* Given and returned:
*   pixcrd    double[naxis]
*                       Pixel coordinate.  The element indicated by mixpix is
*                       given and the remaining elements are returned.
*
* Function return value:
*             int       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.
 *
 *
 * wcsccs() - Change celestial coordinate system
 * ---------------------------------------------
 * 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().
 *
 * Given and returned:
 *   wcs       struct wcsprm*
 *                       Coordinate transformation parameters.  Particular
 *                       "values to be given" elements of the wcsprm struct
 *                       are modified.
 *
 * Given:
 *   lng2p1,
 *   lat2p1    double    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.
 *
 *   lng1p2    double    Longitude in the original celestial coordinate system
 *                       of the pole (i.e. latitude +90) of the new system
 *                       [deg].  See note 1 below.
 *
 *   clng,clat const char*
 *                       Longitude and latitude identifiers of the new CTYPEia
 *                       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.
 *
 *   radesys   const char*
 *                       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.
 *
 *   equinox   double    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.
 *
 *   alt       const char*
 *                       Character code for alternate coordinate descriptions
 *                       (i.e. the 'a' in keyword names such as CTYPEia).  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.
*
* Function return value:
*             int       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.
*
*
* wcssptr() - Spectral axis translation
* -------------------------------------
* 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().
*
* Given and returned:
*   wcs       struct wcsprm*
*                       Coordinate transformation parameters.
*
*   i         int*      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.
*
*   ctype     char[9]   Desired spectral CTYPEia.  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.
 *
 * Function return value:
 *             int       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().
 *
 *
 * wcslib_version() - WCSLIB version number
 * ----------------------------------------
 * wcslib_version() returns the WCSLIB version number.
 *
 * The major version number changes when the ABI changes or when the license
 * conditions change.  ABI changes typically result from a change to the
 * contents of one of the structs.  The major version number is used to
 * distinguish between incompatible versions of the sharable library.
 *
 * The minor version number changes with new functionality or bug fixes that do
 * not involve a change in the ABI.
 *
 * The auxiliary version number (which is often absent) signals changes to the
 * documentation, test suite, build procedures, or any other change that does
 * not affect the compiled library.
 *
 * Returned:
 *   vers[3]   int[3]    The broken-down version number:
 *                         0: Major version number.
 *                         1: Minor version number.
 *                         2: Auxiliary version number (zero if absent).
 *                       May be given as a null pointer if not required.
 *
 * Function return value:
 *             char*     A null-terminated, statically allocated string
 *                       containing the version number in the usual form, i.e.
 *                       "<major>.<minor>.<auxiliary>".
 *
 *
 * wcsprm struct - Coordinate transformation parameters
 * ----------------------------------------------------
 * 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.
 *
 *   int 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.
*
*   int 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.
*
*   double *crpix
*     (Given) Address of the first element of an array of double containing
*     the coordinate reference pixel, CRPIXja.
*
*   double *pc
*     (Given) Address of the first element of the PCi_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.
*
*   double *cdelt
*     (Given) Address of the first element of an array of double containing
*     the coordinate increments, CDELTia.
*
*   double *crval
*     (Given) Address of the first element of an array of double containing
*     the coordinate reference values, CRVALia.
*
*   char (*cunit)[72]
*     (Given) Address of the first element of an array of char[72] containing
*     the CUNITia keyvalues which define the units of measurement of the
*     CRVALia, CDELTia, and CDi_ja keywords.
*
*     As CUNITia 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.
*
*   char (*ctype)[72]
*     (Given) Address of the first element of an array of char[72] containing
*     the coordinate axis types, CTYPEia.
*
*     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.
*
*   double lonpole
*     (Given and returned) The native longitude of the celestial pole, phi_p,
*     given by LONPOLEa [deg] or by PVi_2a [deg] attached to the longitude
*     axis which takes precedence if defined, and ...
*   double latpole
*     (Given and returned) ... the native latitude of the celestial pole,
*     theta_p, given by LATPOLEa [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.
*
*   double restfrq
*     (Given) The rest frequency [Hz], and/or ...
*   double restwav
*     (Given) ... the rest wavelength in vacuo [m], only one of which need be
*     given, the other should be set to zero.
*
*   int npv
*     (Given) The number of entries in the wcsprm::pv[] array.
*
*   int 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().
*
*   struct pvcard *pv
*     (Given) Address of the first element of an array of length npvmax of
*     pvcard structs.
*
*     As a FITS header parser encounters each PVi_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), LONPOLEa and LATPOLEa respectively.
*
*   int nps
*     (Given) The number of entries in the wcsprm::ps[] array.
*
*   int 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().
*
*   struct pscard *ps
*     (Given) Address of the first element of an array of length npsmax of
*     pscard structs.
*
*     As a FITS header parser encounters each PSi_ma keyword it should load it
*     into a pscard struct in the array and increment nps.  wcsset()
*     interprets these as required (currently no PSi_ma keyvalues are
```

```
*     recognized).
*
*   double *cd
*     (Given) For historical compatibility, the wcsprm struct supports two
*     alternate specifications of the linear transformation matrix, those
*     associated with the CDi_ja keywords, and ...
*   double *crota
*     (Given) ... those associated with the CROTAi keywords.  Although these
*     may not formally co-exist with PCi_ja, the approach taken here is simply
*     to ignore them if given in conjunction with PCi_ja.
*
*   int altlin
*     (Given) altlin is a bit flag that denotes which of the PCi_ja, CDi_ja
*     and CROTAi keywords are present in the header:
*
*     - Bit 0: PCi_ja is present.
*
*     - Bit 1: CDi_ja is present.
*
*       Matrix elements in the IRAF convention are equivalent to the product
*       CDi_ja = CDELTia * PCi_ja, but the defaults differ from that of the
*       PCi_ja matrix.  If one or more CDi_ja keywords are present then all
*       unspecified CDi_ja default to zero.  If no CDi_ja (or CROTAi) keywords
*       are present, then the header is assumed to be in PCi_ja form whether
*       or not any PCi_ja keywords are present since this results in an
*       interpretation of CDELTia consistent with the original FITS
*       specification.
*
*       While CDi_ja may not formally co-exist with PCi_ja, it may co-exist
*       with CDELTia and CROTAi which are to be ignored.
*
*     - Bit 2: CROTAi is present.
*
*       In the AIPS convention, CROTAi 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 CDELTia; any other CROTAi
*       keywords are ignored.
*
*       CROTAi may not formally co-exist with PCi_ja.
*
*       CROTAi and CDELTia may formally co-exist with CDi_ja but if so are to
*       be ignored.
*
*     - Bit 3: PCi_ja + CDELTia was derived from CDi_ja by wcspcx().
*
*       This bit is set by wcspcx() when it derives PCi_ja and CDELTia from
*       CDi_ja via an orthonormal decomposition.  In particular, it signals
*       wcsset() not to replace PCi_ja by a copy of CDi_ja with CDELTia set
*       to unity.
*
*     CDi_ja and CROTAi 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 PCi_ja keyword is encountered: altlin |= 1;
*
*     - Whenever a CDi_ja keyword is encountered: altlin |= 2;
*
*     - Whenever a CROTAi keyword is encountered: altlin |= 4;
*
*     If none of these bits are set the PCi_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 PCi_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 CDi_ja is present (and no
*     PCi_ja).
*
*     If CROTAi are present but none is associated with the latitude axis
*     (and no PCi_ja or CDi_ja), then wcsset() reverts to a unity PCi_ja
*     matrix.
*
*   int 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.
*
*   char alt[4]
*     (Given, auxiliary) Character code for alternate coordinate descriptions
*     (i.e. the 'a' in keyword names such as CTYPEia).  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.
*
*   int 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.
*
*   int *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.
*
*   char (*cname)[72]
*     (Given, auxiliary) The address of the first element of an array of
*     char[72] containing the coordinate axis names, CNAMEia.
*
*     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.
*
*   double *crder
*     (Given, auxiliary) Address of the first element of an array of double
*     recording the random error in the coordinate value, CRDERia.
*
*     It is not necessary to reset the wcsprm struct (via wcsset()) when
*     wcsprm::crder is changed.
*
*   double *csyer
*     (Given, auxiliary) Address of the first element of an array of double
*     recording the systematic error in the coordinate value, CSYERia.
*
*     It is not necessary to reset the wcsprm struct (via wcsset()) when
*     wcsprm::csyer is changed.
*
*   double *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.
*
*   double *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.
*
*   char wcsname[72]
*     (Given, auxiliary) The name given to the coordinate representation,
*     WCSNAMEa.  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.
*
*   char timesys[72]
*     (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 CTYPEia
*     set to 'TIME'.
*
*     It is not necessary to reset the wcsprm struct (via wcsset()) when
*     wcsprm::timesys is changed.
*
*   char trefpos[72]
*     (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.
*
*   char trefdir[72]
*     (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.
*
*   char plephem[72]
*     (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.
*
*   char timeunit[72]
*     (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
*     CUNITia for time axes.
*
*     It is not necessary to reset the wcsprm struct (via wcsset()) when
*     wcsprm::timeunit is changed.
*
*   char dateref[72]
*     (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.
*
*   double mjdref[2]
*     (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.
*
*   double 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.
*
*   char dateobs[72]
*     (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-ddThh:mm:ss.
*
*     It is not necessary to reset the wcsprm struct (via wcsset()) when
*     wcsprm::dateobs is changed.
*
*   char datebeg[72]
*     (Given, auxiliary) DATE-BEG keyvalue, being the date at the start of the
*     observation in ISO format, yyyy-mm-ddThh:mm:ss.
*
*     It is not necessary to reset the wcsprm struct (via wcsset()) when
*     wcsprm::datebeg is changed.
*
*   char dateavg[72]
*     (Given, auxiliary) DATE-AVG keyvalue, being the date at a representative
*     mid-point of the observation in ISO format, yyyy-mm-ddThh:mm:ss.
*
*     It is not necessary to reset the wcsprm struct (via wcsset()) when
*     wcsprm::dateavg is changed.
*
*   char dateend[72]
*     (Given, auxiliary) DATE-END keyvalue, baing the date at the end of the
*     observation in ISO format, yyyy-mm-ddThh:mm:ss.
*
*     It is not necessary to reset the wcsprm struct (via wcsset()) when
*     wcsprm::dateend is changed.
*
*   double 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.
*
*   double 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.
*
*   double 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.
*
*   double 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.
*
*   double jepoch
*     (Given, auxiliary) JEPOCH 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.
*
*   double bepoch
*     (Given, auxiliary) BEPOCH 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.
*
*   double 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.
*
*   double 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.
*
*   double 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.
*
*   double 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.
*
*   double 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.
*
*   double 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.
*
*   double 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.
*
*   double 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.
```

```
*
*   double obsgeo[6]
*     (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.
*
*   char obsorbit[72]
*     (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.
*
*   char radesys[72]
*     (Given, auxiliary) The equatorial or ecliptic coordinate system type,
*     RADESYSa.
*
*     It is not necessary to reset the wcsprm struct (via wcsset()) when
*     wcsprm::radesys is changed.
*
*   double equinox
*     (Given, auxiliary) The equinox associated with dynamical equatorial or
*     ecliptic coordinate systems, EQUINOXa (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.
*
*   char specsys[72]
*     (Given, auxiliary) Spectral reference frame (standard of rest),
*     SPECSYSa.
*
*     It is not necessary to reset the wcsprm struct (via wcsset()) when
*     wcsprm::specsys is changed.
*
*   char ssysobs[72]
*     (Given, auxiliary) The spectral reference frame in which there is no
*     differential variation in the spectral coordinate across the
*     field-of-view, SSYSOBSa.
*
*     It is not necessary to reset the wcsprm struct (via wcsset()) when
*     wcsprm::ssysobs is changed.
*
*   double 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, VELOSYSa.
*
*     It is not necessary to reset the wcsprm struct (via wcsset()) when
*     wcsprm::velosys is changed.
*
*   double zsource
*     (Given, auxiliary) The redshift, ZSOURCEa, of the source.
*
*     It is not necessary to reset the wcsprm struct (via wcsset()) when
*     wcsprm::zsource is changed.
*
*   char ssyssrc[72]
*     (Given, auxiliary) The spectral reference frame (standard of rest),
*     SSYSSRCa, in which wcsprm::zsource was measured.
*
*     It is not necessary to reset the wcsprm struct (via wcsset()) when
*     wcsprm::ssyssrc is changed.
*
*   double 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.
*
*   struct auxprm *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.
*
*   int ntab
*     (Given) See wcsprm::tab.
```

```
*
*   int nwtb
*     (Given) See wcsprm::wtb.
*
*   struct tabprm *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.
*
*   struct wtbarr *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.
*
*   char lngtyp[8]
*     (Returned) Four-character WCS celestial longitude and ...
*   char lattyp[8]
*     (Returned) ... latitude axis types. e.g. "RA", "DEC", "GLON", "GLAT",
*     etc. extracted from 'RA--', 'DEC-', 'GLON', 'GLAT', etc. in the first
*     four characters of CTYPEia but with trailing dashes removed.  (Declared
*     as char[8] for alignment reasons.)
*
*   int lng
*     (Returned) Index for the longitude coordinate, and ...
*   int lat
*     (Returned) ... index for the latitude coordinate, and ...
*   int spec
*     (Returned) ... index for the spectral coordinate, and ...
*   int time
*     (Returned) ... index for the time 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 PCi_ja matrix does not transpose axes.
*
*   int 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).
*
*   int *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.
*
*     CTYPEia in "4-3" form with unrecognized algorithm code will have its
*     type set to -1 and generate an error.
*
```

```
 *   struct linprm lin
 *     (Returned) Linear transformation parameters (usage is described in the
 *     prologue to lin.h).
 *
 *   struct celprm cel
 *     (Returned) Celestial transformation parameters (usage is described in
 *     the prologue to cel.h).
 *
 *   struct spcprm spc
 *     (Returned) Spectral transformation parameters (usage is described in the
 *     prologue to spc.h).
 *
 *   struct wcserr *err
 *     (Returned) If enabled, when an error status is returned, this struct
 *     contains detailed information about the error, see wcserr_enable().
 *
 *   int m_flag
 *     (For internal use only.)
 *   int m_naxis
 *     (For internal use only.)
 *   double *m_crpix
 *     (For internal use only.)
 *   double *m_pc
 *     (For internal use only.)
 *   double *m_cdelt
 *     (For internal use only.)
 *   double *m_crval
 *     (For internal use only.)
 *   char (*m_cunit)[72]
 *     (For internal use only.)
 *   char (*m_ctype)[72]
 *     (For internal use only.)
 *   struct pvcard *m_pv
 *     (For internal use only.)
 *   struct pscard *m_ps
 *     (For internal use only.)
 *   double *m_cd
 *     (For internal use only.)
 *   double *m_crota
 *     (For internal use only.)
 *   int *m_colax
 *     (For internal use only.)
 *   char (*m_cname)[72]
 *     (For internal use only.)
 *   double *m_crder
 *     (For internal use only.)
 *   double *m_csyer
 *     (For internal use only.)
 *   double *m_czphs
 *     (For internal use only.)
 *   double *m_cperi
 *     (For internal use only.)
 *   struct tabprm *m_tab
 *     (For internal use only.)
 *   struct wtbarr *m_wtb
 *     (For internal use only.)
 *
 *
 * pvcard struct - Store for PVi_ma keyrecords
 * -------------------------------------------
 * The pvcard struct is used to pass the parsed contents of PVi_ma keyrecords
 * to wcsset() via the wcsprm struct.
 *
 * All members of this struct are to be set by the user.
 *
 *   int i
 *     (Given) Axis number (1-relative), as in the FITS PVi_ma keyword.  If
 *     i == 0, wcsset() will replace it with the latitude axis number.
 *
 *   int m
 *     (Given) Parameter number (non-negative), as in the FITS PVi_ma keyword.
 *
 *   double value
 *     (Given) Parameter value.
 *
 *
 * pscard struct - Store for PSi_ma keyrecords
 * -------------------------------------------
 * The pscard struct is used to pass the parsed contents of PSi_ma keyrecords
 * to wcsset() via the wcsprm struct.
 *
 * All members of this struct are to be set by the user.
 *
 *   int i
 *     (Given) Axis number (1-relative), as in the FITS PSi_ma keyword.
 *
 *   int m
```

```
*     (Given) Parameter number (non-negative), as in the FITS PSi_ma keyword.
*
*   char value[72]
*     (Given) Parameter value.
*
*
* auxprm struct - Additional auxiliary parameters
* -----------------------------------------------
* 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.
*
*   double rsun_ref
*     (Given, auxiliary) Reference radius of the Sun used in coordinate
*     calculations (m).
*
*   double dsun_obs
*     (Given, auxiliary) Distance between the centre of the Sun and the
*     observer (m).
*
*   double crln_obs
*     (Given, auxiliary) Carrington heliographic longitude of the observer
*     (deg).
*
*   double hgln_obs
*     (Given, auxiliary) Stonyhurst heliographic longitude of the observer
*     (deg).
*
*   double hglt_obs
*     (Given, auxiliary) Heliographic latitude (Carrington or Stonyhurst) of
*     the observer (deg).
*
*   double a_radius
*     Length of the semi-major axis of a triaxial ellipsoid approximating the
*     shape of a body (e.g. planet) in the solar system (m).
*
*   double b_radius
*     Length of the intermediate axis, normal to the semi-major and semi-minor
*     axes, of a triaxial ellipsoid approximating the shape of a body (m).
*
*   double c_radius
*     Length of the semi-minor axis, normal to the semi-major axis, of a
*     triaxial ellipsoid approximating the shape of a body (m).
*
*   double blon_obs
*     Bodycentric longitude of the observer in the coordinate system fixed to
*     the planet or other solar system body (deg, in range 0 to 360).
*
*   double blat_obs
*     Bodycentric latitude of the observer in the coordinate system fixed to
*     the planet or other solar system body (deg).
*
*   double bdis_obs
*     Bodycentric distance of the observer (m).
*
* Global variable: const char *wcs_errmsg[] - Status return messages
* ------------------------------------------------------------------
* Error messages to match the status value returned from each function.
*
*===========================================================================*/

#ifndef WCSLIB_WCS
#define WCSLIB_WCS

#include "lin.h"
#include "cel.h"
#include "spc.h"

#ifdef __cplusplus
extern "C" {
#define wtbarr wtbarr_s         // See prologue of wtbarr.h.
#endif

#define WCSSUB_LONGITUDE 0x1001
#define WCSSUB_LATITUDE  0x1002
#define WCSSUB_CUBEFACE  0x1004
#define WCSSUB_CELESTIAL 0x1007
#define WCSSUB_SPECTRAL  0x1008
#define WCSSUB_STOKES    0x1010
#define WCSSUB_TIME      0x1020


#define WCSCOMPARE_ANCILLARY 0x0001
#define WCSCOMPARE_TILING    0x0002
#define WCSCOMPARE_CRPIX     0x0004
```

```c


extern const char *wcs_errmsg[];

enum wcs_errmsg_enum {
  WCSERR_SUCCESS         =  0,    // Success.
  WCSERR_NULL_POINTER    =  1,       // Null wcsprm pointer passed.
  WCSERR_MEMORY          =  2,     // Memory allocation failed.
  WCSERR_SINGULAR_MTX    =  3,       // Linear transformation matrix is singular.
  WCSERR_BAD_CTYPE       =  4,  // Inconsistent or unrecognized coordinate
                                // axis type.
  WCSERR_BAD_PARAM       =  5,  // Invalid parameter value.
  WCSERR_BAD_COORD_TRANS =  6,    // Unrecognized coordinate transformation
                                // parameter.
  WCSERR_ILL_COORD_TRANS =  7,    // Ill-conditioned coordinate transformation
                                // parameter.
  WCSERR_BAD_PIX         =  8,    // One or more of the pixel coordinates were
                                // invalid.
  WCSERR_BAD_WORLD       =  9,  // One or more of the world coordinates were
                                // invalid.
  WCSERR_BAD_WORLD_COORD = 10,    // Invalid world coordinate.
  WCSERR_NO_SOLUTION     = 11,        // No solution found in the specified
                                // interval.
  WCSERR_BAD_SUBIMAGE    = 12,       // Invalid subimage specification.
  WCSERR_NON_SEPARABLE   = 13,      // Non-separable subimage coordinate system.
  WCSERR_UNSET           = 14       // wcsprm struct is unset.
};


// Struct used for storing PVi_ma keywords.
struct pvcard {
  int i;                        // Axis number, as in PVi_ma (1-relative).
  int m;                        // Parameter number, ditto  (0-relative).
  double value;                     // Parameter value.
};

// Size of the pvcard struct in int units, used by the Fortran wrappers.
#define PVLEN (sizeof(struct pvcard)/sizeof(int))

// Struct used for storing PSi_ma keywords.
struct pscard {
  int i;                        // Axis number, as in PSi_ma (1-relative).
  int m;                        // Parameter number, ditto  (0-relative).
  char value[72];           // Parameter value.
};

// Size of the pscard struct in int units, used by the Fortran wrappers.
#define PSLEN (sizeof(struct pscard)/sizeof(int))

// Struct used to hold additional auxiliary parameters.
struct auxprm {
  double rsun_ref;               // Solar radius.
  double dsun_obs;               // Distance from Sun centre to observer.
  double crln_obs;               // Carrington heliographic lng of observer.
  double hgln_obs;               // Stonyhurst heliographic lng of observer.
  double hglt_obs;               // Heliographic latitude of observer.

  double a_radius;               // Semi-major axis of solar system body.
  double b_radius;               // Semi-intermediate axis of solar system body.
  double c_radius;               // Semi-minor axis of solar system body.
  double blon_obs;               // Bodycentric longitude of observer.
  double blat_obs;               // Bodycentric latitude of observer.
  double bdis_obs;               // Bodycentric distance of observer.
  double dummy[2];               // Reserved for future use.
};

// Size of the auxprm struct in int units, used by the Fortran wrappers.
#define AUXLEN (sizeof(struct auxprm)/sizeof(int))


struct wcsprm {
  // Initialization flag (see the prologue above).
  //------------------------------------------------------------------------
  int    flag;                       // Set to zero to force initialization.

  // FITS header keyvalues to be provided (see the prologue above).
  //------------------------------------------------------------------------
  int    naxis;                     // Number of axes (pixel and coordinate).
  double *crpix;            // CRPIXja keyvalues for each pixel axis.
  double *pc;                  // PCi_ja  linear transformation matrix.
  double *cdelt;            // CDELTia keyvalues for each coord axis.
  double *crval;            // CRVALia keyvalues for each coord axis.

  char   (*cunit)[72];              // CUNITia keyvalues for each coord axis.
  char   (*ctype)[72];              // CTYPEia keyvalues for each coord axis.

  double lonpole;                 // LONPOLEa keyvalue.
```

```
  double latpole;                // LATPOLEa keyvalue.

  double restfrq;                // RESTFRQa keyvalue.
  double restwav;                // RESTWAVa keyvalue.

  int    npv;                        // Number of PVi_ma keywords, and the
  int    npvmax;           // number for which space was allocated.
  struct pvcard *pv;             // PVi_ma keywords for each i and m.

  int    nps;                        // Number of PSi_ma keywords, and the
  int    npsmax;           // number for which space was allocated.
  struct pscard *ps;             // PSi_ma keywords for each i and m.

  // Alternative header keyvalues (see the prologue above).
  //------------------------------------------------------------------------
  double *cd;                 // CDi_ja linear transformation matrix.
  double *crota;           // CROTAi keyvalues for each coord axis.
  int    altlin;           // Alternative representations
                                //   Bit 0: PCi_ja is present,
                                //   Bit 1: CDi_ja is present,
                                //   Bit 2: CROTAi is present.
  int    velref;           // AIPS velocity code, VELREF.

  // Auxiliary coordinate system information of a general nature.  Not
  // used by WCSLIB.  Refer to the prologue comments above for a brief
  // explanation of these values.
  char   alt[4];
  int    colnum;
  int    *colax;
                                // Auxiliary coordinate axis information.
  char   (*cname)[72];
  double *crder;
  double *csyer;
  double *czphs;
  double *cperi;

  char   wcsname[72];
                                // Time reference system and measurement.
  char   timesys[72], trefpos[72], trefdir[72], plephem[72];
  char   timeunit[72];
  char   dateref[72];
  double mjdref[2];
  double timeoffs;
                                // Data timestamps and durations.
  char   dateobs[72], datebeg[72], dateavg[72], dateend[72];
  double mjdobs, mjdbeg, mjdavg, mjdend;
  double jepoch, bepoch;
  double tstart, tstop;
  double xposure, telapse;
                                // Timing accuracy.
  double timsyer, timrder;
  double timedel, timepixr;
                                // Spatial & celestial reference frame.
  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;

  // Additional auxiliary coordinate system information of a specialist
  // nature.  Not used by WCSLIB.  Refer to the prologue comments above.
  struct auxprm *aux;

  // Coordinate lookup tables (see the prologue above).
  //------------------------------------------------------------------------
  int    ntab;                      // Number of separate tables.
  int    nwtb;                      // Number of wtbarr structs.
  struct tabprm *tab;           // Tabular transformation parameters.
  struct wtbarr *wtb;           // Array of wtbarr structs.

  //------------------------------------------------------------------------
  // Information derived from the FITS header keyvalues by wcsset().
  //------------------------------------------------------------------------
  char   lngtyp[8], lattyp[8];      // Celestial axis types, e.g. RA, DEC.
  int    lng, lat, spec, time;    // Longitude, latitude, spectral, and time
                                // axis indices (0-relative).
  int    cubeface;              // True if there is a CUBEFACE axis.
  int    dummy;                     // Dummy for alignment purposes.
  int    *types;           // Coordinate type codes for each axis.

  struct linprm lin;            //    Linear transformation parameters.
  struct celprm cel;            // Celestial transformation parameters.
```

```
  struct spcprm spc;          //  Spectral transformation parameters.

  //----------------------------------------------------------------------------
  //             THE REMAINDER OF THE WCSPRM STRUCT IS PRIVATE.
  //----------------------------------------------------------------------------

  // Error handling, if enabled.
  //----------------------------------------------------------------------------
  struct wcserr *err;

  // Memory management.
  //----------------------------------------------------------------------------
  int    m_flag, m_naxis;
  double *m_crpix, *m_pc, *m_cdelt, *m_crval;
  char  (*m_cunit)[72], (*m_ctype)[72];
  struct pvcard *m_pv;
  struct pscard *m_ps;
  double *m_cd, *m_crota;
  int    *m_colax;
  char  (*m_cname)[72];
  double *m_crder, *m_csyer, *m_czphs, *m_cperi;
  struct auxprm *m_aux;
  struct tabprm *m_tab;
  struct wtbarr *m_wtb;
};

// Size of the wcsprm struct in int units, used by the Fortran wrappers.
#define WCSLEN (sizeof(struct wcsprm)/sizeof(int))


int wcsnpv(int n);

int wcsnps(int n);

int wcsini(int alloc, int naxis, struct wcsprm *wcs);

int wcsinit(int alloc, int naxis, struct wcsprm *wcs, int npvmax, int npsmax,
            int ndpmax);

int wcsauxi(int alloc, struct wcsprm *wcs);

int wcssub(int alloc, const struct wcsprm *wcssrc, int *nsub, int axes[],
           struct wcsprm *wcsdst);

int wcscompare(int cmp, double tol, const struct wcsprm *wcs1,
               const struct wcsprm *wcs2, int *equal);

int wcsfree(struct wcsprm *wcs);

int wcstrim(struct wcsprm *wcs);

int wcssize(const struct wcsprm *wcs, int sizes[2]);

int auxsize(const struct auxprm *aux, int sizes[2]);

int wcsprt(const struct wcsprm *wcs);

int wcsperr(const struct wcsprm *wcs, const char *prefix);

int wcsbchk(struct wcsprm *wcs, int bounds);

int wcsset(struct wcsprm *wcs);

int wcsp2s(struct wcsprm *wcs, int ncoord, int nelem, const double pixcrd[],
           double imgcrd[], double phi[], double theta[], double world[],
           int stat[]);

int wcss2p(struct wcsprm *wcs, int ncoord, int nelem, const double world[],
           double phi[], double theta[], double imgcrd[], double pixcrd[],
           int stat[]);

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[]);

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);

int wcssptr(struct wcsprm *wcs, int *i, char ctype[9]);

const char* wcslib_version(int vers[3]);

// Defined mainly for backwards compatibility, use wcssub() instead.
#define wcscopy(alloc, wcssrc, wcsdst) wcssub(alloc, wcssrc, 0x0, 0x0, wcsdst)
```

```
// Deprecated.
#define wcsini_errmsg wcs_errmsg
#define wcssub_errmsg wcs_errmsg
#define wcscopy_errmsg wcs_errmsg
#define wcsfree_errmsg wcs_errmsg
#define wcsprt_errmsg wcs_errmsg
#define wcsset_errmsg wcs_errmsg
#define wcsp2s_errmsg wcs_errmsg
#define wcss2p_errmsg wcs_errmsg
#define wcsmix_errmsg wcs_errmsg

#ifdef __cplusplus
#undef wtbarr
}
#endif

#endif // WCSLIB_WCS
```

6.25 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.

6.25.1 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!

6.25.2 Macro Definition Documentation

ERRLEN

```
#define ERRLEN (sizeof(struct wcserr)/sizeof(int))
```

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.

6.25.3 Function Documentation

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.

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.

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.

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.

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.

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.

6.26 wcserr.h

Go to the documentation of this file.

```
/*============================================================================
  WCSLIB 8.1 - an implementation of the FITS WCS standard.
  Copyright (C) 1995-2023, Mark Calabretta

  This file is part of WCSLIB.

  WCSLIB is free software: you can redistribute it and/or modify it under the
  terms of the GNU Lesser General Public License as published by the Free
  Software Foundation, either version 3 of the License, or (at your option)
  any later version.

  WCSLIB is distributed in the hope that it will be useful, but WITHOUT ANY
  WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
  FOR A PARTICULAR PURPOSE.  See the GNU Lesser General Public License for
  more details.

  You should have received a copy of the GNU Lesser General Public License
  along with WCSLIB.  If not, see http://www.gnu.org/licenses.

  Author: Mark Calabretta, Australia Telescope National Facility, CSIRO.
  Module author: Michael Droettboom
  http://www.atnf.csiro.au/people/Mark.Calabretta
  $Id: wcserr.h,v 8.1 2023/07/05 17:12:07 mcalabre Exp $
*=============================================================================
*
* WCSLIB 8.1 - C routines that implement the FITS World Coordinate System
* (WCS) standard.  Refer to the README file provided with WCSLIB for an
* overview of the library.
*
* Summary of the wcserr routines
* ------------------------------
* 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!
*
*
* wcserr struct - Error message handling
* --------------------------------------
* 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.
*
*   int 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.
*
*   int 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.
*
*   char *msg
*     Informative error message.
*
*
* wcserr_enable() - Enable/disable error messaging
* ------------------------------------------------
* wcserr_enable() enables or disables wcserr error messaging.  By default it
* is disabled.
*
* PLEASE NOTE: This function is not thread-safe.
*
* Given:
*   enable    int       If true (non-zero), enable error messaging, else
*                       disable it.
*
* Function return value:
*             int       Status return value:
*                         0: Error messaging is disabled.
*                         1: Error messaging is enabled.
*
*
* wcserr_size() - Compute the size of a wcserr struct
* ---------------------------------------------------
* wcserr_size() computes the full size of a wcserr struct, including allocated
* memory.
*
* Given:
*   err       const struct wcserr*
*                       The error object.
*
*                       If NULL, the base size of the struct and the allocated
*                       size are both set to zero.
*
* Returned:
*   sizes     int[2]    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.
*
* Function return value:
*             int       Status return value:
*                         0: Success.
*
*
* wcserr_prt() - Print a wcserr struct
* ------------------------------------
* wcserr_prt() prints the error message (if any) contained in a wcserr struct.
* It uses the wcsprintf() functions.
*
* Given:
*   err       const struct wcserr*
```

```
*                       The error object.  If NULL, nothing is printed.
*
*   prefix    const char *
*                       If non-NULL, each output line will be prefixed with
*                       this string.
*
* Function return value:
*             int       Status return value:
*                         0: Success.
*                         2: Error messaging is not enabled.
*
*
* wcserr_clear() - Clear a wcserr struct
* --------------------------------------
* wcserr_clear() clears (deletes) a wcserr struct.
*
* Given and returned:
*   err       struct wcserr**
*                       The error object.  If NULL, nothing is done.  Set to
*                       NULL on return.
*
* Function return value:
*             int       Status return value:
*                         0: Success.
*
*
* wcserr_set() - Fill in the contents of an error object
* ------------------------------------------------------
* 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.
*
* Given and returned:
*   err       struct wcserr**
*                       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).
*
* Given:
*   status    int       Numeric status code to set.  If 0, then *err will be
*                       deleted and *err will be returned as NULL.
*
*   function  const char *
*                       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.
*
*   file      const char *
*                       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.
*
*   line_no   int       Line number in the source file generating the error
*                       such as given by the __LINE__ preprocessor macro.
*
*   format    const char *
*                       Format string of the error message.  May contain
*                       printf-style %-formatting codes.
*
*   ...       mixed     The remaining variable arguments are applied (like
*                       printf) to the format string to generate the error
*                       message.
*
* Function return value:
*             int       The status return code passed in.
*
*
* wcserr_copy() - Copy an error object
* ------------------------------------
* 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.
*
* Given:
*   src       const struct wcserr*
```

```
*                        Source error object.  If src is NULL, dst is cleared.
*
* Returned:
*   dst       struct wcserr*
*                        Destination error object.  If NULL, no copy is made.
*
* Function return value:
*             int       Numeric status code of the source error object.
*
*
* WCSERR_SET() macro - Fill in the contents of an error object
* ------------------------------------------------------------
* 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.
*
*===========================================================================*/

#ifndef WCSLIB_WCSERR
#define WCSLIB_WCSERR

#ifdef __cplusplus
extern "C" {
#endif

struct wcserr {
  int  status;                     // Status code for the error.
  int  line_no;                    // Line number where the error occurred.
  const char *function;            // Function name.
  const char *file;            // Source file name.
  char *msg;                   // Informative error message.
};

// Size of the wcserr struct in int units, used by the Fortran wrappers.
#define ERRLEN (sizeof(struct wcserr)/sizeof(int))

int wcserr_enable(int enable);

int wcserr_size(const struct wcserr *err, int sizes[2]);

int wcserr_prt(const struct wcserr *err, const char *prefix);

int wcserr_clear(struct wcserr **err);


// INTERNAL USE ONLY -------------------------------------------------------

int wcserr_set(struct wcserr **err, int status, const char *function,
  const char *file, int line_no, const char *format, ...);

int wcserr_copy(const struct wcserr *src, struct wcserr *dst);

// Convenience macro for invoking wcserr_set().
#define WCSERR_SET(status) err, status, function, __FILE__, __LINE__

#ifdef __cplusplus
}
#endif

#endif // WSCLIB_WCSERR
```

6.27 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 **CD**`i_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 **OBS**GEO-[XYZLBH] vector of observatory coordinates.*
- int unitfix (int ctrl, struct wcsprm ∗wcs)

  *Correct aberrant **CUNIT**`ia` 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.

6.27.1 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.

6.27.2 Macro Definition Documentation

CDFIX

```
#define CDFIX 0
```

Index of the status value returned by cdfix() in the status vector returned by wcsfix().

DATFIX

```
#define DATFIX 1
```

Index of the status value returned by datfix() in the status vector returned by wcsfix().

OBSFIX

```
#define OBSFIX 2
```

UNITFIX

```
#define UNITFIX 3
```

Index of the status value returned by unitfix() in the status vector returned by wcsfix().

SPCFIX

```
#define SPCFIX 4
```

Index of the status value returned by spcfix() in the status vector returned by wcsfix().

CELFIX

```
#define CELFIX 5
```

Index of the status value returned by celfix() in the status vector returned by wcsfix().

CYLFIX

```
#define CYLFIX 6
```

Index of the status value returned by cylfix() in the status vector returned by wcsfix().

NWCSFIX

```
#define NWCSFIX 7
```

Number of elements in the status vector returned by wcsfix().

cylfix_errmsg

```
#define cylfix_errmsg wcsfix_errmsg
```

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

6.27.3 Enumeration Type Documentation

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	

6.27.4 Function Documentation

wcsfix()

```
int wcsfix (
            int ctrl,
            const int naxis[],
            struct wcsprm * wcs,
            int stat[] )
```

wcsfix() is identical to **wcsfixi**(), but lacks the info argument.

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.
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.

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.

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*.

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.

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.

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.

celfix()

```
int celfix (
            struct wcsprm * wcs )
```

celfix() translates AIPS-convention celestial projection types, **NCP** and **GLS**, set in the ctype[] member of the wcsprm 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().

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().

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 Sum(j=1,N)(**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.

Parameters

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.

6.27.5 Variable Documentation

wcsfix_errmsg

`const char * wcsfix_errmsg[] [extern]`

Error messages to match the status value returned from each function.

6.28 wcsfix.h

Go to the documentation of this file.

```
/*============================================================================
  WCSLIB 8.1 - an implementation of the FITS WCS standard.
  Copyright (C) 1995-2023, Mark Calabretta

  This file is part of WCSLIB.

  WCSLIB is free software: you can redistribute it and/or modify it under the
  terms of the GNU Lesser General Public License as published by the Free
  Software Foundation, either version 3 of the License, or (at your option)
  any later version.

  WCSLIB is distributed in the hope that it will be useful, but WITHOUT ANY
  WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
  FOR A PARTICULAR PURPOSE.  See the GNU Lesser General Public License for
  more details.

  You should have received a copy of the GNU Lesser General Public License
  along with WCSLIB.  If not, see http://www.gnu.org/licenses.

  Author: Mark Calabretta, Australia Telescope National Facility, CSIRO.
  http://www.atnf.csiro.au/people/Mark.Calabretta
  $Id: wcsfix.h,v 8.1 2023/07/05 17:12:07 mcalabre Exp $
*=============================================================================
```

```
*
* WCSLIB 8.1 - C routines that implement the FITS World Coordinate System
* (WCS) standard.  Refer to the README file provided with WCSLIB for an
* overview of the library.
*
*
* Summary of the wcsfix routines
* ------------------------------
* 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 CTYPEia 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 CDi_ja matrix to 1.0 if all
*     CDi_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 CUNITia keyvalues, e.g. 'DEG' -> 'deg'.
 *
 *   - spcfix(): translate AIPS-convention spectral types, 'FREQ-LSR',
 *     'FELO-HEL', etc., in ctype[] as set from CTYPEia.
 *
 *   - celfix(): translate AIPS-convention celestial projection types, NCP and
 *     GLS, in ctype[] as set from CTYPEia.
 *
 *   - cylfix(): fixes WCS keyvalues for malformed cylindrical projections that
 *     suffer from the problem described in Sect. 7.3.4 of Paper I.
 *
 *
 * wcsfix() - Translate a non-standard WCS struct
 * ----------------------------------------------
 * wcsfix() is identical to wcsfixi(), but lacks the info argument.
 *
 *
 * wcsfixi() - Translate a non-standard WCS struct
 * -----------------------------------------------
 * wcsfixi() applies all of the corrections handled separately by cdfix(),
 * datfix(), obsfix(), unitfix(), spcfix(), celfix(), and cylfix().
 *
 * Given:
 *   ctrl      int       Do potentially unsafe translations of non-standard
 *                       unit strings as described in the usage notes to
 *                       wcsutrn().
 *
 *   naxis     const int []
 *                       Image axis lengths.  If this array pointer is set to
 *                       zero then cylfix() will not be invoked.
 *
 * Given and returned:
 *   wcs       struct wcsprm*
 *                       Coordinate transformation parameters.
 *
 * Returned:
 *   stat      int [NWCSFIX]
 *                       Status returns from each of the functions.  Use the
 *                       preprocessor macros NWCSFIX to dimension this vector
 *                       and CDFIX, DATFIX, OBSFIX, UNITFIX, SPCFIX, CELFIX,
 *                       and CYLFIX to access its elements.  A status value
 *                       of -2 is set for functions that were not invoked.
 *
 *   info      struct wcserr [NWCSFIX]
 *                       Status messages from each of the functions.  Use the
 *                       preprocessor macros NWCSFIX to dimension this vector
 *                       and CDFIX, DATFIX, OBSFIX, 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().
 *
 * Function return value:
 *             int       Status return value:
 *                         0: Success.
 *                         1: One or more of the translation functions
 *                            returned an error.
 *
 *
 * cdfix() - Fix erroneously omitted CDi_ja keywords
 * -------------------------------------------------
 * cdfix() sets the diagonal element of the CDi_ja matrix to unity if all
 * CDi_ja keywords associated with a given axis were omitted.  According to WCS
 * Paper I, if any CDi_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.
 *
 * Given and returned:
 *   wcs       struct wcsprm*
 *                       Coordinate transformation parameters.
 *
 * Function return value:
 *             int       Status return value:
 *                        -1: No change required (not an error).
 *                         0: Success.
 *                         1: Null wcsprm pointer passed.
 *
 *
```

```
* datfix() - Translate DATE-OBS and derive MJD-OBS or vice versa
* ---------------------------------------------------------------
* datfix() translates the old DATE-OBS date format set in wcsprm::dateobs to
* year-2000 standard form (yyyy-mm-ddThh: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().
*
* Given and returned:
*   wcs       struct wcsprm*
*                       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.
*
* Function return value:
*             int       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.
*
*
* obsfix() - complete the OBSGEO-[XYZLBH] vector of observatory coordinates
* -------------------------------------------------------------------------
* 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().
*
* Given:
*   ctrl      int       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.
*
* Given and returned:
*   wcs       struct wcsprm*
*                       Coordinate transformation parameters.
*                       wcsprm::obsgeo may be changed.
*
* Function return value:
*             int       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.
*
*
* unitfix() - Correct aberrant CUNITia keyvalues
* ----------------------------------------------
* unitfix() applies wcsutrn() to translate non-standard CUNITia keyvalues,
* e.g. 'DEG' -> 'deg', also stripping off unnecessary whitespace.
*
* unitfix() is expected to be invoked before wcsset(), which will fail if
* non-standard CUNITia keyvalues have not been translated.
*
* Given:
*   ctrl      int       Do potentially unsafe translations described in the
*                       usage notes to wcsutrn().
*
* Given and returned:
*   wcs       struct wcsprm*
*                       Coordinate transformation parameters.
*
* Function return value:
*             int       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.
*
*
* spcfix() - Translate AIPS-convention spectral types
* ---------------------------------------------------
* 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 CTYPEia 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 CTYPEia keyvalues.
*
* The operations done by spcfix() do not affect and are not affected by
* wcsset().
*
* Given and returned:
*   wcs       struct wcsprm*
*                       Coordinate transformation parameters.  wcsprm::ctype[]
*                       and/or wcsprm::specsys may be changed.
*
* Function return value:
*             int       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.
*
*
* celfix() - Translate AIPS-convention celestial projection types
* ---------------------------------------------------------------
* celfix() translates AIPS-convention celestial projection types, NCP and
* GLS, set in the ctype[] member of the wcsprm 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 CTYPEia 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 CTYPEia keyvalues and the relevant PVi_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.
*
* Given and returned:
*   wcs       struct wcsprm*
*                       Coordinate transformation parameters.  wcsprm::ctype[]
*                       and/or wcsprm::pv[] may be changed.
*
* Function return value:
*             int       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().
*
*
* cylfix() - Fix malformed cylindrical projections
* ------------------------------------------------
* 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().
*
* Given:
*   naxis     const int []
```

```
*                       Image axis lengths.
*
* Given and returned:
*   wcs       struct wcsprm*
*                       Coordinate transformation parameters.
*
* Function return value:
*             int       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().
*
*
* wcspcx() - regularize PCi_j
* ---------------------------
* wcspcx() "regularizes" the linear transformation matrix component of the
* coordinate transformation (PCi_ja) to make it more human-readable.
*
* Normally, upon encountering a FITS header containing a CDi_ja matrix,
* wcsset() simply treats it as PCi_ja and sets CDELTia to unity.  However,
* wcspcx() decomposes CDi_ja into PCi_ja and CDELTia in such a way that
* CDELTia form meaningful scaling parameters.  In practice, the residual
* PCi_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 Sum(j=1,N)(PCi_ja)^2 = 1,
* in replacing the sum over j with the sum over i.  Consequently, the columns
* of PCi_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 CDELTia is chosen to make the PCi_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 CDELTia is set
* negative for the longitude axis and positive for the latitude axis.
*
* Optionally, rows of the PCi_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 CDi_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 CDELTia, if CDi_ja was
* translated to PCi_ja plus CDELTia, 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().
*
* Given and returned:
*   wcs       struct wcsprm*
*                       Coordinate transformation parameters.
*
* Given:
*   dopc      int       If 1, then PCi_ja and CDELTia, as given, will be
*                       recomposed according to the above prescription.  If 0,
*                       the operation is restricted to decomposing CDi_ja.
*
*   permute   int       If 1, then after decomposition (or recomposition),
*                       permute rows of PCi_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 CTYPEia,
*                       CRVALia, CDELTia, etc., including coordinate lookup
*                       tables, will also be permuted as necessary to account
*                       for the change to PCi_ja.  This does not apply to
*                       CRPIXja, nor prior distortion functions.  These
*                       operate on pixel coordinates, which are not affected
*                       by the permutation.
*
* Returned:
*   rotn      double[2] Rotation angle [deg] of each basis vector associated
*                       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.
*
* Function return value:
*             int       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.
*
*
* Global variable: const char *wcsfix_errmsg[] - Status return messages
* ---------------------------------------------------------------------
* Error messages to match the status value returned from each function.
*
*===========================================================================*/

#ifndef WCSLIB_WCSFIX
#define WCSLIB_WCSFIX

#include "wcs.h"
#include "wcserr.h"

#ifdef __cplusplus
extern "C" {
#endif

#define CDFIX    0
#define DATFIX   1
#define OBSFIX   2
#define UNITFIX  3
#define SPCFIX   4
#define CELFIX   5
#define CYLFIX   6
#define NWCSFIX  7

extern const char *wcsfix_errmsg[];
#define cylfix_errmsg wcsfix_errmsg

enum wcsfix_errmsg_enum {
  FIXERR_OBSGEO_FIX       = -5, // Observatory coordinates amended.
  FIXERR_DATE_FIX         = -4, // Date string reformatted.
  FIXERR_SPC_UPDATE       = -3, // Spectral axis type modified.
  FIXERR_UNITS_ALIAS      = -2,       // Units alias translation.
  FIXERR_NO_CHANGE        = -1, // No change.
  FIXERR_SUCCESS          =  0,   // Success.
  FIXERR_NULL_POINTER     =  1,      // Null wcsprm pointer passed.
  FIXERR_MEMORY           =  2,    // Memory allocation failed.
  FIXERR_SINGULAR_MTX     =  3,      // Linear transformation matrix is singular.
  FIXERR_BAD_CTYPE        =  4, // Inconsistent or unrecognized coordinate
                                // axis types.
  FIXERR_BAD_PARAM        =  5, // Invalid parameter value.
  FIXERR_BAD_COORD_TRANS  =  6,   // Invalid coordinate transformation
                                // parameters.
  FIXERR_ILL_COORD_TRANS  =  7,   // Ill-conditioned coordinate transformation
                                // parameters.
  FIXERR_BAD_CORNER_PIX   =  8,    // All of the corner pixel coordinates are
                                // invalid.
  FIXERR_NO_REF_PIX_COORD =  9,  // Could not determine reference pixel
                                // coordinate.
  FIXERR_NO_REF_PIX_VAL   = 10     // Could not determine reference pixel value.
};

int wcsfix(int ctrl, const int naxis[], struct wcsprm *wcs, int stat[]);

int wcsfixi(int ctrl, const int naxis[], struct wcsprm *wcs, int stat[],
```

```
            struct wcserr info[]);

int cdfix(struct wcsprm *wcs);

int datfix(struct wcsprm *wcs);

int obsfix(int ctrl, struct wcsprm *wcs);

int unitfix(int ctrl, struct wcsprm *wcs);

int spcfix(struct wcsprm *wcs);

int celfix(struct wcsprm *wcs);

int cylfix(const int naxis[], struct wcsprm *wcs);

int wcspcx(struct wcsprm *wcs, int dopc, int permute, double rotn[2]);


#ifdef __cplusplus
}
#endif

#endif // WCSLIB_WCSFIX
```

6.29 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 **CD00**i**00**j.*
- #define WCSHDR_PC00i00j 0x00000008

  *Bit mask for wcspih() and wcsbth() - accept **PC00**i**00**j.*
- #define WCSHDR_PROJPn 0x00000010

  *Bit mask for wcspih() and wcsbth() - accept **PROJP**n.*
- #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 **DOBS**n.*
- #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 ***EPOCH***`a`.
- #define WCSHDR_VSOURCE 0x00002000

  Bit mask for wcspih() and wcsbth() - accept ***VSOURCE***`a`.
- #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 `i`***CNAM***`n`, ***TCNAM***`n`, `i`***CRDE***`n`, ***TCRDE***`n`, `i`***CSYE***`n`, ***TCSYE***`n`.
- #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 ***DOBS***`n`.
- #define WCSHDO_TPCn_ka 0x00002

  Bit mask for wcshdo() - write ***TPC***`n_ka`.
- #define WCSHDO_PVn_ma 0x00004

  Bit mask for wcshdo() - write `i`***PV***`n_ma`, ***TPV***`n_ma`, `i`***PS***`n_ma`, ***TPS***`n_ma`.
- #define WCSHDO_CRPXna 0x00008

  Bit mask for 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`.
- #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 represention.
- 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.

6.29.1 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.

6.29.2 Macro Definition Documentation

WCSHDR_none

```
#define WCSHDR_none 0x00000000
```

Bit mask for the *relax* argument of wcspih() and wcsbth() - reject all extensions.

Refer to wcsbth() note 5.

WCSHDR_all

```
#define WCSHDR_all 0x000FFFFF
```

Bit mask for the *relax* argument of wcspih() and wcsbth() - accept all extensions.

Refer to wcsbth() note 5.

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.

WCSHDR_strict

```
#define WCSHDR_strict 0x20000000
```

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.

WCSHDR_VELREFa

```
#define WCSHDR_VELREFa 0x00000002
```

Bit mask for the *relax* argument of wcspih() and wcsbth() - accept **VELREF**a.

Refer to wcsbth() note 5.

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.

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.

WCSHDR_PROJPn

```
#define WCSHDR_PROJPn 0x00000010
```

Bit mask for the *relax* argument of wcspih() and wcsbth() - accept **PROJP**n.

Refer to wcsbth() note 5.

WCSHDR_CD0i_0ja

```
#define WCSHDR_CD0i_0ja 0x00000020
```

WCSHDR_PC0i_0ja

```
#define WCSHDR_PC0i_0ja 0x00000040
```

WCSHDR_PV0i_0ma

```
#define WCSHDR_PV0i_0ma 0x00000080
```

WCSHDR_PS0i_0ma

```
#define WCSHDR_PS0i_0ma 0x00000100
```

WCSHDR_DOBSn

```
#define WCSHDR_DOBSn 0x00000200
```

Bit mask for the *relax* argument of wcspih() and wcsbth() - accept **DOBS**n.

Refer to wcsbth() note 5.

WCSHDR_OBSGLBHn

```
#define WCSHDR_OBSGLBHn 0x00000400
```

WCSHDR_RADECSYS

```
#define WCSHDR_RADECSYS 0x00000800
```

Bit mask for the *relax* argument of wcspih() and wcsbth() - accept **RADECSYS**.

Refer to wcsbth() note 5.

WCSHDR_EPOCHa

```
#define WCSHDR_EPOCHa 0x00001000
```

Bit mask for the *relax* argument of wcspih() and wcsbth() - accept **EPOCH**a.

Refer to wcsbth() note 5.

WCSHDR_VSOURCE

```
#define WCSHDR_VSOURCE 0x00002000
```

Bit mask for the *relax* argument of wcspih() and wcsbth() - accept **VSOURCE**a.

Refer to wcsbth() note 5.

WCSHDR_DATEREF

```
#define WCSHDR_DATEREF 0x00004000
```

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.

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.

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.

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.

WCSHDR_IMGHEAD

```
#define WCSHDR_IMGHEAD 0x00100000
```

Bit mask for the *keysel* argument of wcsbth() - restrict keyword types considered to image header keywords only.

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.

WCSHDR_PIXLIST

```
#define WCSHDR_PIXLIST 0x00400000
```

Bit mask for the *keysel* argument of wcsbth() - restrict keyword types considered to pixel list keywords only.

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().

WCSHDO_all

```
#define WCSHDO_all 0x000FF
```

Bit mask for the *relax* argument of wcshdo() - write all extensions.

Refer to the notes for wcshdo().

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().

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().

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().

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().

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().

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().

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().

WCSHDO_P12

```
#define WCSHDO_P12 0x01000
```

WCSHDO_P13

```
#define WCSHDO_P13 0x02000
```

WCSHDO_P14

```
#define WCSHDO_P14 0x04000
```

WCSHDO_P15

```
#define WCSHDO_P15 0x08000
```

WCSHDO_P16

```
#define WCSHDO_P16 0x10000
```

WCSHDO_P17

```
#define WCSHDO_P17 0x20000
```

WCSHDO_EFMT

```
#define WCSHDO_EFMT 0x40000
```

6.29.3 Enumeration Type Documentation

wcshdr_errmsg_enum

```
enum wcshdr_errmsg_enum
```

Enumerator

WCSHDRERR_SUCCESS	
WCSHDRERR_NULL_POINTER	
WCSHDRERR_MEMORY	
WCSHDRERR_BAD_COLUMN	
WCSHDRERR_PARSER	
WCSHDRERR_BAD_TABULAR_PARAMS	

6.29.4 Function Documentation

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.

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 $m = n \leq 9$, 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 keywords ({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. **TPC**n_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 **TPS**n_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

i**CRDE**na	**TCRDE**na	↩ :	--	i**CRD**na	--	**TCRD**na	**CRDER**ia
i**CSYE**na	**TCSYE**na	↩ :	--	i**CSY**na	--	**TCSY**na	**CSYER**ia
TCZPHna	**TCZPH**na	↩ :	--	**TCZP**na	--	**TCZP**na	CZPHSia
i**CPER**na	**TCPER**na	↩ :	--	i**CPR**na	--	**TCPR**na	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.

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().

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.

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' represention 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.

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.

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

  - jCRPna, iCDEna, iCUNna, iCTYna and iCRVna for bintables and
  - TCRPna, TCDEna, TCUNna, TCTYna and TCRVna 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
  - iCNAna, iCRDna, and iCSYna for bintables and
  - TCNAna, TCRDna, and TCSYna for pixel lists

  By analogy with the above, the long forms would be

  - iCNAMna, iCRDEna, and iCSYEna for bintables and
  - TCNAMna, TCRDEna, and TCSYEna 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.

6.29.5 Variable Documentation

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**().

6.30 wcshdr.h

Go to the documentation of this file.

```
/*============================================================================
  WCSLIB 8.1 - an implementation of the FITS WCS standard.
  Copyright (C) 1995-2023, Mark Calabretta

  This file is part of WCSLIB.

  WCSLIB is free software: you can redistribute it and/or modify it under the
  terms of the GNU Lesser General Public License as published by the Free
  Software Foundation, either version 3 of the License, or (at your option)
  any later version.

  WCSLIB is distributed in the hope that it will be useful, but WITHOUT ANY
  WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
  FOR A PARTICULAR PURPOSE.  See the GNU Lesser General Public License for
  more details.

  You should have received a copy of the GNU Lesser General Public License
  along with WCSLIB.  If not, see http://www.gnu.org/licenses.

  Author: Mark Calabretta, Australia Telescope National Facility, CSIRO.
  http://www.atnf.csiro.au/people/Mark.Calabretta
  $Id: wcshdr.h,v 8.1 2023/07/05 17:12:07 mcalabre Exp $
*=============================================================================
*
* WCSLIB 8.1 - C routines that implement the FITS World Coordinate System
* (WCS) standard.  Refer to the README file provided with WCSLIB for an
* overview of the library.
*
*
* Summary of the wcshdr routines
* ------------------------------
* 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 PVi_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.
*
*
* wcspih() - FITS WCS parser routine for image headers
* ----------------------------------------------------
* 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).
*
* Given and returned:
*   header    char[]    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.
*
* Given:
*   nkeyrec   int       Number of keyrecords in header[].
*
*   relax     int       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.
*
*   ctrl      int       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.
*
* Returned:
*   nreject   int*      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.
*
*   nwcs      int*      Number of coordinate representations found.
*
*   wcs       struct wcsprm**
*                       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).
*
* Function return value:
*             int       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.
*
*
* wcsbth() - FITS WCS parser routine for binary table and image headers
* ---------------------------------------------------------------------
* 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.
*
* Given and returned:
*   header    char[]    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.
*
* Given:
*   nkeyrec   int       Number of keyrecords in header[].
*
*   relax     int       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.
*
*   ctrl      int       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.
*
*   keysel    int       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 EQUIna or RFRQna that are common to
*                       binary table image arrays and pixel lists (including
*                       WCSNna and TWCSna, 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.
*
*   colsel    int*      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 TPn_ka and TCn_ka (and
*                       TPCn_ka and TCDn_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.
*
* Returned:
*   nreject   int*      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.
*
*   nwcs      int*      Number of coordinate representations found.
*
*   wcs       struct wcsprm**
*                       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).
*
* Function return value:
*             int       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 CTYPEia) from the higher of
*
*        a: NAXIS,
*        b: WCSAXESa,
*        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 CROTAn, CDi_ja, and PCi_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 WCSNAMEa as
*      TWCSna instead of WCSNna; 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 WCSNna and TWCSna 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 CRPIXja keyword, but in
*                  fact the leading zero on the axis number violates the basic
```

```
*                 FITS standard.  Likewise, "LONPOLE2" is not a valid
*                 LONPOLEa 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 CROTAia (wcspih()),
*                                iCROTna (wcsbth()),
*                                TCROTna (wcsbth()).
*       - WCSHDR_VELREFa: Accept VELREFa.
*               wcspih() always recognizes the AIPS-convention keywords,
*               CROTAn, EPOCH, and VELREF for the primary representation
*               (a = ' ') but alternates are non-standard.
*
*               wcsbth() accepts EPOCHa and VELREFa only if WCSHDR_AUXIMG is
*               also enabled.
*
*       - WCSHDR_CD00i00j: Accept CD00i00j (wcspih()).
*       - WCSHDR_PC00i00j: Accept PC00i00j (wcspih()).
*       - WCSHDR_PROJPn:   Accept PROJPn   (wcspih()).
*               These appeared in early drafts of WCS Paper I+II (before they
*               were split) and are equivalent to CDi_ja, PCi_ja, and PVi_ma
*               for the primary representation (a = ' ').  PROJPn is
*               equivalent to PVi_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. CTYPEia) "shall
*               not have leading zeroes", and later in Sect. 8.1 that "leading
*               zeroes must not be used" on PVi_ma and PSi_ma.  However, by an
*               oversight, it is silent on PCi_ja and CDi_ja.
*
*       - WCSHDR_DOBSn (wcsbth() only): Allow DOBSn, the column-specific
*               analogue of DATE-OBS.  By an oversight this was never formally
*               defined in the standard.
*
*       - WCSHDR_OBSGLBHn (wcsbth() only): Allow OBSGLn, OBSGBn, and OBSGHn,
*               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 RADESYSa.
*
*               wcsbth() accepts RADECSYS only if WCSHDR_AUXIMG is also
*               enabled.
*
*       - WCSHDR_EPOCHa:  Accept EPOCHa.
*
*       - WCSHDR_VSOURCE: Accept VSOURCEa or VSOUna (wcsbth()).  This appeared
*               in early drafts of WCS Paper III and was subsequently dropped
*               in favour of ZSOURCEa and ZSOUna.
*
*               wcsbth() accepts VSOURCEa only if WCSHDR_AUXIMG is also
*               enabled.
*
*       - WCSHDR_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 keywords
*               ({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
*
#                jCRPXna  TCRPXna  :  jCRPXn  jCRPna  TCRPXn  TCRPna  CRPIXja
#                   -     TPCn_ka  :    -     ijPCna    -     TPn_ka  PCi_ja
#                   -     TCDn_ka  :    -     ijCDna    -     TCn_ka  CDi_ja
#                iCDLTna  TCDLTna  :  iCDLTn  iCDEna  TCDLTn  TCDEna  CDELTia
#                iCUNIna  TCUNIna  :  iCUNIn  iCUNna  TCUNIn  TCUNna  CUNITia
#                iCTYPna  TCTYPna  :  iCTYPn  iCTYna  TCTYPn  TCTYna  CTYPEia
#                iCRVLna  TCRVLna  :  iCRVLn  iCRVna  TCRVLn  TCRVna  CRVALia
```

```
#               iPVn_ma  TPVn_ma  :    -     iVn_ma    -     TVn_ma  PVi_ma
#               iPSn_ma  TPSn_ma  :    -     iSn_ma    -     TSn_ma  PSi_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, iPVn_ma, and TPVn_ma appeared by mistake
*             in the examples in WCS Paper II and subsequently these and
*             also TCDn_ka, iPSn_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
*
#               iCNAMna  TCNAMna  :   ---   iCNAna    ---   TCNAna  CNAMEia
#               iCRDEna  TCRDEna  :   ---   iCRDna    ---   TCRDna  CRDERia
#               iCSYEna  TCSYEna  :   ---   iCSYna    ---   TCSYna  CSYERia
#               iCZPHna  TCZPHna  :   ---   iCZPna    ---   TCZPna  CZPHSia
#               iCPERna  TCPERna  :   ---   iCPRna    ---   TCPRna  CPERIia
*
*             Note that CNAMEia, CRDERia, CSYERia, 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 iCNAMn, iCRDEn, iCSYEn, iCZPHn,
*             iCPERn, TCNAMn, TCRDEn, TCSYEn, TCZPHn, and TCPERn, i.e. with
*             "a" blank.  While non-standard, these are the obvious analogues
*             of iCTYPn, TCTYPn, 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 EQUINOXa would apply to all image
*             arrays in a binary table, or all pixel list columns with
*             alternate representation "a" unless overridden by EQUIna.
*
*             Specifically the keywords are:
*
#               LONPOLEa  for LONPna
#               LATPOLEa  for LATPna
#               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 WCSNna  ... Or TWCSna (see below).
#
#               DATE-OBS  for DOBSn
#               MJD-OBS   for MJDOBn
#
#               RADESYSa  for RADEna
#               RADECSYS  for RADEna  ... Only if WCSHDR_RADECSYS is set.
#               EPOCH          -       ... (No column-specific form.)
#               EPOCHa         -       ... Only if WCSHDR_EPOCHa is set.
#               EQUINOXa  for EQUIna
*
*             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 RFRQna
#               RESTFRQa  for RFRQna
#               RESTWAVa  for RWAVna
*
*             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 TRPOSn
#               TREFDIR   for TRDIRn
#               PLEPHEM        -
#               TIMEUNIT       -
```

```
#                DATEREF      -
#                MJDREF       -
#                MJDREFI      -
#                MJDREFF      -
#                JDREF        -
#                JDREFI       -
#                JDREFF       -
#                TIMEOFFS     -
#
#                DATE-BEG     -
#                DATE-AVG  for DAVGn
#                DATE-END     -
#                MJD-BEG      -
#                MJD-AVG   for MJDAn
#                MJD-END      -
#                JEPOCH       -
#                BEPOCH       -
#                TSTART       -
#                TSTOP        -
#                XPOSURE      -
#                TELAPSE      -
#
#                TIMSYER      -
#                TIMRDER      -
#                TIMEDEL      -
#                TIMEPIXR     -
#
#                OBSGEO-X  for OBSGXn
#                OBSGEO-Y  for OBSGYn
#                OBSGEO-Z  for OBSGZn
#                OBSGEO-L  for OBSGLn
#                OBSGEO-B  for OBSGBn
#                OBSGEO-H  for OBSGHn
#                OBSORBIT     -
#
#                SPECSYSa  for SPECna
#                SSYSOBSa  for SOBSna
#                VELOSYSa  for VSYSna
#                VSOURCEa  for VSOUna  ... Only if WCSHDR_VSOURCE is set.
#                ZSOURCEa  for ZSOUna
#                SSYSSRCa  for SSRCna
#                VELANGLa  for VANGna
*
*              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 LONPOLEa and
*              LATPOLEa, and also RADESYSa and EQUINOXa 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 MJDOBn 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 CTYPEia.
*
*        - 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 CRPIXja would apply to all image
*              arrays in a binary table with alternate representation "a"
*              unless overridden by jCRPna.
*
```

```
*              Specifically the keywords are those listed above for
*              WCSHDR_AUXIMG plus
*
#                WCSAXESa  for WCAXna
*
*              which defines the coordinate dimensionality, and the following
*              keywords that are parameterized by axis number:
*
#                CRPIXja   for jCRPna
#                PCi_ja    for ijPCna
#                CDi_ja    for ijCDna
#                CDELTia   for iCDEna
#                CROTAi    for iCROTn
#                CROTAia        -      ... Only if WCSHDR_CROTAia is set.
#                CUNITia   for iCUNna
#                CTYPEia   for iCTYna
#                CRVALia   for iCRVna
#                PVi_ma    for iVn_ma
#                PSi_ma    for iSn_ma
#
#                CNAMEia   for iCNAna
#                CRDERia   for iCRDna
#                CSYERia   for iCSYna
#                CZPHSia   for iCZPna
#                CPERIia   for iCPRna
*
*              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 CNAMEia, CRDERia, CSYERia, 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 CD00i00j and PC00i00j and reject all other
*      extensions, use
*
=        relax = WCSHDR_reject | WCSHDR_CD00i00j | WCSHDR_PC00i00j;
*
*      The parser always treats EPOCH as subordinate to EQUINOXa if both are
*      present, and VSOURCEa is always subordinate to ZSOURCEa.
*
*      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 WCSNna or MJDOBn,
*      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 TCTYna) 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 LONPna or
*      EQUIna), regardless of the table columns the image occupies.  In other
*      words, the column number is ignored (the "n" value in LONPna or
*      EQUIna).  This also applies for global, binary-table WCS keywords that
*      have no alternates, such as MJDOBn and OBSGXn, 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
*      OBSGXn, OBSGYn, and OBSGZn 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 MJDOBn
*      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.
*
*
* wcstab() - Tabular construction routine
* ---------------------------------------
* 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 PVi_ma and PSi_ma keywords in the image header.
*
* wcstab() parses the PVi_ma and PSi_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.
*
* Given and returned:
*   wcs       struct wcsprm*
*                       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().
*
* Function return value:
*             int       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().
*
*
* wcsidx() - Index alternate coordinate representations
* -----------------------------------------------------
* 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.
*
* Given:
*   nwcs      int       Number of coordinate representations in the array.
*
*   wcs       const struct wcsprm**
*                       Pointer to an array of wcsprm structs returned by
*                       wcspih() or wcsbth().
*
* Returned:
*   alts      int[27]   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];
*
* Function return value:
*             int       Status return value:
*                         0: Success.
*                         1: Null wcsprm pointer passed.
*
*
* wcsbdx() - Index alternate coordinate representions
* ---------------------------------------------------
* 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.
*
* Given:
*   nwcs      int       Number of coordinate representations in the array.
*
*   wcs       const struct wcsprm**
*                       Pointer to an array of wcsprm structs returned by
*                       wcsbth().
*
```

```
*   type      int       Select the type of coordinate representation:
*                         0: binary table image arrays,
*                         1: pixel lists.
*
* Returned:
*   alts      short[1000][28]
*                       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' represention 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];
*
* Function return value:
*             int       Status return value:
*                         0: Success.
*                         1: Null wcsprm pointer passed.
*
*
* wcsvfree() - Free the array of wcsprm structs
* ---------------------------------------------
* wcsvfree() frees the memory allocated by wcspih() or wcsbth() for the array
* of wcsprm structs, first invoking wcsfree() on each of the array members.
*
* Given and returned:
*   nwcs      int*      Number of coordinate representations found; set to 0
*                       on return.
*
*   wcs       struct wcsprm**
*                       Pointer to the array of wcsprm structs; set to 0x0 on
*                       return.
*
* Function return value:
*             int       Status return value:
*                         0: Success.
*                         1: Null wcsprm pointer passed.
*
*
* wcshdo() - Write out a wcsprm struct as a FITS 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 PCi_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. CROTAn, RESTFREQ, VELREF, RADECSYS, EPOCH, VSOURCEa) or
*     non-standard usage will be translated to standard (this is partially
*     dependent on whether wcsfix() was applied).
*
*   - Additional keywords such as WCSAXESa, CUNITia, LONPOLEa and LATPOLEa 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.
*
* Given:
*   ctrl      int       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 LONPOLEa, 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: CRPIXja, PCi_ja, and CDELTia
*                       (n.b. excluding CRVALia).  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, CDELTia
*                       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".
*
* Given and returned:
*   wcs       struct wcsprm*
*                       Pointer to a wcsprm struct containing coordinate
*                       transformation parameters.  Will be initialized if
*                       necessary.
*
* Returned:
*   nkeyrec   int*      Number of FITS header keyrecords returned in the
*                       "header" array.
*
*   header    char**    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.
*
* Function return value:
*             int       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 DOBSn, the column-specific analogue of DATE-OBS
*              for use in binary tables and pixel lists.  WCS Paper III
*              introduced DATE-AVG and DAVGn but by an oversight DOBSn (the
*              obvious analogy) was never formally defined by the standard.
*              The alternative to using DOBSn 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
*
*              - TPn_ka and TCn_ka for pixel lists
*
*              but WCS Paper II uses TPCn_ka in one example and subsequently
*              the errata for the WCS papers legitimized the use of
*
*              - TPCn_ka and TCDn_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
*
*              - iVn_ma and iSn_ma for bintables and
*              - TVn_ma and TSn_ma for pixel lists
*
*              but WCS Paper II uses iPVn_ma and TPVn_ma in the examples and
*              subsequently the errata for the WCS papers legitimized the use
*              of
*
*              - iPVn_ma and iPSn_ma for bintables and
*              - TPVn_ma and TPSn_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
*
*              - jCRPXn, iCDLTn, iCUNIn, iCTYPn, and iCRVLn for bintables and
*              - TCRPXn, TCDLTn, TCUNIn, TCTYPn, and TCRVLn 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
*
*              - jCRPna, iCDEna, iCUNna, iCTYna and iCRVna for bintables and
*              - TCRPna, TCDEna, TCUNna, TCTYna and TCRVna 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
*
*              - iCNAna,  iCRDna,  and iCSYna  for bintables and
*              - TCNAna,  TCRDna,  and TCSYna  for pixel lists
```

```
 *
 *               By analogy with the above, the long forms would be
 *
 *               - iCNAMna, iCRDEna, and iCSYEna for bintables and
 *               - TCNAMna, TCRDEna, and TCSYEna 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 WCSNna instead of
 *               TWCSna for pixel lists.  While wcsbth() treats WCSNna and
 *               TWCSna as equivalent, other parsers may not.  Consequently,
 *               this usage is potentially unsafe and is not recommended at this
 *               time.
 *
 *
 * Global variable: const char *wcshdr_errmsg[] - Status return messages
 * ---------------------------------------------------------------------
 * Error messages to match the status value returned from each function.
 * Use wcs_errmsg[] for status returns from wcshdo().
 *
 *===========================================================================*/

#ifndef WCSLIB_WCSHDR
#define WCSLIB_WCSHDR

#include "wcs.h"

#ifdef __cplusplus
extern "C" {
#endif

#define WCSHDR_none     0x00000000
#define WCSHDR_all      0x000FFFFF
#define WCSHDR_reject   0x10000000
#define WCSHDR_strict   0x20000000

#define WCSHDR_CROTAia  0x00000001
#define WCSHDR_VELREFa  0x00000002
#define WCSHDR_CD00i00j 0x00000004
#define WCSHDR_PC00i00j 0x00000008
#define WCSHDR_PROJPn   0x00000010
#define WCSHDR_CD0i_0ja 0x00000020
#define WCSHDR_PC0i_0ja 0x00000040
#define WCSHDR_PV0i_0ma 0x00000080
#define WCSHDR_PS0i_0ma 0x00000100
#define WCSHDR_DOBSn    0x00000200
#define WCSHDR_OBSGLBHn 0x00000400
#define WCSHDR_RADECSYS 0x00000800
#define WCSHDR_EPOCHa   0x00001000
#define WCSHDR_VSOURCE  0x00002000
#define WCSHDR_DATEREF  0x00004000
#define WCSHDR_LONGKEY  0x00008000
#define WCSHDR_CNAMn    0x00010000
#define WCSHDR_AUXIMG   0x00020000
#define WCSHDR_ALLIMG   0x00040000

#define WCSHDR_IMGHEAD  0x00100000
#define WCSHDR_BIMGARR  0x00200000
#define WCSHDR_PIXLIST  0x00400000

#define WCSHDO_none     0x00000
#define WCSHDO_all      0x000FF
#define WCSHDO_safe     0x0000F
#define WCSHDO_DOBSn    0x00001
#define WCSHDO_TPCn_ka  0x00002
#define WCSHDO_PVn_ma   0x00004
#define WCSHDO_CRPXna   0x00008
#define WCSHDO_CNAMna   0x00010
#define WCSHDO_WCSNna   0x00020
#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


extern const char *wcshdr_errmsg[];

enum wcshdr_errmsg_enum {
  WCSHDRERR_SUCCESS            = 0,    // Success.
  WCSHDRERR_NULL_POINTER       = 1,       // Null wcsprm pointer passed.
  WCSHDRERR_MEMORY             = 2,     // Memory allocation failed.
```

```
  WCSHDRERR_BAD_COLUMN          = 3, // Invalid column selection.
  WCSHDRERR_PARSER              = 4,     // Fatal error returned by Flex
                                         // parser.
  WCSHDRERR_BAD_TABULAR_PARAMS = 5  // Invalid tabular parameters.
};

int wcspih(char *header, int nkeyrec, int relax, int ctrl, int *nreject,
           int *nwcs, struct wcsprm **wcs);

int wcsbth(char *header, int nkeyrec, int relax, int ctrl, int keysel,
           int *colsel, int *nreject, int *nwcs, struct wcsprm **wcs);

int wcstab(struct wcsprm *wcs);

int wcsidx(int nwcs, struct wcsprm **wcs, int alts[27]);

int wcsbdx(int nwcs, struct wcsprm **wcs, int type, short alts[1000][28]);

int wcsvfree(int *nwcs, struct wcsprm **wcs);

int wcshdo(int ctrl, struct wcsprm *wcs, int *nkeyrec, char **header);


#ifdef __cplusplus
}
#endif

#endif // WCSLIB_WCSHDR
```

6.31 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.

6.31.1 Detailed Description

Definition of mathematical constants used by WCSLIB.

6.31.2 Macro Definition Documentation

PI

```
#define PI 3.141592653589793238462643
```

D2R

```
#define D2R PI/180.0
```

Factor $\pi/180^\circ$ to convert from degrees to radians.

R2D

```
#define R2D 180.0/PI
```

Factor $180°/\pi$ to convert from radians to degrees.

SQRT2

```
#define SQRT2 1.4142135623730950488
```

$\sqrt{2}$, used only by molset() (**MOL** projection).

SQRT2INV

```
#define SQRT2INV 1.0/SQRT2
```

$1/\sqrt{2}$, used only by qscx2s() (**QSC** projection).

UNDEFINED

```
#define UNDEFINED 987654321.0e99
```

Value used to indicate an undefined quantity (noting that NaNs cannot be used portably).

undefined

```
#define undefined(
              value ) (value == UNDEFINED)
```

Macro used to test for an undefined value.

6.32 wcsmath.h

Go to the documentation of this file.

```
/*============================================================================
  WCSLIB 8.1 - an implementation of the FITS WCS standard.
  Copyright (C) 1995-2023, Mark Calabretta

  This file is part of WCSLIB.

  WCSLIB is free software: you can redistribute it and/or modify it under the
  terms of the GNU Lesser General Public License as published by the Free
  Software Foundation, either version 3 of the License, or (at your option)
  any later version.

  WCSLIB is distributed in the hope that it will be useful, but WITHOUT ANY
  WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
  FOR A PARTICULAR PURPOSE.  See the GNU Lesser General Public License for
  more details.

  You should have received a copy of the GNU Lesser General Public License
  along with WCSLIB.  If not, see http://www.gnu.org/licenses.

  Author: Mark Calabretta, Australia Telescope National Facility, CSIRO.
  http://www.atnf.csiro.au/people/Mark.Calabretta
  $Id: wcsmath.h,v 8.1 2023/07/05 17:12:07 mcalabre Exp $
*============================================================================
```

```
*
* WCSLIB 8.1 - C routines that implement the FITS World Coordinate System
* (WCS) standard.  Refer to the README file provided with WCSLIB for an
* overview of the library.
*
*
* Summary of wcsmath.h
* --------------------
* Definition of mathematical constants used by WCSLIB.
*
*===========================================================================*/

#ifndef WCSLIB_WCSMATH
#define WCSLIB_WCSMATH

#ifdef PI
#undef PI
#endif

#ifdef D2R
#undef D2R
#endif

#ifdef R2D
#undef R2D
#endif

#ifdef SQRT2
#undef SQRT2
#endif

#ifdef SQRT2INV
#undef SQRT2INV
#endif

#define PI 3.141592653589793238462643
#define D2R PI/180.0
#define R2D 180.0/PI
#define SQRT2 1.4142135623730950488
#define SQRT2INV 1.0/SQRT2

#ifdef UNDEFINED
#undef UNDEFINED
#endif

#define UNDEFINED 987654321.0e99
#define undefined(value) (value == UNDEFINED)

#endif // WCSLIB_WCSMATH
```

6.33 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.

6.33.1 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.

6.33.2 Macro Definition Documentation

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.

6.33.3 Function Documentation

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.

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.

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.

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.

6.34 wcsprintf.h

Go to the documentation of this file.

```
/*============================================================================
  WCSLIB 8.1 - an implementation of the FITS WCS standard.
  Copyright (C) 1995-2023, Mark Calabretta

  This file is part of WCSLIB.

  WCSLIB is free software: you can redistribute it and/or modify it under the
  terms of the GNU Lesser General Public License as published by the Free
  Software Foundation, either version 3 of the License, or (at your option)
  any later version.

  WCSLIB is distributed in the hope that it will be useful, but WITHOUT ANY
  WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
  FOR A PARTICULAR PURPOSE.  See the GNU Lesser General Public License for
  more details.

  You should have received a copy of the GNU Lesser General Public License
  along with WCSLIB.  If not, see http://www.gnu.org/licenses.

  Author: Mark Calabretta, Australia Telescope National Facility, CSIRO.
  http://www.atnf.csiro.au/people/Mark.Calabretta
  $Id: wcsprintf.h,v 8.1 2023/07/05 17:12:07 mcalabre Exp $
*=============================================================================
*
* WCSLIB 8.1 - C routines that implement the FITS World Coordinate System
* (WCS) standard.  Refer to the README file provided with WCSLIB for an
* overview of the library.
*
*
* Summary of the wcsprintf routines
* ---------------------------------
* 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.
*
*
* wcsprintf() - Print function used by WCSLIB diagnostic routines
* ---------------------------------------------------------------
* 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().
*
* Given:
*   format    char*     Format string, passed to one of the printf(3) family
*                       of stdio library functions.
*
*   ...       mixed     Argument list matching format, as per printf(3).
*
* Function return value:
*             int       Number of bytes written.
*
*
* wcsfprintf() - Print function used by WCSLIB diagnostic routines
* ----------------------------------------------------------------
* 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().
*
* Given:
*   stream    FILE*     The output stream if not overridden by a call to
*                       wcsprintf_set().
*
*   format    char*     Format string, passed to one of the printf(3) family
*                       of stdio library functions.
*
*   ...       mixed     Argument list matching format, as per printf(3).
*
* Function return value:
*             int       Number of bytes written.
*
*
* wcsprintf_set() - Set output disposition for wcsprintf() and wcsfprintf()
* -------------------------------------------------------------------------
* 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().
*
* Given:
```

```
*   wcsout    FILE*     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().
*
* Function return value:
*             int       Status return value:
*                         0: Success.
*
*
* wcsprintf_buf() - Get the address of the internal string buffer
* ---------------------------------------------------------------
* wcsprintf_buf() returns the address of the internal string buffer created
* when wcsprintf_set() is invoked with its FILE* argument set to zero.
*
* Function return value:
*             const char *
*                       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.
*
*
* WCSPRINTF_PTR() macro - Print addresses in a consistent way
* -----------------------------------------------------------
* 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.
*
*===========================================================================*/

#ifndef WCSLIB_WCSPRINTF
#define WCSLIB_WCSPRINTF

#include <inttypes.h>
#include <stdio.h>

#ifdef __cplusplus
extern "C" {
#endif

#define WCSPRINTF_PTR(str1, ptr, str2) \
  if (ptr) { \
    wcsprintf("%s%#" PRIxPTR "%s", (str1), (uintptr_t)(ptr), (str2)); \
  } else { \
    wcsprintf("%s0x0%s", (str1), (str2)); \
  }

int wcsprintf_set(FILE *wcsout);
int wcsprintf(const char *format, ...);
int wcsfprintf(FILE *stream, const char *format, ...);
const char *wcsprintf_buf(void);

#ifdef __cplusplus
}
#endif

#endif // WCSLIB_WCSPRINTF
```

6.35 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.

6.35.1 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.

6.35.2 Macro Definition Documentation

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 $|v| == 1$.

6.35.3 Function Documentation

cosd()

```
double cosd (
            double angle )
```

cosd() returns the cosine of an angle given in degrees.

Parameters

in	*angle*	[deg].

Returns

Cosine of the angle.

sind()

```
double sind (
            double angle )
```

sind() returns the sine of an angle given in degrees.

Parameters

in	*angle*	[deg].

Returns

Sine of the angle.

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

tand()

```
double tand (
            double angle )
```

tand() returns the tangent of an angle given in degrees.

Parameters

in	*angle*	[deg].

Returns

Tangent of the angle.

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].

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].

atand()

```
double atand (
            double s )
```

atand() returns the inverse tangent in degrees.

Parameters

in	*s*	

Returns

Inverse tangent of s [deg].

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].

6.36 wcstrig.h

Go to the documentation of this file.

```
/*============================================================================
  WCSLIB 8.1 - an implementation of the FITS WCS standard.
  Copyright (C) 1995-2023, Mark Calabretta

  This file is part of WCSLIB.

  WCSLIB is free software: you can redistribute it and/or modify it under the
  terms of the GNU Lesser General Public License as published by the Free
  Software Foundation, either version 3 of the License, or (at your option)
  any later version.

  WCSLIB is distributed in the hope that it will be useful, but WITHOUT ANY
  WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
  FOR A PARTICULAR PURPOSE.  See the GNU Lesser General Public License for
  more details.

  You should have received a copy of the GNU Lesser General Public License
  along with WCSLIB.  If not, see http://www.gnu.org/licenses.

  Author: Mark Calabretta, Australia Telescope National Facility, CSIRO.
  http://www.atnf.csiro.au/people/Mark.Calabretta
  $Id: wcstrig.h,v 8.1 2023/07/05 17:12:07 mcalabre Exp $
*=============================================================================
*
* WCSLIB 8.1 - C routines that implement the FITS World Coordinate System
* (WCS) standard.  Refer to the README file provided with WCSLIB for an
* overview of the library.
*
*
* Summary of the wcstrig routines
* -------------------------------
* 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 degrees 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 degrees.
*
* However, wcstrig.h also provides the choice of using preprocessor macro
* implementations of the trigd functions that don't test for multiples of
* 90 degrees (compile with -DWCSTRIG_MACRO).  These are typically 20% faster
* but may lead to problems near the poles.
*
*
* cosd() - Cosine of an angle in degrees
* --------------------------------------
* cosd() returns the cosine of an angle given in degrees.
*
* Given:
*   angle     double    [deg].
*
* Function return value:
*             double    Cosine of the angle.
*
*
* sind() - Sine of an angle in degrees
* ------------------------------------
* sind() returns the sine of an angle given in degrees.
*
* Given:
*   angle     double    [deg].
*
* Function return value:
*             double    Sine of the angle.
*
*
* sincosd() - Sine and cosine of an angle in degrees
* --------------------------------------------------
* sincosd() returns the sine and cosine of an angle given in degrees.
*
* Given:
*   angle     double    [deg].
*
```

```
* Returned:
*   sin       *double   Sine of the angle.
*
*   cos       *double   Cosine of the angle.
*
* Function return value:
*             void
*
*
* tand() - Tangent of an angle in degrees
* ---------------------------------------
* tand() returns the tangent of an angle given in degrees.
*
* Given:
*   angle     double    [deg].
*
* Function return value:
*             double    Tangent of the angle.
*
*
* acosd() - Inverse cosine, returning angle in degrees
* ----------------------------------------------------
* acosd() returns the inverse cosine in degrees.
*
* Given:
*   x         double    in the range [-1,1].
*
* Function return value:
*             double    Inverse cosine of x [deg].
*
*
* asind() - Inverse sine, returning angle in degrees
* --------------------------------------------------
* asind() returns the inverse sine in degrees.
*
* Given:
*   y         double    in the range [-1,1].
*
* Function return value:
*             double    Inverse sine of y [deg].
*
*
* atand() - Inverse tangent, returning angle in degrees
* -----------------------------------------------------
* atand() returns the inverse tangent in degrees.
*
* Given:
*   s         double
*
* Function return value:
*             double    Inverse tangent of s [deg].
*
*
* atan2d() - Polar angle of (x,y), in degrees
* -------------------------------------------
* atan2d() returns the polar angle, beta, in degrees, of polar coordinates
* (rho,beta) corresponding to Cartesian coordinates (x,y).  It is equivalent
* to the arg(x,y) function of WCS Paper II, though with transposed arguments.
*
* Given:
*   y         double    Cartesian y-coordinate.
*
*   x         double    Cartesian x-coordinate.
*
* Function return value:
*             double    Polar angle of (x,y) [deg].
*
*===========================================================================*/

#ifndef WCSLIB_WCSTRIG
#define WCSLIB_WCSTRIG

#include <math.h>

#include "wcsconfig.h"

#ifdef HAVE_SINCOS
  void sincos(double angle, double *sin, double *cos);
#endif

#ifdef __cplusplus
extern "C" {
#endif


#ifdef WCSTRIG_MACRO

```

```
// Macro implementation of the trigd functions.
#include "wcsmath.h"

#define cosd(X) cos((X)*D2R)
#define sind(X) sin((X)*D2R)
#define tand(X) tan((X)*D2R)
#define acosd(X) acos(X)*R2D
#define asind(X) asin(X)*R2D
#define atand(X) atan(X)*R2D
#define atan2d(Y,X) atan2(Y,X)*R2D
#ifdef HAVE_SINCOS
  #define sincosd(X,S,C) sincos((X)*D2R,(S),(C))
#else
  #define sincosd(X,S,C) *(S) = sin((X)*D2R); *(C) = cos((X)*D2R);
#endif

#else

// Use WCSLIB wrappers or native trigd functions.

double cosd(double angle);
double sind(double angle);
void sincosd(double angle, double *sin, double *cos);
double tand(double angle);
double acosd(double x);
double asind(double y);
double atand(double s);
double atan2d(double y, double x);

// Domain tolerance for asin() and acos() functions.
#define WCSTRIG_TOL 1e-10

#endif // WCSTRIG_MACRO


#ifdef __cplusplus
}
#endif

#endif // WCSLIB_WCSTRIG
```

6.37 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.

6.37.1 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.

6.37.2 Macro Definition Documentation

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

6.37.3 Enumeration Type Documentation

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	

6.37.4 Function Documentation

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.

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.

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()

Parameters

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.

wcsunits()

```
int wcsunits (
            const char have[],
            const char want[],
            double * scale,
            double * offset,
            double * power )
```

wcsutrn()

```
int wcsutrn (
            int ctrl,
            char unitstr[] )
```

wcsulex()

```
int wcsulex (
            const char unitstr[],
            int * func,
            double * scale,
            double units[WCSUNITS_NTYPE] )
```

6.37.5 Variable Documentation

wcsunits_errmsg

```
const char * wcsunits_errmsg[]  [extern]
```

Error messages to match the status value returned from each function.

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

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.

6.38 wcsunits.h

Go to the documentation of this file.

```
/*============================================================================
  WCSLIB 8.1 - an implementation of the FITS WCS standard.
  Copyright (C) 1995-2023, Mark Calabretta

  This file is part of WCSLIB.

  WCSLIB is free software: you can redistribute it and/or modify it under the
  terms of the GNU Lesser General Public License as published by the Free
  Software Foundation, either version 3 of the License, or (at your option)
  any later version.

  WCSLIB is distributed in the hope that it will be useful, but WITHOUT ANY
  WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
  FOR A PARTICULAR PURPOSE.  See the GNU Lesser General Public License for
  more details.

  You should have received a copy of the GNU Lesser General Public License
  along with WCSLIB.  If not, see http://www.gnu.org/licenses.
```

```

  Author: Mark Calabretta, Australia Telescope National Facility, CSIRO.
  http://www.atnf.csiro.au/people/Mark.Calabretta
  $Id: wcsunits.h,v 8.1 2023/07/05 17:12:07 mcalabre Exp $
*=============================================================================
*
* WCSLIB 8.1 - C routines that implement the FITS World Coordinate System
* (WCS) standard.  Refer to the README file provided with WCSLIB for an
* overview of the library.
*
*
* Summary of the wcsunits routines
* --------------------------------
* 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://fits.gsfc.nasa.gov
*
* 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.
*
*
* wcsunitse() - FITS units specification conversion
* -------------------------------------------------
* wcsunitse() derives the conversion from one system of units to another.
*
* A deprecated form of this function, wcsunits(), lacks the wcserr**
* parameter.
*
* Given:
*   have      const char []
*                       FITS units specification to convert from (null-
*                       terminated), with or without surrounding square
*                       brackets (for inline specifications); text following
*                       the closing bracket is ignored.
*
*   want      const char []
*                       FITS units specification to convert to (null-
*                       terminated), with or without surrounding square
*                       brackets (for inline specifications); text following
*                       the closing bracket is ignored.
*
* Returned:
*   scale,
*   offset,
*   power     double*   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;
*
*   err       struct wcserr **
*                       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.
*
* Function return value:
*             int       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.
*
*
* wcsutrne() - Translation of non-standard unit specifications
* ------------------------------------------------------------
* 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.
*
* Given:
*   ctrl      int       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.
*
* Given and returned:
*   unitstr   char []   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.
*
*   err       struct wcserr **
*                       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.
*
* Function return value:
*             int       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.
*
*
* wcsulexe() - FITS units specification parser
* --------------------------------------------
* 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.
*
* Given:
*   unitstr   const char []
*                       Null-terminated character array containing the units
*                       specification, with or without surrounding square
*                       brackets (for inline specifications); text following
*                       the closing bracket is ignored.
*
* Returned:
*   func      int*      Special function type, see note 4:
*                         0: None
*                         1: log()   ...base 10
*                         2: ln()    ...base e
*                         3: exp()
*
*   scale     double*   Scale factor for the unit specification; multiply a
*                       value expressed in the given units by this factor to
*                       convert it to canonical units.
*
*   units     double[WCSUNITS_NTYPE]
*                       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.
*
*   err       struct wcserr **
*                       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.
*
* Function return value:
*             int       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 1e6 will be returned.
*
*
* Global variable: const char *wcsunits_errmsg[] - Status return messages
* -----------------------------------------------------------------------
* Error messages to match the status value returned from each function.
*
*
* Global variable: const char *wcsunits_types[] - Names of physical quantities
* ----------------------------------------------------------------------------
* 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
*
*
* Global variable: const char *wcsunits_units[] - Names of units
* --------------------------------------------------------------
* 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.
*===========================================================================*/

#ifndef WCSLIB_WCSUNITS
#define WCSLIB_WCSUNITS

#include "wcserr.h"

#ifdef __cplusplus
extern "C" {
#endif


extern const char *wcsunits_errmsg[];

enum wcsunits_errmsg_enum {
  UNITSERR_SUCCESS            =  0,     // Success.
  UNITSERR_BAD_NUM_MULTIPLIER =  1,  // Invalid numeric multiplier.
  UNITSERR_DANGLING_BINOP     =  2,     // Dangling binary operator.
  UNITSERR_BAD_INITIAL_SYMBOL =  3,  // Invalid symbol in INITIAL context.
  UNITSERR_FUNCTION_CONTEXT   =  4,    // Function in invalid context.
  UNITSERR_BAD_EXPON_SYMBOL   =  5,    // Invalid symbol in EXPON context.
  UNITSERR_UNBAL_BRACKET      =  6,       // Unbalanced bracket.
  UNITSERR_UNBAL_PAREN        =  7, // Unbalanced parenthesis.
  UNITSERR_CONSEC_BINOPS      =  8,       // Consecutive binary operators.
  UNITSERR_PARSER_ERROR       =  9,        // Internal parser error.
  UNITSERR_BAD_UNIT_SPEC      = 10,       // Non-conformant unit specifications.
  UNITSERR_BAD_FUNCS          = 11,   // Non-conformant functions.
  UNITSERR_UNSAFE_TRANS       = 12 // Potentially unsafe translation.
};

extern const char *wcsunits_types[];
extern const char *wcsunits_units[];
```

```

#define WCSUNITS_PLANE_ANGLE 0
#define WCSUNITS_SOLID_ANGLE 1
#define WCSUNITS_CHARGE      2
#define WCSUNITS_MOLE        3
#define WCSUNITS_TEMPERATURE 4
#define WCSUNITS_LUMINTEN    5
#define WCSUNITS_MASS        6
#define WCSUNITS_LENGTH      7
#define WCSUNITS_TIME        8
#define WCSUNITS_BEAM        9
#define WCSUNITS_BIN        10
#define WCSUNITS_BIT        11
#define WCSUNITS_COUNT      12
#define WCSUNITS_MAGNITUDE  13
#define WCSUNITS_PIXEL      14
#define WCSUNITS_SOLRATIO   15
#define WCSUNITS_VOXEL      16

#define WCSUNITS_NTYPE      17


int wcsunitse(const char have[], const char want[], double *scale,
              double *offset, double *power, struct wcserr **err);

int wcsutrne(int ctrl, char unitstr[], struct wcserr **err);

int wcsulexe(const char unitstr[], int *func, double *scale,
             double units[WCSUNITS_NTYPE], struct wcserr **err);

// Deprecated.
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]);

#ifdef __cplusplus
}
#endif

#endif // WCSLIB_WCSUNITS
```

6.39 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.

6.39.1 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!

6.39.2 Function Documentation

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

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

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

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

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.

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.

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.

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.

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.

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.

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.

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

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

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

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.

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.

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.

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.

6.40 wcsutil.h

Go to the documentation of this file.

```
/*============================================================================
  WCSLIB 8.1 - an implementation of the FITS WCS standard.
  Copyright (C) 1995-2023, Mark Calabretta

  This file is part of WCSLIB.

  WCSLIB is free software: you can redistribute it and/or modify it under the
  terms of the GNU Lesser General Public License as published by the Free
  Software Foundation, either version 3 of the License, or (at your option)
  any later version.

  WCSLIB is distributed in the hope that it will be useful, but WITHOUT ANY
  WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
  FOR A PARTICULAR PURPOSE.  See the GNU Lesser General Public License for
  more details.

  You should have received a copy of the GNU Lesser General Public License
  along with WCSLIB.  If not, see http://www.gnu.org/licenses.

  Author: Mark Calabretta, Australia Telescope National Facility, CSIRO.
  http://www.atnf.csiro.au/people/Mark.Calabretta
  $Id: wcsutil.h,v 8.1 2023/07/05 17:12:07 mcalabre Exp $
*=============================================================================
*
* WCSLIB 8.1 - C routines that implement the FITS World Coordinate System
* (WCS) standard.  Refer to the README file provided with WCSLIB for an
* overview of the library.
*
*
* Summary of the wcsutil routines
* -------------------------------
* 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!
*
*
* wcsdealloc() - free memory allocated by WCSLIB functions
* --------------------------------------------------------
* 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.
*
* Given and returned:
*   ptr       void*     Address of the allocated memory.
*
* Function return value:
*             void
*
*
* wcsutil_strcvt() - Copy character string with padding
* -----------------------------------------------------
* 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.
*
* Given:
*   n         int       Maximum number of characters to copy.
*
*   c         char      Substitute character, either NULL or blank (anything
*                       other than NULL).
*
*   nt        int       If true, then dst is of length n+1, with the last
*                       character always set to NULL.
*
*   src       char[]    Character string to be copied.  If null-terminated,
*                       then need not be of length n, otherwise it must be.
*
* Returned:
*   dst       char[]    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.
*
* Function return value:
*             void
*
*
* wcsutil_blank_fill() - Fill a character string with blanks
* ----------------------------------------------------------
* INTERNAL USE ONLY.
*
* wcsutil_blank_fill() pads a character sub-string with blanks starting with
* the terminating NULL character (if any).
*
* Given:
*   n         int       Length of the sub-string.
*
* Given and returned:
*   c         char[]    The character sub-string, which will not be
*                       null-terminated on return.
*
* Function return value:
*             void
*
*
* wcsutil_null_fill() - Fill a character string with NULLs
* --------------------------------------------------------
* 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.
*
* Given:
*   n         int       Number of characters.
*
* Given and returned:
*   c         char[]    The character (sub-)string.
*
* Function return value:
*             void
*
*
* wcsutil_all_ival() - Test if all elements an int array have a given value
* -------------------------------------------------------------------------
* INTERNAL USE ONLY.
*
* wcsutil_all_ival() tests whether all elements of an array of type int all
* have the specified value.
*
* Given:
*   nelem     int       The length of the array.
*
*   ival      int       Value to be tested.
*
*   iarr      const int[]
```

```
*                         Pointer to the first element of the array.
*
* Function return value:
*             int       Status return value:
*                           0: Not all equal.
*                           1: All equal.
*
*
* wcsutil_all_dval() - Test if all elements a double array have a given value
* ---------------------------------------------------------------------------
* INTERNAL USE ONLY.
*
* wcsutil_all_dval() tests whether all elements of an array of type double all
* have the specified value.
*
* Given:
*   nelem     int       The length of the array.
*
*   dval      int       Value to be tested.
*
*   darr      const double[]
*                       Pointer to the first element of the array.
*
* Function return value:
*             int       Status return value:
*                         0: Not all equal.
*                         1: All equal.
*
*
* wcsutil_all_sval() - Test if all elements a string array have a given value
* ---------------------------------------------------------------------------
* INTERNAL USE ONLY.
*
* wcsutil_all_sval() tests whether the elements of an array of type
* char (*)[72] all have the specified value.
*
* Given:
*   nelem     int       The length of the array.
*
*   sval      const char *
*                       String to be tested.
*
*   sarr      const char (*)[72]
*                       Pointer to the first element of the array.
*
* Function return value:
*             int       Status return value:
*                         0: Not all equal.
*                         1: All equal.
*
*
* wcsutil_allEq() - Test for equality of a particular vector element
* ------------------------------------------------------------------
* INTERNAL USE ONLY.
*
* wcsutil_allEq() tests for equality of a particular element in a set of
* vectors.
*
* Given:
*   nvec      int       The number of vectors.
*
*   nelem     int       The length of each vector.
*
*   first     const double*
*                       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];
*
* Function return value:
*             int       Status return value:
*                         0: Not all equal.
*                         1: All equal.
*
*
* wcsutil_dblEq() - Test for equality of two arrays of type double
* ----------------------------------------------------------------
* INTERNAL USE ONLY.
*
* wcsutil_dblEq() tests for equality of two double-precision arrays.
```

```
*
* Given:
*   nelem     int       The number of elements in each array.
*
*   tol       double    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.
*
*   arr1      const double*
*                       The first array.
*
*   arr2      const double*
*                       The second array
*
* Function return value:
*             int       Status return value:
*                         0: Not equal.
*                         1: Equal.
*
*
* wcsutil_intEq() - Test for equality of two arrays of type int
* -------------------------------------------------------------
* INTERNAL USE ONLY.
*
* wcsutil_intEq() tests for equality of two int arrays.
*
* Given:
*   nelem     int       The number of elements in each array.
*
*   arr1      const int*
*                       The first array.
*
*   arr2      const int*
*                       The second array
*
* Function return value:
*             int       Status return value:
*                         0: Not equal.
*                         1: Equal.
*
*
* wcsutil_strEq() - Test for equality of two string arrays
* --------------------------------------------------------
* INTERNAL USE ONLY.
*
* wcsutil_strEq() tests for equality of two string arrays.
*
* Given:
*   nelem     int       The number of elements in each array.
*
*   arr1      const char**
*                       The first array.
*
*   arr2      const char**
*                       The second array
*
* Function return value:
*             int       Status return value:
*                         0: Not equal.
*                         1: Equal.
*
*
* wcsutil_setAll() - Set a particular vector element
* --------------------------------------------------
* INTERNAL USE ONLY.
*
* wcsutil_setAll() sets the value of a particular element in a set of vectors
* of type double.
*
* Given:
*   nvec      int       The number of vectors.
*
*   nelem     int       The length of each vector.
*
* Given and returned:
*   first     double*   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];
```

```
*
* Function return value:
*             void
*
*
* wcsutil_setAli() - Set a particular vector element
* --------------------------------------------------
* INTERNAL USE ONLY.
*
* wcsutil_setAli() sets the value of a particular element in a set of vectors
* of type int.
*
* Given:
*   nvec      int       The number of vectors.
*
*   nelem     int       The length of each vector.
*
* Given and returned:
*   first     int*      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];
*
* Function return value:
*             void
*
*
* wcsutil_setBit() - Set bits in selected elements of an array
* ------------------------------------------------------------
* INTERNAL USE ONLY.
*
* wcsutil_setBit() sets bits in selected elements of an array.
*
* Given:
*   nelem     int       Number of elements in the array.
*
*   sel       const int*
*                       Address of a selection array of length nelem.  May
*                       be specified as the null pointer in which case all
*                       elements are selected.
*
*   bits      int       Bit mask.
*
* Given and returned:
*   array     int*      Address of the array of length nelem.
*
* Function return value:
*             void
*
*
* wcsutil_fptr2str() - Translate pointer-to-function to string
* ------------------------------------------------------------
* 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-pointer
*
* Given:
*   fptr      void(*)() Pointer to function.
*
* Returned:
*   hext      char[19]  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'.
*
* Function return value:
*             char *    The address of hext.
*
*
* wcsutil_double2str() - Translate double to string ignoring the locale
* ---------------------------------------------------------------------
* 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.
```

```
*
* Returned:
*   buf       char *    The buffer to write the string into.
*
* Given:
*   format    char *    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.
*
*   value     double    The value to convert to a string.
*
*
* wcsutil_str2double() - Translate string to a double, ignoring the locale
* -------------------------------------------------------------------------
* 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.
*
* Given:
*   buf       char *    The string containing the value
*
* Returned:
*   value     double *  The double value parsed from the string.
*
*
* wcsutil_str2double2() - Translate string to doubles, ignoring the locale
* -------------------------------------------------------------------------
* 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.
*
* Given:
*   buf       char *    The string containing the value
*
* Returned:
*   value     double[2] The double value, split into integer and fractional
*                       parts, parsed from the string.
*
*===========================================================================*/

#ifndef WCSLIB_WCSUTIL
#define WCSLIB_WCSUTIL

#ifdef __cplusplus
extern "C" {
#endif

void wcsdealloc(void *ptr);

void wcsutil_strcvt(int n, char c, int nt, const char src[], char dst[]);

void wcsutil_blank_fill(int n, char c[]);
void wcsutil_null_fill (int n, char c[]);

int  wcsutil_all_ival(int nelem, int ival, const int iarr[]);
int  wcsutil_all_dval(int nelem, double dval, const double darr[]);
int  wcsutil_all_sval(int nelem, const char *sval, const char (*sarr)[72]);
int  wcsutil_allEq (int nvec, int nelem, const double *first);

int  wcsutil_dblEq(int nelem, double tol, const double *arr1,
                   const double *arr2);
int  wcsutil_intEq(int nelem, const int *arr1, const int *arr2);
int  wcsutil_strEq(int nelem, char (*arr1)[72], char (*arr2)[72]);
void wcsutil_setAll(int nvec, int nelem, double *first);
void wcsutil_setAli(int nvec, int nelem, int *first);
void wcsutil_setBit(int nelem, const int *sel, int bits, int *array);
char *wcsutil_fptr2str(void (*fptr)(void), char hext[19]);
void wcsutil_double2str(char *buf, const char *format, double value);
int  wcsutil_str2double(const char *buf, double *value);
int  wcsutil_str2double2(const char *buf, double *value);

#ifdef __cplusplus
}
#endif

#endif // WCSLIB_WCSUTIL
```

6.41 wtbarr.h File Reference

Data Structures

- struct wtbarr

    Extraction of coordinate lookup tables from BINTABLE.

6.41.1 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.

6.42 wtbarr.h

Go to the documentation of this file.

```
/*============================================================================
  WCSLIB 8.1 - an implementation of the FITS WCS standard.
  Copyright (C) 1995-2023, Mark Calabretta

  This file is part of WCSLIB.

  WCSLIB is free software: you can redistribute it and/or modify it under the
  terms of the GNU Lesser General Public License as published by the Free
  Software Foundation, either version 3 of the License, or (at your option)
  any later version.

  WCSLIB is distributed in the hope that it will be useful, but WITHOUT ANY
  WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
  FOR A PARTICULAR PURPOSE.  See the GNU Lesser General Public License for
  more details.

  You should have received a copy of the GNU Lesser General Public License
  along with WCSLIB.  If not, see http://www.gnu.org/licenses.

  Author: Mark Calabretta, Australia Telescope National Facility, CSIRO.
  http://www.atnf.csiro.au/people/Mark.Calabretta
  $Id: wtbarr.h,v 8.1 2023/07/05 17:12:07 mcalabre Exp $
*=============================================================================
*
* WCSLIB 8.1 - C routines that implement the FITS World Coordinate System
* (WCS) standard.  Refer to the README file provided with WCSLIB for an
* overview of the library.
*
*
* Summary of the wtbarr struct
* ----------------------------
* 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.
*
*
* wtbarr struct - Extraction of coordinate lookup tables from BINTABLE
* --------------------------------------------------------------------
* 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.
*
*   int i
*     (Given) Image axis number.
*
*   int m
*     (Given) wcstab array axis number for index vectors.
*
*   int kind
*     (Given) Character identifying the wcstab array type:
*       - c: coordinate array,
```

```
*      - i: index vector.
*
*   char extnam[72]
*     (Given) EXTNAME identifying the binary table extension.
*
*   int extver
*     (Given) EXTVER identifying the binary table extension.
*
*   int extlev
*     (Given) EXTLEV identifying the binary table extension.
*
*   char ttype[72]
*     (Given) TTYPEn identifying the column of the binary table that contains
*     the wcstab array.
*
*   long row
*     (Given) Table row number.
*
*   int ndim
*     (Given) Expected dimensionality of the wcstab array.
*
*   int *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.
*
*   double **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.
*
*===========================================================================*/

#ifndef WCSLIB_WTBARR
#define WCSLIB_WTBARR

#ifdef __cplusplus
extern "C" {
#define wtbarr wtbarr_s         // See prologue above.
#endif
                                // For extracting wcstab arrays.  Matches
                                // the wtbarr typedef defined in CFITSIO
                                // header fitsio.h.
struct wtbarr {
  int  i;                       // Image axis number.
  int  m;                       // Array axis number for index vectors.
  int  kind;                    // wcstab array type.
  char extnam[72];              // EXTNAME of binary table extension.
  int  extver;                  // EXTVER  of binary table extension.
  int  extlev;                  // EXTLEV  of binary table extension.
  char ttype[72];               // TTYPEn of column containing the array.
  long row;                     // Table row number.
  int  ndim;                    // Expected wcstab array dimensionality.
  int  *dimlen;                 // Where to write the array axis lengths.
  double **arrayp;              // Where to write the address of the array
                                // allocated to store the wcstab array.
};

#ifdef __cplusplus
#undef wtbarr
}
#endif

#endif // WCSLIB_WTBARR
```

6.43 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"
```

6.43.1 Detailed Description

This header file is provided purely for convenience. Use it to include all of the separate WCSLIB headers.

6.44 wcslib.h

Go to the documentation of this file.

```
/*============================================================================
  WCSLIB 8.1 - an implementation of the FITS WCS standard.
  Copyright (C) 1995-2023, Mark Calabretta

  This file is part of WCSLIB.

  WCSLIB is free software: you can redistribute it and/or modify it under the
  terms of the GNU Lesser General Public License as published by the Free
  Software Foundation, either version 3 of the License, or (at your option)
  any later version.

  WCSLIB is distributed in the hope that it will be useful, but WITHOUT ANY
  WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
  FOR A PARTICULAR PURPOSE.  See the GNU Lesser General Public License for
  more details.

  You should have received a copy of the GNU Lesser General Public License
  along with WCSLIB.  If not, see http://www.gnu.org/licenses.

  Author: Mark Calabretta, Australia Telescope National Facility, CSIRO.
  http://www.atnf.csiro.au/people/Mark.Calabretta
  $Id: wcslib.h,v 8.1 2023/07/05 17:12:07 mcalabre Exp $
*=============================================================================
*
* WCSLIB 8.1 - C routines that implement the FITS World Coordinate System
* (WCS) standard.  Refer to the README file provided with WCSLIB for an
* overview of the library.
*
* Summary of wcslib.h
* -------------------
* This header file is provided purely for convenience.  Use it to include all
* of the separate WCSLIB headers.
*
*===========================================================================*/

#ifndef WCSLIB_WCSLIB
#define WCSLIB_WCSLIB

#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"

#endif // WCSLIB_WCSLIB
  wcserr_enable(1);
  wcsprintf_set(stderr);

  ...

  if (wcsset(&wcs) {
    wcsperr(&wcs);
    return wcs.err->status;
  }
@endverbatim
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 @c twcs test program, which tests wcserr among other things, produces a
traceback similar to this:
@verbatim
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.
@endverbatim

Each of the @ref structs "structs" in @ref overview "WCSLIB" includes a
pointer, called @a err, to a wcserr struct.  When an error occurs, a struct is
allocated and error information stored in it.  The wcserr pointers and the
@ref memory "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).
*/


```

Index