WCSLIB 4.4 Reference Manual

Generated by Doxygen 1.5.1

Mon Sep 14 17:03:55 2009

Contents

1 WCSLIB 4.4 and PGSBOX 4.4

1.1 Contents

1.2 Copyright

```
Correspondence concerning WCSLIB may be directed to:
  Internet email: mcalabre@atnf.csiro.au
  Postal address: Dr. Mark Calabretta
                  Australia Telescope National Facility, CSIRO
                  PO Box 76
                  Epping NSW 1710
                  AUSTRALIA
```


2 WCSLIB 4.4 Data Structure Index

2.1 WCSLIB 4.4 Data Structures

Here are the data structures with brief descriptions:

3 WCSLIB 4.4 File Index

3.1 WCSLIB 4.4 File List

Here is a list of all files with brief descriptions:

4 WCSLIB 4.4 Page Index

4.1 WCSLIB 4.4 Related Pages

Here is a list of all related documentation pages:

5 WCSLIB 4.4 Data Structure Documentation

5.1 celprm Struct Reference

Celestial transformation parameters.

```
#include <cel.h>
```

Data Fields

- int flag
- int offset
- double phi0
- double theta0
- double ref [4]
- prjprm prj

- double euler [5]
- int latpreq
- int isolat

5.1.1 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.

5.1.2 Field Documentation

5.1.2.1 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.

5.1.2.2 int celprm::offset

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

5.1.2.3 double celprm::phi0

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

5.1.2.4 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.

5.1.2.5 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).

LONPOLE`a` 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.

5.1.2.6 struct prjprm celprm::prj

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

5.1.2.7 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.

5.1.2.8 int celprm::latpreq

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

- 0: Not required, $\theta_{\rm p}$ (== $\delta_{\rm p}$) was determined uniquely by the **CRVAL**`ia` and **LONPOLE**`a` keywords.
- 1: Required to select between two valid solutions of $\theta_{\rm p}$.
- 2: $\theta_{\rm p}$ was specified solely by **LATPOLE**`a`.

5.1.2.9 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.

5.2 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]

5.2.1 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.

5.2.2 Field Documentation

5.2.2.1 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.

5.2.2.2 int fitskey::keyid

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

5.2.2.3 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.

5.2.2.4 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.

5.2.2.5 int fitskey::type

(*Returned*) Keyvalue data type:

- 0: No keyvalue.
- 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.

5.2.2.6 int fitskey::padding

(An unused variable inserted for alignment purposes only.)

5.2.2.7 int fitskey::i

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

5.2.2.8 int64 fitskey::k

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

5.2.2.9 int fitskey::l

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

5.2.2.10 double fitskey::f

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

5.2.2.11 double fitskey::c

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

5.2.2.12 char fitskey::s

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

5.2.2.13 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.

5.2.2.14 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.

5.2.2.15 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).

5.3 fitskeyid Struct Reference

Keyword indexing.

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

Data Fields

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

5.3.1 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().

5.3.2 Field Documentation

5.3.2.1 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.

5.3.2.2 int fitskeyid::count

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

5.3.2.3 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.

5.4 linprm Struct Reference

Linear transformation parameters.

```
#include <lin.h>
```

Data Fields

- int flag
- int naxis
- double ∗ crpix
- double ∗ pc
- double ∗ cdelt
- double ∗ piximg
- double ∗ imgpix
- int unity
- int i_naxis
- int m_flag
- int m_naxis
- double ∗ m_crpix
- double ∗ m_pc
- double ∗ m_cdelt

5.4.1 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*).

5.4.2 Field Documentation

5.4.2.1 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.

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 linini() 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.

5.4.2.2 int linprm::naxis

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

If linini() 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.

5.4.2.3 double ∗ linprm::crpix

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

5.4.2.4 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.

5.4.2.5 double ∗ linprm::cdelt

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

5.4.2.6 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.

5.4.2.7 double * linprm::imgpix

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

5.4.2.8 int linprm::unity

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

5.4.2.9 int linprm::i_naxis

(For internal use only.)

5.4.2.10 int linprm::m_flag

(For internal use only.)

5.4.2.11 int linprm::m_naxis

(For internal use only.)

5.4.2.12 double * linprm::m_crpix

(For internal use only.)

5.4.2.13 double * linprm::m_pc

(For internal use only.)

5.4.2.14 double * linprm::m_cdelt

(For internal use only.)

5.5 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
- double w [10]
- int n
- int padding
- int(* prjx2s)(PRJX2S_ARGS)
- int(* prjs2x)(PRJS2X_ARGS)

5.5.1 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.

5.5.2 Field Documentation

5.5.2.1 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.

5.5.2.2 char prjprm::code

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

5.5.2.3 double prjprm::r0

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

5.5.2.4 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.

5.5.2.5 double prjprm::phi0

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

5.5.2.6 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.

5.5.2.7 int prjprm::bounds

(*Given*) Controls strict bounds checking for the **AZP**, **SZP**, **TAN**, **SIN**, **ZPN**, and **COP** projections; set to zero to disable checking.

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

5.5.2.8 char prjprm::name

(*Returned*) Long name of the projection.

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

5.5.2.9 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.

5.5.2.10 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.

5.5.2.11 int prjprm::simplezen

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

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

5.5.2.12 int prjprm::equiareal

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

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

5.5.2.13 int prjprm::conformal

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

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

5.5.2.14 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.

5.5.2.15 int prjprm::divergent

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

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

5.5.2.16 double prjprm::x0

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

5.5.2.17 double prjprm::y0

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

5.5.2.18 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.

5.5.2.19 int prjprm::n

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

5.5.2.20 int prjprm::padding

(An unused variable inserted for alignment purposes only.)

5.5.2.21 prjprm::prjx2s

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

5.5.2.22 prjprm::prjs2x

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

5.6 pscard Struct Reference

Store for **PS**i_ma keyrecords.

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

Data Fields

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

5.6.1 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.

5.6.2 Field Documentation

5.6.2.1 int pscard::i

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

5.6.2.2 int pscard::m

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

5.6.2.3 char pscard::value

(*Given*) Parameter value.

5.7 pvcard Struct Reference

Store for **PV**i_ma keyrecords.

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

Data Fields

- int i
- int m
- double value

5.7.1 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.

5.7.2 Field Documentation

5.7.2.1 int pvcard::i

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

5.7.2.2 int pvcard::m

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

5.7.2.3 double pvcard::value

(*Given*) Parameter value.

5.8 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 padding
- int(∗ spxX2P)(SPX_ARGS)
- int(∗ spxP2S)(SPX_ARGS)
- int(∗ spxS2P)(SPX_ARGS)
- int(∗ spxP2X)(SPX_ARGS)

5.8.1 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.

5.8.2 Field Documentation

5.8.2.1 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.

5.8.2.2 char spcprm::type

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

5.8.2.3 char spcprm::code

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

5.8.2.4 double spcprm::crval

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

5.8.2.5 double spcprm::restfrq

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

5.8.2.6 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.

5.8.2.7 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:

5.8.2.8 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.

5.8.2.9 int spcprm::isGrism

(*Returned*) Grism coordinates?

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

5.8.2.10 int spcprm::padding

(An unused variable inserted for alignment purposes only.)

5.8.2.11 spcprm::spxX2P

(*Returned*) The first and ...

5.8.2.12 spcprm::spxP2S

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

5.8.2.13 spcprm::spxS2P

(*Returned*) The first and ...

5.8.2.14 spcprm::spxP2X

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

5.9 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

5.9.1 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.

5.9.2 Field Documentation

5.9.2.1 double spxprm::restfrq

(*Returned*) Rest frequency [Hz].

5.9.2.2 double spxprm::restwav

(*Returned*) Rest wavelength [m].

5.9.2.3 int spxprm::wavetype

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

5.9.2.4 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.

5.9.2.5 double spxprm::freq

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

5.9.2.6 double spxprm::afrq

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

5.9.2.7 double spxprm::ener

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

5.9.2.8 double spxprm::wavn

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

5.9.2.9 double spxprm::vrad

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

5.9.2.10 double spxprm::wave

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

5.9.2.11 double spxprm::vopt

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

5.9.2.12 double spxprm::zopt

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

5.9.2.13 double spxprm::awav

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

5.9.2.14 double spxprm::velo

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

5.9.2.15 double spxprm::beta

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

5.9.2.16 double spxprm::dfreqafrq

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

5.9.2.17 double spxprm::dafrqfreq

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

5.9.2.18 double spxprm::dfreqener

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

5.9.2.19 double spxprm::denerfreq

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

5.9.2.20 double spxprm::dfreqwavn

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

5.9.2.21 double spxprm::dwavnfreq

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

5.9.2.22 double spxprm::dfreqvrad

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

5.9.2.23 double spxprm::dvradfreq

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

5.9.2.24 double spxprm::dfreqwave

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

5.9.2.25 double spxprm::dwavefreq

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

5.9.2.26 double spxprm::dfreqawav

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

5.9.2.27 double spxprm::dawavfreq

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

5.9.2.28 double spxprm::dfreqvelo

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

5.9.2.29 double spxprm::dvelofreq

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

5.9.2.30 double spxprm::dwavevopt

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

5.9.2.31 double spxprm::dvoptwave

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

5.9.2.32 double spxprm::dwavezopt

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

5.9.2.33 double spxprm::dzoptwave

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

5.9.2.34 double spxprm::dwaveawav

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

5.9.2.35 double spxprm::dawavwave

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

5.9.2.36 double spxprm::dwavevelo

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

5.9.2.37 double spxprm::dvelowave

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

5.9.2.38 double spxprm::dawavvelo

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

5.9.2.39 double spxprm::dveloawav

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

5.9.2.40 double spxprm::dvelobeta

(*Returned*) Derivative of relativistic velocity with respect to relativistic beta [m/s] (constant, $= c$, the speed of light in vacuu0), and ...

5.9.2.41 double spxprm::dbetavelo

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

5.10 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
- 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

5.10.1 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.

5.10.2 Field Documentation

5.10.2.1 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.

5.10.2.2 int tabprm::M

(Given or returned) Number of tabular coordinate axes.

If tabini() is used to initialize the linprm 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.

5.10.2.3 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 linprm 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.

5.10.2.4 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.

5.10.2.5 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.

5.10.2.6 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;
```

5.10.2.7 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.

5.10.2.8 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).

5.10.2.9 int tabprm::padding

(An unused variable inserted for alignment purposes only.)

5.10.2.10 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).

5.10.2.11 double * 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] + tabprm::delta[m].

5.10.2.12 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] + delta[m].

5.10.2.13 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.

5.10.2.14 int tabprm::m_flag

(For internal use only.)

5.10.2.15 int tabprm::m_M

(For internal use only.)

5.10.2.16 int tabprm::m_N

(For internal use only.)

5.10.2.17 int tabprm::set_M

(For internal use only.)

5.10.2.18 int tabprm::m_K

(For internal use only.)

5.10.2.19 int tabprm::m_map

(For internal use only.)

5.10.2.20 int tabprm::m_crval

(For internal use only.)

5.10.2.21 int tabprm::m_index

(For internal use only.)

5.10.2.22 int tabprm::m_indxs

(For internal use only.)

5.10.2.23 int tabprm::m_coord

(For internal use only.)

5.11 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
- pvcard * pv
- int nps
- int npsmax
- pscard * ps
- double * cd
- double * crota
- int altlin
- int padding
- char alt [4]
- int colnum
- int * colax
- char(* cname)[72]
- double * crder

- double ∗ csyer
- char dateavg [72]
- char dateobs [72]
- double equinox
- double mjdavg
- double mjdobs
- double obsgeo [3]
- char radesys [72]
- char specsys [72]
- char ssysobs [72]
- double velosys
- double zsource
- char ssyssrc [72]
- double velangl
- char wcsname [72]
- int ntab
- int nwtb
- tabprm ∗ tab
- wtbarr ∗ wtb
- int ∗ types
- char lngtyp [8]
- char lattyp [8]
- int lng
- int lat
- int spec
- int cubeface
- linprm lin
- celprm cel
- spcprm spc
- 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]
- pvcard ∗ m_pv
- pscard ∗ m_ps
- double ∗ m_cd
- double ∗ m_crota
- int ∗ m_colax
- char(∗ m_cname)[72]
- double ∗ m_crder
- double ∗ m_csyer
- tabprm ∗ m_tab
- wtbarr ∗ m_wtb

5.11.1 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*). Some of the former 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 wcsini() 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.

5.11.2 Field Documentation

5.11.2.1 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.

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

PLEASE NOTE: flag should be set to -1 when wcsini() 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.

5.11.2.2 int wcsprm::naxis

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

If wcsini() 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.

5.11.2.3 double ∗ wcsprm::crpix

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

5.11.2.4 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.

5.11.2.5 double ∗ wcsprm::cdelt

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

5.11.2.6 double ∗ wcsprm::crval

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

5.11.2.7 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.

5.11.2.8 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.

5.11.2.9 double wcsprm::lonpole

(Given and returned) The native longitude of the celestial pole, $\phi_{\rm p}$, given by **LONPOLE**a [deg] or by PV**i**_2**a** [deg] attached to the longitude axis which takes precedence if defined, and ...

5.11.2.10 double wcsprm::latpole

(Given and returned) ... the native latitude of the celestial pole, $\theta_{\rm p}$, given by **LATPOLE**a [deg] or by PV**i**_3**a** [deg] attached to the longitude axis which takes precedence if defined.

lonpole and latpole may be left to default to values set by wcsini() (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 wcsini(), they will no longer have their default values.

5.11.2.11 double wcsprm::restfrq

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

5.11.2.12 double wcsprm::restwav

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

5.11.2.13 int wcsprm::npv

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

5.11.2.14 int wcsprm::npvmax

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

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

5.11.2.15 struct pvcard * wcsprm::pv

(Given or returned) Address of the first element of an array of length npvmax of pvcard structs. Set by wcsini() if it allocates memory for pv[], otherwise it must be set by the user. See also wcsnpv().

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 `PV`**i_**`1`**a**, `PV`**i_**`2`**a**, `PV`**i_**`3`**a**, and `PV`**i_**`4`**a** for longitude axis **i** to match (ϕ_0, θ_0), the native longitude and latitude of the reference point given by **LONPOLE**`a` and **LATPOLE**`a`.

5.11.2.16 int wcsprm::nps

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

5.11.2.17 int wcsprm::npsmax

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

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

5.11.2.18 struct pscard * wcsprm::ps

(Given or returned) Address of the first element of an array of length npsmax of pscard structs. Set by wcsini() if it allocates memory for ps[], otherwise it must be set by the user. See also wcsnps().

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).

5.11.2.19 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 ...

5.11.2.20 double * wcsprm::crota

(*Given*) ... those associated with the **CROTAia** 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`.

5.11.2.21 int wcsprm::altlin

(*Given*) altlin is a bit flag that denotes which of the **PC**`i_ja`, **CD**`i_ja` and **CROTAia** 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 **CROTAia**) 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 **CROTAia** which are to be ignored.

- Bit 2: **CROTAia** is present.

  In the AIPS convention, **CROTAia** 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 **CROTAia** keywords are ignored.

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

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

CD`i_ja` and **CROTAia** 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 **CROTAia** 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, wcsset() resets wcsprm::cdelt to unity if **CD**`i_ja` is present (and no **PC**`i_ja`).

If **CROTAia** 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.

5.11.2.22 int wcsprm::padding

(An unused variable inserted for alignment purposes only.)

5.11.2.23 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.

5.11.2.24 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.

5.11.2.25 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.

5.11.2.26 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.

5.11.2.27 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`.

5.11.2.28 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`.

5.11.2.29 char wcsprm::dateavg

(Given, auxiliary) The date of a representative mid-point of the observation in ISO format, *yyyy-mm-dd***T***hh:mm:ss*.

5.11.2.30 char wcsprm::dateobs

(Given, auxiliary) The date of the start of the observation unless otherwise explained in the comment field of the **DATE-OBS** keyword, in ISO format, *yyyy-mm-dd***T***hh:mm:ss*.

5.11.2.31 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.

5.11.2.32 double wcsprm::mjdavg

(Given, auxiliary) Modified Julian Date (MJD = JD - 2400000.5), **MJD-AVG**, corresponding to **DATE-AVG**.

5.11.2.33 double wcsprm::mjdobs

(Given, auxiliary) Modified Julian Date (MJD = JD - 2400000.5), **MJD-OBS**, corresponding to **DATE-OBS**.

5.11.2.34 double wcsprm::obsgeo

(Given, auxiliary) Location of the observer in a standard terrestrial reference frame, **OBSGEO-X**, **OBSGEO-Y**, **OBSGEO-Z** [m].

5.11.2.35 char wcsprm::radesys

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

5.11.2.36 char wcsprm::specsys

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

5.11.2.37 char wcsprm::ssysobs

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

5.11.2.38 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`.

5.11.2.39 double wcsprm::zsource

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

5.11.2.40 char wcsprm::ssyssrc

(Given, auxiliary) ... the spectral reference frame (standard of rest) in which this was measured, **SSYSSRC**`a`.

5.11.2.41 double wcsprm::velangl

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

5.11.2.42 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.

5.11.2.43 int wcsprm::ntab

(*Given*) See wcsprm::tab.

5.11.2.44 int wcsprm::nwtb

(*Given*) See wcsprm::wtb.

5.11.2.45 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.

5.11.2.46 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.

5.11.2.47 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.
- 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.

5.11.2.48 char wcsprm::lngtyp

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

5.11.2.49 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.)

5.11.2.50 int wcsprm::lng

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

5.11.2.51 int wcsprm::lat

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

5.11.2.52 int wcsprm::spec

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

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

5.11.2.53 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).

5.11.2.54 struct linprm wcsprm::lin

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

5.11.2.55 struct celprm wcsprm::cel

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

5.11.2.56 struct spcprm wcsprm::spc

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

5.11.2.57 int wcsprm::m_flag

(For internal use only.)

5.11.2.58 int wcsprm::m_naxis

(For internal use only.)

5.11.2.59 double * wcsprm::m_crpix

(For internal use only.)

5.11.2.60 double * wcsprm::m_pc

(For internal use only.)

5.11.2.61 double * wcsprm::m_cdelt

(For internal use only.)

5.11.2.62 double * wcsprm::m_crval

(For internal use only.)

5.11.2.63 wcsprm::m_cunit

(For internal use only.)

5.11.2.64 wcsprm::m_ctype

(For internal use only.)

5.11.2.65 struct pvcard * wcsprm::m_pv

(For internal use only.)

5.11.2.66 struct pscard * wcsprm::m_ps

(For internal use only.)

5.11.2.67 double * wcsprm::m_cd

(For internal use only.)

5.11.2.68 double * wcsprm::m_crota

(For internal use only.)

5.11.2.69 int * wcsprm::m_colax

(For internal use only.)

5.11.2.70 wcsprm::m_cname

(For internal use only.)

5.11.2.71 double * wcsprm::m_crder

(For internal use only.)

5.11.2.72 double * wcsprm::m_csyer

(For internal use only.)

5.11.2.73 struct tabprm * wcsprm::m_tab

(For internal use only.)

5.11.2.74 struct wtbarr * wcsprm::m_wtb

(For internal use only.)

5.12 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
- int * dimlen
- double ** arrayp

5.12.1 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 wcs.h itself.

5.12.2 Field Documentation

5.12.2.1 int wtbarr::i

(*Given*) Image axis number.

5.12.2.2 int wtbarr::m

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

5.12.2.3 int wtbarr::kind

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

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

5.12.2.4 char wtbarr::extnam

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

5.12.2.5 int wtbarr::extver

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

5.12.2.6 int wtbarr::extlev

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

5.12.2.7 char wtbarr::ttype

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

5.12.2.8 long wtbarr::row

(*Given*) Table row number.

5.12.2.9 int wtbarr::ndim

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

5.12.2.10 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.

5.12.2.11 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.

5.12.2.12 int* wtbarr::dimlen

5.12.2.13 double** wtbarr::arrayp

6 WCSLIB 4.4 File Documentation

6.1 cel.h File Reference

```
#include "prj.h"
```

Data Structures

- struct celprm

  Celestial transformation parameters.

Defines

- #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.

Functions

- int celini (struct celprm *cel)

  Default constructor for the celprm struct.

- int celprt (const struct celprm *cel)

  Print routine for the 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 [ ]

  Status return messages.

6.1.1 Detailed Description

These routines implement the part of the FITS World Coordinate System (WCS) standard that deals with celestial coordinates. They 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, and another, celprt(), to print its contents.

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 Define Documentation

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

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

6.1.2.2 #define celini_errmsg cel_errmsg

Deprecated

Added for backwards compatibility, use cel_errmsg directly now instead.

6.1.2.3 #define celprt_errmsg cel_errmsg

Deprecated

Added for backwards compatibility, use cel_errmsg directly now instead.

6.1.2.4 #define celset_errmsg cel_errmsg

Deprecated

Added for backwards compatibility, use cel_errmsg directly now instead.

6.1.2.5 #define celx2s_errmsg cel_errmsg

Deprecated

Added for backwards compatibility, use cel_errmsg directly now instead.

6.1.2.6 #define cels2x_errmsg cel_errmsg

Deprecated

Added for backwards compatibility, use cel_errmsg directly now instead.

6.1.3 Function Documentation

6.1.3.1 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.

Parameters:

→ ***cel*** Celestial transformation parameters.

Returns:

Status return value:

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

6.1.3.2 int celprt (const struct celprm * *cel*)

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

Parameters:

← ***cel*** Celestial transformation parameters.

Returns:

Status return value:

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

6.1.3.3 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:

↔ ***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.

6.1.3.4 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:

↔ ***cel*** Celestial transformation parameters.

← ***nx,ny*** Vector lengths.

← ***sxy,sll*** Vector strides.

← ***x,y*** Projected coordinates in pseudo "degrees".

→ ***phi,theta*** Longitude and latitude (ϕ, θ) in the native coordinate system of the projection [deg].

→ ***lng,lat*** Celestial longitude and latitude (α, δ) of the projected point [deg].

→ ***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.

6.1.3.5 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:

$\leftrightarrow$ ***cel*** Celestial transformation parameters.

$\leftarrow$ ***nlng,nlat*** Vector lengths.

$\leftarrow$ ***sll,sxy*** Vector strides.

$\leftarrow$ ***lng,lat*** Celestial longitude and latitude (α, δ) of the projected point [deg].

$\rightarrow$ ***phi,theta*** Longitude and latitude (ϕ, θ) in the native coordinate system of the projection [deg].

$\rightarrow$ ***x,y*** Projected coordinates in pseudo "degrees".

$\rightarrow$ ***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.

6.1.4 Variable Documentation

6.1.4.1 const char $*$ cel_errmsg[ ]

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

6.2 fitshdr.h File Reference

```
#include "wcsconfig.h"
```

Data Structures

- struct fitskeyid

  Keyword indexing.

- struct fitskey

  Keyword/value information.

Defines

- #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.

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.

6.2.1 Detailed Description

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.

6.2.2 Define Documentation

6.2.2.1 #define FITSHDR_KEYWORD 0x01

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

6.2.2.2 #define FITSHDR_KEYVALUE 0x02

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

6.2.2.3 #define FITSHDR_COMMENT 0x04

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

6.2.2.4 #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.

6.2.2.5 #define FITSHDR_CARD 0x08

Deprecated

Added for backwards compatibility, use *FITSHDR_KEYREC* instead.

6.2.2.6 #define FITSHDR_TRAILER 0x10

Bit mask for the status flag bit-vector returned by fitshdr() indicating a keyrecord following a valid END keyrecord.

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

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

6.2.3 Typedef Documentation

6.2.3.1 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.

6.2.4 Function Documentation

6.2.4.1 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:

← ***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.

← ***nkeyrec*** Number of keyrecords in header[].

← ***nkeyids*** Number of entries in keyids[].

↔ ***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::name are returned by **fitshdr**(). All matched keywords will have their fitskey::keyno member negated.

→ ***nreject*** Number of header keyrecords rejected for syntax errors.

→ ***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 by invoking free() on the array.

Returns:

Status return value:

- 0: Success.
- 1: Null fitskey pointer passed.
- 2: Memory allocation failed.
- 3: Fatal error returned by Flex parser.

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 translates 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.

6.2.5 Variable Documentation

6.2.5.1 const char ∗ fitshdr_errmsg[ ]

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

6.3 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.*

6.3.1 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.

6.3.2 Function Documentation

6.3.2.1 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:

← ***fptr*** Pointer to the file handle returned, for example, by the fits_open_file() routine in CFITSIO.

← ***nwtb*** Number of arrays to be read from the binary table(s).

↔ ***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.

→ ***status*** CFITSIO status value.

Returns:

CFITSIO status value.

Notes:

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.

6.4 lin.h File Reference

Data Structures

- struct linprm

  Linear transformation parameters.

Defines

- #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.

Functions

- int linini (int alloc, int naxis, struct linprm ∗lin)

  Default constructor for the 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 linprt (const struct linprm ∗lin)

  Print routine for the 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 matinv (int n, const double mat[ ], double inv[ ])

  Matrix inversion.

Variables

- const char ∗ lin_errmsg [ ]

  Status return messages.

6.4.1 Detailed Description

These routines apply the linear transformation defined by the FITS WCS standard. They 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.

Three routines, linini(), lincpy(), and linfree() are provided to manage the linprm struct, and another, linprt(), prints its contents.

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 matrix inversion routine, matinv(), is included. It uses LU-triangular factorization with scaled partial pivoting.

6.4.2 Define Documentation

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

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

6.4.2.2 #define linini_errmsg lin_errmsg

Deprecated

Added for backwards compatibility, use lin_errmsg directly now instead.

6.4.2.3 #define lincpy_errmsg lin_errmsg

Deprecated

Added for backwards compatibility, use lin_errmsg directly now instead.

6.4.2.4 #define linfree_errmsg lin_errmsg

Deprecated

Added for backwards compatibility, use lin_errmsg directly now instead.

6.4.2.5 #define linprt_errmsg lin_errmsg

Deprecated

Added for backwards compatibility, use lin_errmsg directly now instead.

6.4.2.6 #define linset_errmsg lin_errmsg

Deprecated

Added for backwards compatibility, use lin_errmsg directly now instead.

6.4.2.7 #define linp2x_errmsg lin_errmsg

Deprecated

Added for backwards compatibility, use lin_errmsg directly now instead.

6.4.2.8 #define linx2p_errmsg lin_errmsg

Deprecated

Added for backwards compatibility, use lin_errmsg directly now instead.

6.4.3 Function Documentation

6.4.3.1 int linini (int *alloc*, int *naxis*, struct linprm ∗ *lin*)

linini() allocates memory for arrays in a linprm struct and sets all members of the struct to default values.

PLEASE NOTE: every linprm struct should be initialized by **linini**(), 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 **linini**() will actually be used to allocate memory.

Parameters:

← ***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.)

← ***naxis*** The number of world coordinate axes, used to determine array sizes.

↔ ***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).

Returns:

Status return value:

- 0: Success.
- 1: Null linprm pointer passed.
- 2: Memory allocation failed.

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

lincpy() does a deep copy of one linprm struct to another, using linini() 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:

← ***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.

← ***linsrc*** Struct to copy from.

↔ ***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.

6.4.3.3 int linfree (struct linprm * *lin*)

linfree() frees memory allocated for the linprm arrays by linini() and/or linset(). linini() 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 linini().

Parameters:

← ***lin*** Linear transformation parameters.

Returns:

Status return value:

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

6.4.3.4 int linprt (const struct linprm * *lin*)

linprt() prints the contents of a linprm struct.

Parameters:

← ***lin*** Linear transformation parameters.

Returns:

Status return value:

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

6.4.3.5 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:

↔ ***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.

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

linp2x() transforms pixel coordinates to intermediate world coordinates.

Parameters:

↔ ***lin*** Linear transformation parameters.

← ***ncoord,nelem*** The number of coordinates, each of vector length nelem but containing lin.naxis coordinate elements.

← ***pixcrd*** Array of pixel coordinates.

→ ***imgcrd*** Array of intermediate world coordinates.

Returns:

Status return value:

- 0: Success.
- 1: Null linprm pointer passed.
- 2: Memory allocation failed.
- 3: **PC**`i_ja` matrix is singular.

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

linx2p() transforms intermediate world coordinates to pixel coordinates.

Parameters:

↔ ***lin*** Linear transformation parameters.

← ***ncoord,nelem*** The number of coordinates, each of vector length nelem but containing lin.naxis coordinate elements.

← ***imgcrd*** Array of intermediate world coordinates.

→ ***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.

6.4.3.8 matinv (int *n*, const double *mat*[ ], double *inv*[ ])

matinv() performs matrix inversion using LU-triangular factorization with scaled partial pivoting.

Parameters:

← ***n*** Order of the matrix ($n \times n$).

← ***mat*** Matrix to be inverted, stored as mat[$in + j$] where i and j are the row and column indices respectively.

→ ***inv*** Inverse of mat with the same storage convention.

Returns:

Status return value:

- 0: Success.
- 2: Memory allocation failed.
- 3: Singular matrix.

6.4.4 Variable Documentation

6.4.4.1 const char ∗ lin_errmsg[ ]

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

6.5 log.h File Reference

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.

6.5.1 Detailed Description

These routines implement the part of the FITS WCS standard that deals with logarithmic coordinates. They 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.

6.5.2 Function Documentation

6.5.2.1 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:

↔ ***crval*** Log-coordinate reference value (**CRVAL**`ia`).

← ***nx*** Vector length.

← ***sx*** Vector stride.

← ***slogc*** Vector stride.

← ***x*** Intermediate world coordinates, in SI units.

→ ***logc*** Logarithmic coordinates, in SI units.

→ ***stat*** Status return value status for each vector element:

- 0: Success.
- 1: Invalid value of x.

Returns:

Status return value:

- 0: Success.
- 2: Invalid log-coordinate reference value.
- 3: One or more of the x coordinates were invalid, as indicated by the stat vector.

6.5.2.2 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:

↔ ***crval*** Log-coordinate reference value (**CRVAL**`ia`).

← ***nlogc*** Vector length.

← ***slogc*** Vector stride.

← ***sx*** Vector stride.

← ***logc*** Logarithmic coordinates, in SI units.

→ ***x*** Intermediate world coordinates, in SI units.

→ ***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.

6.5.3 Variable Documentation

6.5.3.1 const char * log_errmsg[ ]

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

6.6 prj.h File Reference

Data Structures

- struct prjprm

  Projection parameters.

Defines

- #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.

Functions

- int prjini (struct prjprm ∗prj)

  Default constructor for the prjprm struct.

- int prjprt (const struct prjprm ∗prj)

  Print routine for the prjprm struct.

- 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.*

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 [27][4]

  Recognized three-letter projection codes.

6.6.1 Detailed Description

These routines implement the spherical map projections defined by the FITS WCS standard. They 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, and another, prjprt(), to print its contents.

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.
- prjprt() Routine to print the prjprm struct.
- 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)

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 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 no 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.

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.

6.6.2 Define Documentation

6.6.2.1 #define PVN 30

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

6.6.2.2 #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.

6.6.2.3 #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.

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

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

6.6.2.5 #define prjini_errmsg prj_errmsg

Deprecated

Added for backwards compatibility, use prj_errmsg directly now instead.

6.6.2.6 #define prjprt_errmsg prj_errmsg

Deprecated

Added for backwards compatibility, use prj_errmsg directly now instead.

6.6.2.7 #define prjset_errmsg prj_errmsg

Deprecated

Added for backwards compatibility, use prj_errmsg directly now instead.

6.6.2.8 #define prjx2s_errmsg prj_errmsg

Deprecated

Added for backwards compatibility, use prj_errmsg directly now instead.

6.6.2.9 #define prjs2x_errmsg prj_errmsg

Deprecated

Added for backwards compatibility, use prj_errmsg directly now instead.

6.6.3 Function Documentation

6.6.3.1 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.

Parameters:

→ ***prj*** Projection parameters.

Returns:

Status return value:

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

6.6.3.2 int prjprt (const struct prjprm * *prj*)

prjprt() prints the contents of a prjprm struct.

Parameters:

← ***prj*** Projection parameters.

Returns:

Status return value:

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

6.6.3.3 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:

↔ ***prj*** Projection parameters.

Returns:

Status return value:

- 0: Success.
- 1: Null prjprm pointer passed.
- 2: Invalid projection parameters.

6.6.3.4 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:

↔ ***prj*** Projection parameters.

← ***nx,ny*** Vector lengths.

← ***sxy,spt*** Vector strides.

← ***x,y*** Projected coordinates.

→ ***phi,theta*** Longitude and latitude (ϕ, θ) of the projected point in native spherical coordinates [deg].

→ ***stat*** Status return 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.

6.6.3.5 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:

$\leftrightarrow$ ***prj*** Projection parameters.

$\leftarrow$ ***nphi,ntheta*** Vector lengths.

$\leftarrow$ ***spt,sxy*** Vector strides.

$\leftarrow$ ***phi,theta*** Longitude and latitude (ϕ, θ) of the projected point in native spherical coordinates [deg].

$\rightarrow$ ***x,y*** Projected coordinates.

$\rightarrow$ ***stat*** Status return 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.

6.6.3.6 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.

6.6.3.7 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.

6.6.3.8 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.

6.6.3.9 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.

6.6.3.10 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.

6.6.3.11 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.

6.6.3.12 int tanset (struct prjprm * *prj*)

tanset() sets up a prjprm struct for a **gnomonic** (`TAN`) projection.

See prjset() for a description of the API.

6.6.3.13 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.

6.6.3.14 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.

6.6.3.15 int stgset (struct prjprm * *prj*)

stgset() sets up a prjprm struct for a **stereographic** (`STG`) projection.

See prjset() for a description of the API.

6.6.3.16 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.

6.6.3.17 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.

6.6.3.18 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.

6.6.3.19 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.

6.6.3.20 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.

6.6.3.21 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.

6.6.3.22 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.

6.6.3.23 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.

6.6.3.24 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.

6.6.3.25 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.

6.6.3.26 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.

6.6.3.27 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.

6.6.3.28 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.

6.6.3.29 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.

6.6.3.30 int airset (struct prjprm * *prj*)

airset() sets up a prjprm struct for an **Airy (**`AIR`**)** projection.

See prjset() for a description of the API.

6.6.3.31 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.

6.6.3.32 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.

6.6.3.33 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.

6.6.3.34 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.

6.6.3.35 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.

6.6.3.36 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.

6.6.3.37 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.

6.6.3.38 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.

6.6.3.39 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.

6.6.3.40 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.

6.6.3.41 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.

6.6.3.42 int merset (struct prjprm * *prj*)

merset() sets up a prjprm struct for a **Mercator** (MER) projection.

See prjset() for a description of the API.

6.6.3.43 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.

6.6.3.44 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.

6.6.3.45 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.

6.6.3.46 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.

6.6.3.47 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.

6.6.3.48 int parset (struct prjprm * *prj*)

parset() sets up a prjprm struct for a **parabolic** (PAR) projection.

See prjset() for a description of the API.

6.6.3.49 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.

6.6.3.50 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.

6.6.3.51 int molset (struct prjprm * *prj*)

molset() sets up a prjprm struct for a **Mollweide** (MOL) projection.

See prjset() for a description of the API.

6.6.3.52 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.

6.6.3.53 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.

6.6.3.54 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.

6.6.3.55 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.

6.6.3.56 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.

6.6.3.57 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.

6.6.3.58 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.

6.6.3.59 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.

6.6.3.60 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.

6.6.3.61 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.

6.6.3.62 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.

6.6.3.63 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.

6.6.3.64 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.

6.6.3.65 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.

6.6.3.66 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.

6.6.3.67 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.

6.6.3.68 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.

6.6.3.69 int bonset (struct prjprm * *prj*)

bonset() sets up a prjprm struct for a **Bonne** (BON) projection.

See prjset() for a description of the API.

6.6.3.70 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.

6.6.3.71 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.

6.6.3.72 int pcoset (struct prjprm * *prj*)

pcoset() sets up a prjprm struct for a **polyconic** (PCO) projection.

See prjset() for a description of the API.

6.6.3.73 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.

6.6.3.74 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.

6.6.3.75 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.

6.6.3.76 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.

6.6.3.77 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.

6.6.3.78 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.

6.6.3.79 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.

6.6.3.80 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.

6.6.3.81 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.

6.6.3.82 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.

6.6.3.83 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.

6.6.3.84 int hpxset (struct prjprm * *prj*)

hpxset() sets up a prjprm struct for a **HEALPix** (`HPX`) projection.

See prjset() for a description of the API.

6.6.3.85 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.

6.6.3.86 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.

6.6.4 Variable Documentation

6.6.4.1 const char * prj_errmsg[ ]

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

6.6.4.2 const int CONIC

Identifier for conic projections, see prjprm::category.

6.6.4.3 const int CONVENTIONAL

Identifier for conventional projections, see prjprm::category.

6.6.4.4 const int CYLINDRICAL

Identifier for cylindrical projections, see prjprm::category.

6.6.4.5 const int POLYCONIC

Identifier for polyconic projections, see prjprm::category.

6.6.4.6 const int PSEUDOCYLINDRICAL

Identifier for pseudocylindrical projections, see prjprm::category.

6.6.4.7 const int QUADCUBE

Identifier for quadcube projections, see prjprm::category.

6.6.4.8 const int ZENITHAL

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

6.6.4.9 const int HEALPIX

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

6.6.4.10 const char prj_categories[9][32]

Names of the projection categories, all in lower-case except for "HEALPix".

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

6.6.4.11 const int prj_ncode

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

6.6.4.12 const char prj_codes[27][4]

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

6.7 spc.h File Reference

```
#include "spx.h"
```

Data Structures

- struct spcprm

    Spectral transformation parameters.

Defines

- #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.

Functions

- int spcini (struct spcprm *spc)

    Default constructor for the spcprm struct.

- int spcprt (const struct spcprm ∗spc)

  Print routine for the 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 spctyp (const char ctype[ ], char stype[ ], char scode[ ], char sname[ ], char units[ ], char ∗ptype, char ∗xtype, int ∗restreq)

  Spectral **CTYPE**`ia` *keyword analysis.*

- int spcspx (const char ctypeS[ ], double crvalS, double restfrq, double restwav, char ∗ptype, char ∗xtype, int ∗restreq, double ∗crvalX, double ∗dXdS)

  Spectral keyword analysis.

- int spcxps (const char ctypeS[ ], double crvalX, double restfrq, double restwav, char ∗ptype, char ∗xtype, int ∗restreq, double ∗crvalS, double ∗dSdX)

  Spectral keyword synthesis.

- int spctrn (const char ctypeS1[ ], double crvalS1, double cdeltS1, double restfrq, double restwav, char ctypeS2[ ], double ∗crvalS2, double ∗cdeltS2)

  Spectral keyword translation.

Variables

- const char ∗ spc_errmsg [ ]

  Status return messages.

6.7.1 Detailed Description

These routines implement the part of the FITS WCS standard that deals with spectral coordinates. They 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, and another, spcprt(), to print its contents.

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:

- spctyp() checks a spectral **CTYPE**`ia` keyword for validity and returns information derived from it.
- Spectral keyword analysis routine spcspx() computes the values of the X-type spectral variables for the S-type variables supplied.
- Spectral keyword synthesis routine, spcxps(), computes the S-type variables for the X-types supplied.
- Given a set of spectral keywords, a translation routine, spctrn(), produces the corresponding set for the specified spectral **CTYPE**`ia`.

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.

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.

6.7.2 Define Documentation

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

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

6.7.2.2 #define spcini_errmsg spc_errmsg

Deprecated

Added for backwards compatibility, use spc_errmsg directly now instead.

6.7.2.3 #define spcprt_errmsg spc_errmsg

Deprecated

Added for backwards compatibility, use spc_errmsg directly now instead.

6.7.2.4 #define spcset_errmsg spc_errmsg

Deprecated

Added for backwards compatibility, use spc_errmsg directly now instead.

6.7.2.5 #define spcx2s_errmsg spc_errmsg

Deprecated

Added for backwards compatibility, use spc_errmsg directly now instead.

6.7.2.6 #define spcs2x_errmsg spc_errmsg

Deprecated

Added for backwards compatibility, use spc_errmsg directly now instead.

6.7.3 Function Documentation

6.7.3.1 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.

Parameters:

↔ ***spc*** Spectral transformation parameters.

Returns:

Status return value:

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

6.7.3.2 int spcprt (const struct spcprm * *spc*)

spcprt() prints the contents of a spcprm struct.

Parameters:

← ***spc*** Spectral transformation parameters.

Returns:

Status return value:

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

6.7.3.3 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:

↔ ***spc*** Spectral transformation parameters.

Returns:

Status return value:

- 0: Success.
- 1: Null spcprm pointer passed.
- 2: Invalid spectral parameters.

6.7.3.4 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:

↔ ***spc*** Spectral transformation parameters.

← ***nx*** Vector length.

← ***sx*** Vector stride.

← ***sspec*** Vector stride.

← ***x*** Intermediate world coordinates, in SI units.

→ ***spec*** Spectral coordinates, in SI units.

→ ***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.

6.7.3.5 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:

↔ ***spc*** Spectral transformation parameters.

← ***nspec*** Vector length.

← ***sspec*** Vector stride.

← ***sx*** Vector stride.

← ***spec*** Spectral coordinates, in SI units.

→ ***x*** Intermediate world coordinates, in SI units.

→ ***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.

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

spctyp() 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). It recognizes and translates AIPS-convention spectral **CTYPE**`ia` keyvalues.

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.

Parameters:

← ***ctype*** The **CTYPE**`ia` keyvalue, (eight characters with null termination).

→ ***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.

→ ***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.

→ ***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.

→ ***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.

→ ***ptype*** Character code for the P-type spectral variable derived from ctype, one of 'F', 'W', 'A', or 'V'.

→ ***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.

→ ***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
```

.

Returns:

Status return value:

- 0: Success.
- 2: Invalid spectral parameters (not a spectral **CTYPE**`ia`).

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

spcspx() analyses the **CTYPE**`ia` and **CRVAL**`ia` FITS spectral axis keyword values and returns information about the associated X-type spectral variable.

Parameters:

← ***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).

← ***crvalS*** Value of the S-type spectral variable at the reference point, i.e. the **CRVAL**`ia` keyvalue, SI units.

← ***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'**.

→ ***ptype*** Character code for the P-type spectral variable derived from ctypeS, one of 'F', 'W', 'A', or 'V'.

→ ***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.

→ ***restreq*** Multivalued flag that indicates whether rest frequency or wavelength is required to compute spectral variables for this **CTYPE**`ia`, as for spctyp().

→ ***crvalX*** Value of the X-type spectral variable at the reference point, SI units.

→ ***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.

Returns:

Status return value:

- 0: Success.
- 2: Invalid spectral parameters.

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

spcxps(), 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.

Parameters:

← ***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).

← ***crvalX*** Value of the X-type spectral variable at the reference point (N.B. NOT the **CRVAL**`ia` keyvalue), SI units.

← ***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'**.

→ ***ptype*** Character code for the P-type spectral variable derived from ctypeS, one of 'F', 'W', 'A', or 'V'.

→ ***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.

→ ***restreq*** Multivalued flag that indicates whether rest frequency or wavelength is required to compute spectral variables for this **CTYPE**`ia`, as for spctyp().

→ ***crvalS*** Value of the S-type spectral variable at the reference point (i.e. the appropriate **CRVAL**`ia` keyvalue), SI units.

→ ***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.

Returns:

Status return value:

- 0: Success.
- 2: Invalid spectral parameters.

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

spctrn() 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.

Parameters:

← ***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). AIPS-convention spectral types are accepted for ctypeS1 but the Doppler frame encoded within them will not be used.

← ***crvalS1*** Value of the S-type spectral variable at the reference point, i.e. the **CRVAL**`ia` keyvalue, SI units.

← ***cdeltS1*** Increment of the S-type spectral variable at the reference point, SI units.

← ***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'**.

↔ ***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).

AIPS-convention spectral types are not accepted for ctypeS2.

→ ***crvalS2*** Value of the new S-type spectral variable at the reference point, i.e. the new **CRVAL**`ia` keyvalue, SI units.

→ ***cdeltS2*** Increment of the new S-type spectral variable at the reference point, i.e. the new **CDELT**`ia` keyvalue, SI units.

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.

6.7.4 Variable Documentation

6.7.4.1 const char * spc_errmsg[ ]

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

6.8 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[ ])

  Angular distance and position angle.

6.8.1 Detailed Description

The WCS spherical coordinate transformations are implemented via separate functions, sphx2s() and sphs2x(), for the transformation in each direction.

A utility function, sphdpa(), uses these to compute the angular distance and position angle from a given point on the sky to a number of other points.

6.8.2 Function Documentation

6.8.2.1 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:

← ***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])

← ***nphi,ntheta*** Vector lengths.

← ***spt,sxy*** Vector strides.

← ***phi,theta*** Longitude and latitude in the native coordinate system of the projection [deg].

→ ***lng,lat*** Celestial longitude and latitude [deg].

Returns:

Status return value:

- 0: Success.

6.8.2.2 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:

← ***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])

← ***nlng,nlat*** Vector lengths.

← ***sll,spt*** Vector strides.

← ***lng,lat*** Celestial longitude and latitude [deg].

→ ***phi,theta*** Longitude and latitude in the native coordinate system of the projection [deg].

Returns:

Status return value:

- 0: Success.

6.8.2.3 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.

Parameters:

← ***nfield*** The number of field points.

← ***lng0,lat0*** Spherical coordinates of the reference point [deg].

← ***lng,lat*** Spherical coordinates of the field points [deg].

→ ***dist,pa*** Angular distance and position angle [deg].

Returns:

Status return value:

- 0: Success.

Notes:

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.

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.

6.9 spx.h File Reference

Data Structures

- struct spxprm

  Spectral variables and their derivatives.

Defines

- #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.

Functions

- int specx (const char *type, double spec, double restfrq, double restwav, struct spxprm *specs)
  Spectral cross conversions (scalar).
- 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 [ ]

  Status return messages.

6.9.1 Detailed Description

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.

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.

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.

6.9.2 Define Documentation

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

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

6.9.2.2 #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.

6.9.3 Function Documentation

6.9.3.1 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:

← ***type*** The type of spectral variable given by spec, **FREQ**, **AFRQ**, **ENER**, **WAVN**, **VRAD**, **WAVE**, **VOPT**, **ZOPT**, **AWAV**, **VELO**, or **BETA** (case sensitive).

← ***spec*** The spectral variable given, in SI units.

← ***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**.

↔ ***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.

6.9.3.2 int freqafrq (SPX_ARGS)

freqafrq() converts frequency to angular frequency.

Parameters:

← ***param*** Ignored.

← ***nspec*** Vector length.

← ***instep,outstep*** Vector strides.

← ***inspec*** Input spectral variables, in SI units.

→ ***outspec*** Output spectral variables, in SI units.

→ ***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.

6.9.3.3 int afrqfreq (SPX_ARGS)

afrqfreq() converts angular frequency to frequency.

See freqafrq() for a description of the API.

6.9.3.4 int freqener (SPX_ARGS)

freqener() converts frequency to photon energy.

See freqafrq() for a description of the API.

6.9.3.5 int enerfreq (SPX_ARGS)

enerfreq() converts photon energy to frequency.

See freqafrq() for a description of the API.

6.9.3.6 int freqwavn (SPX_ARGS)

freqwavn() converts frequency to wave number.

See freqafrq() for a description of the API.

6.9.3.7 int wavnfreq (SPX_ARGS)

wavnfreq() converts wave number to frequency.

See freqafrq() for a description of the API.

6.9.3.8 int freqwave (SPX_ARGS)

freqwave() converts frequency to vacuum wavelength.

See freqafrq() for a description of the API.

6.9.3.9 int wavefreq (SPX_ARGS)

wavefreq() converts vacuum wavelength to frequency.

See freqafrq() for a description of the API.

6.9.3.10 int freqawav (SPX_ARGS)

freqawav() converts frequency to air wavelength.

See freqafrq() for a description of the API.

6.9.3.11 int awavfreq (SPX_ARGS)

awavfreq() converts air wavelength to frequency.

See freqafrq() for a description of the API.

6.9.3.12 int waveawav (SPX_ARGS)

waveawav() converts vacuum wavelength to air wavelength.

See freqafrq() for a description of the API.

6.9.3.13 int awavwave (SPX_ARGS)

awavwave() converts air wavelength to vacuum wavelength.

See freqafrq() for a description of the API.

6.9.3.14 int velobeta (SPX_ARGS)

velobeta() converts relativistic velocity to relativistic beta.

See freqafrq() for a description of the API.

6.9.3.15 int betavelo (SPX_ARGS)

betavelo() converts relativistic beta to relativistic velocity.

See freqafrq() for a description of the API.

6.9.3.16 int freqvelo (SPX_ARGS)

freqvelo() converts frequency to relativistic velocity.

Parameters:

← ***param*** Rest frequency [Hz].

← ***nspec*** Vector length.

← ***instep,outstep*** Vector strides.

← ***inspec*** Input spectral variables, in SI units.

→ ***outspec*** Output spectral variables, in SI units.

→ ***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.

6.9.3.17 int velofreq (SPX_ARGS)

velofreq() converts relativistic velocity to frequency.

See freqvelo() for a description of the API.

6.9.3.18 int freqvrad (SPX_ARGS)

freqvrad() converts frequency to radio velocity.

See freqvelo() for a description of the API.

6.9.3.19 int vradfreq (SPX_ARGS)

vradfreq() converts radio velocity to frequency.

See freqvelo() for a description of the API.

6.9.3.20 int wavevelo (SPX_ARGS)

wavevelo() converts vacuum wavelength to relativistic velocity.

Parameters:

← ***param*** Rest wavelength in vacuo [m].

← ***nspec*** Vector length.

← ***instep,outstep*** Vector strides.

← ***inspec*** Input spectral variables, in SI units.

→ ***outspec*** Output spectral variables, in SI units.

→ ***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.

6.9.3.21 int velowave (SPX_ARGS)

velowave() converts relativistic velocity to vacuum wavelength.

See freqvelo() for a description of the API.

6.9.3.22 int awavvelo (SPX_ARGS)

awavvelo() converts air wavelength to relativistic velocity.

See freqvelo() for a description of the API.

6.9.3.23 int veloawav (SPX_ARGS)

veloawav() converts relativistic velocity to air wavelength.

See freqvelo() for a description of the API.

6.9.3.24 int wavevopt (SPX_ARGS)

wavevopt() converts vacuum wavelength to optical velocity.

See freqvelo() for a description of the API.

6.9.3.25 int voptwave (SPX_ARGS)

voptwave() converts optical velocity to vacuum wavelength.

See freqvelo() for a description of the API.

6.9.3.26 int wavezopt (SPX_ARGS)

wavevopt() converts vacuum wavelength to redshift.

See freqvelo() for a description of the API.

6.9.3.27 int zoptwave (SPX_ARGS)

zoptwave() converts redshift to vacuum wavelength.

See freqvelo() for a description of the API.

6.9.4 Variable Documentation

6.9.4.1 const char ∗ spx_errmsg[ ]

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

6.10 tab.h File Reference

Data Structures

- struct tabprm

  Tabular transformation parameters.

Defines

- #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.

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 tabfree (struct tabprm ∗tab)
  Destructor for the tabprm struct.
- int tabprt (const struct tabprm ∗tab)
  Print routine for the 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.

6.10.1 Detailed Description

These routines implement the part of the FITS WCS standard that deals with tabular coordinates, i.e. coordinates that are defined via a lookup table. They 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, and another, tabprt(), to print its contents.

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.

6.10.2 Define Documentation

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

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

6.10.2.2 #define tabini_errmsg tab_errmsg

Deprecated

Added for backwards compatibility, use tab_errmsg directly now instead.

6.10.2.3 #define tabcpy_errmsg tab_errmsg

Deprecated

Added for backwards compatibility, use tab_errmsg directly now instead.

6.10.2.4 #define tabfree_errmsg tab_errmsg

Deprecated

Added for backwards compatibility, use tab_errmsg directly now instead.

6.10.2.5 #define tabprt_errmsg tab_errmsg

Deprecated

Added for backwards compatibility, use tab_errmsg directly now instead.

6.10.2.6 #define tabset_errmsg tab_errmsg

Deprecated

Added for backwards compatibility, use tab_errmsg directly now instead.

6.10.2.7 #define tabx2s_errmsg tab_errmsg

Deprecated

Added for backwards compatibility, use tab_errmsg directly now instead.

6.10.2.8 #define tabs2x_errmsg tab_errmsg

Deprecated

Added for backwards compatibility, use tab_errmsg directly now instead.

6.10.3 Function Documentation

6.10.3.1 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:

← ***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.)

← ***M*** The number of tabular coordinate axes.

← ***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().

↔ ***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.

6.10.3.2 int tabmem (struct tabprm ∗ *tab*)

tabmem() takes control of memory allocated by the user for arrays in the tabprm struct.

Parameters:

↔ ***tab*** Tabular transformation parameters.

Returns:

Status return value:

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

6.10.3.3 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:

← ***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.)

← ***tabsrc*** Struct to copy from.

↔ ***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.

6.10.3.4 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:

→ ***tab*** Coordinate transformation parameters.

Returns:

Status return value:

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

6.10.3.5 int tabprt (const struct tabprm * *tab*)

tabprt() prints the contents of a tabprm struct.

Parameters:

← ***tab*** Tabular transformation parameters.

Returns:

Status return value:

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

6.10.3.6 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:

↔ ***tab*** Tabular transformation parameters.

Returns:

Status return value:

- 0: Success.
- 1: Null tabprm pointer passed.
- 3: Invalid tabular parameters.

6.10.3.7 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:

↔ ***tab*** Tabular transformation parameters.

← ***ncoord,nelem*** The number of coordinates, each of vector length nelem.

← ***x*** Array of intermediate world coordinates, SI units.

→ ***world*** Array of world coordinates, in SI units.

→ ***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.

6.10.3.8 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:

↔ ***tab*** Tabular transformation parameters.

← ***ncoord,nelem*** The number of coordinates, each of vector length nelem.

← ***world*** Array of world coordinates, in SI units.

→ ***x*** Array of intermediate world coordinates, SI units.

→ ***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.

6.10.4 Variable Documentation

6.10.4.1 const char * tab_errmsg[ ]

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

6.11 wcs.h File Reference

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

Data Structures

- struct pvcard
  Store for **PV**`i_ma` *keyrecords.*
- struct pscard
  Store for **PS**`i_ma` *keyrecords.*
- struct wtbarr
  Extraction of coordinate lookup tables from BINTABLE.
- struct wcsprm
  Coordinate transformation parameters.

Defines

- #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 WCSLEN (sizeof(struct wcsprm)/sizeof(int))
  Size of the wcsprm struct in int units.
- #define wcscopy(alloc, wcssrc, wcsdst) wcssub(alloc, wcssrc, 0, 0, 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.

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

  Subimage extraction routine for the wcsprm struct.

- int wcsfree (struct wcsprm ∗wcs)

  Destructor for the wcsprm struct.

- int wcsprt (const struct wcsprm ∗wcs)

  Print routine for the wcsprm struct.

- 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[ ], double vstep, int viter, double world[ ], double phi[ ], double theta[ ], double imgcrd[ ], double pixcrd[ ])

  Hybrid coordinate transformation.

- int wcssptr (struct wcsprm ∗wcs, int ∗i, char ctype[9])

  Spectral axis translation.

Variables

- const char ∗ wcs_errmsg [ ]

  Status return messages.

6.11.1 Detailed Description

These routines 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. They 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.

Three routines, wcsini(), wcssub(), and wcsfree() are provided to manage the wcsprm struct and another, wcsprt(), to 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().

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.

wcssptr() translates the spectral axis in a wcsprm struct. For example, a '**FREQ**' axis may be translated into **'ZOPT-F2W'** and vice versa.

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.

6.11.2 Define Documentation

6.11.2.1 #define WCSSUB_LONGITUDE 0x1001

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

6.11.2.2 #define WCSSUB_LATITUDE 0x1002

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

6.11.2.3 #define WCSSUB_CUBEFACE 0x1004

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

6.11.2.4 #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().

6.11.2.5 #define WCSSUB_SPECTRAL 0x1008

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

6.11.2.6 #define WCSSUB_STOKES 0x1010

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

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

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

6.11.2.8 #define wcscopy(alloc, wcssrc, wcsdst) wcssub(alloc, wcssrc, 0, 0, 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.

6.11.2.9 #define wcsini_errmsg wcs_errmsg

Deprecated

Added for backwards compatibility, use wcs_errmsg directly now instead.

6.11.2.10 #define wcssub_errmsg wcs_errmsg

Deprecated

Added for backwards compatibility, use wcs_errmsg directly now instead.

6.11.2.11 #define wcscopy_errmsg wcs_errmsg

Deprecated

Added for backwards compatibility, use wcs_errmsg directly now instead.

6.11.2.12 #define wcsfree_errmsg wcs_errmsg

Deprecated

Added for backwards compatibility, use wcs_errmsg directly now instead.

6.11.2.13 #define wcsprt_errmsg wcs_errmsg

Deprecated

Added for backwards compatibility, use wcs_errmsg directly now instead.

6.11.2.14 #define wcsset_errmsg wcs_errmsg

Deprecated

Added for backwards compatibility, use wcs_errmsg directly now instead.

6.11.2.15 #define wcsp2s_errmsg wcs_errmsg

Deprecated

Added for backwards compatibility, use wcs_errmsg directly now instead.

6.11.2.16 #define wcss2p_errmsg wcs_errmsg

Deprecated

Added for backwards compatibility, use wcs_errmsg directly now instead.

6.11.2.17 #define wcsmix_errmsg wcs_errmsg

Deprecated

Added for backwards compatibility, use wcs_errmsg directly now instead.

6.11.3 Function Documentation

6.11.3.1 int wcsnpv (int *n*)

wcsnpv() changes the value of NPVMAX (default 64). This global variable controls the number of **PV**`i_ma` keywords that wcsini() should allocate space for.

PLEASE NOTE: This function is not thread-safe.

Parameters:

← ***n*** Value of NPVMAX; ignored if < 0.

Returns:

Current value of NPVMAX.

6.11.3.2 int wcsnps (int *n*)

wcsnps() changes the values of NPSMAX (default 8). This global variable controls the number of **PS**`i_ma` keywords that wcsini() should allocate space for.

PLEASE NOTE: This function is not thread-safe.

Parameters:

← ***n*** Value of NPSMAX; ignored if < 0.

Returns:

Current value of NPSMAX.

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

wcsini() optionally allocates memory for arrays in a wcsprm struct and sets all members of the struct to default values. Memory is allocated for up to NPVMAX **PV**`i_ma` keywords or NPSMAX **PS**`i_ma` keywords per WCS representation. These may be changed via wcsnpv() and wcsnps() before **wcsini**() is called.

PLEASE NOTE: every wcsprm struct should be initialized by **wcsini**(), 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 **wcsini**() will actually be used to allocate memory.

Parameters:

← ***alloc*** If true, allocate memory unconditionally for the crpix, etc. arrays.
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*** 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.

↔ ***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).

Returns:

Status return value:

- 0: Success.
- 1: Null wcsprm pointer passed.
- 2: Memory allocation failed.

6.11.3.4 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 wcsini() to allocate memory for its arrays if required. Only the "information to be provided" part of the struct is extracted; a call to wcsset() is required to set up the remainder.

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 **PC**`i_ja` 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.

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.)

Parameters:

← ***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.

← ***wcssrc*** Struct to extract from.

↔ ***nsub***

↔ ***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.

nsub (the pointer) may be set to zero, and so also may nsub, to indicate the number of axes in the input image; the number of axes will be returned if nsub != 0. axes itself (the pointer) may be set to zero to indicate the first ∗nsub axes in their original order.

Set both 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.

Refer to the notes (below) for further usage examples.

On return, ∗nsub will contain 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. 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 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.

Notes:

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.

6.11.3.5 int wcsfree (struct wcsprm ∗ *wcs*)

wcsfree() frees memory allocated for the wcsprm arrays by wcsini() and/or wcsset(). wcsini() 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 wcsini().

Parameters:

→ ***wcs*** Coordinate transformation parameters.

Returns:

Status return value:

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

6.11.3.6 int wcsprt (const struct wcsprm ∗ *wcs*)

wcsprt() prints the contents of a wcsprm struct.

Parameters:

← ***wcs*** Coordinate transformation parameters.

Returns:

Status return value:

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

6.11.3.7 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 it also recognizes **GLS** as a synonym for **SFL**. It does alias translation for the AIPS spectral types (**'FREQ-LSR'**, **'FELO-HEL'**, etc.) but without changing the input header keywords.

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:

↔ ***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.

6.11.3.8 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:

↔ ***wcs*** Coordinate transformation parameters.

← ***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.

← ***pixcrd*** Array of pixel coordinates.

→ ***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.

→ ***phi,theta*** Longitude and latitude in the native coordinate system of the projection [deg].

→ ***world*** Array of world coordinates. For celestial axes, world[][wcs.lng] and world[][wcs.lat] are the celestial longitude and latitude [deg]. For spectral axes, imgcrd[][wcs.spec] is the intermediate spectral coordinate, in SI units.

→ ***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.

6.11.3.9 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:

↔ ***wcs*** Coordinate transformation parameters.

← ***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.

← ***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.

→ ***phi,theta*** Longitude and latitude in the native coordinate system of the projection [deg].

→ ***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.

→ ***pixcrd*** Array of pixel coordinates.

→ ***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.

6.11.3.10 int wcsmix (struct wcsprm ∗ *wcs*, int *mixpix*, int *mixcel*, const double *vspan*[ ], 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:

↔ ***wcs*** Indices for the celestial coordinates obtained by parsing the wcsprm::ctype[].

← ***mixpix*** Which element of the pixel coordinate is given.

← ***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].

← ***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.

← ***vstep*** Step size for solution search [deg]. If zero, a sensible, although perhaps non-optimal default will be used.

← ***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.

↔ ***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.

→ ***phi,theta*** Longitude and latitude in the native coordinate system of the projection [deg].

→ ***imgcrd*** Image coordinate elements. imgcrd[wcs.lng] and imgcrd[wcs.lat] are the projected x-, and y-coordinates in pseudo "degrees".

↔ ***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.

Notes:

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.

6.11.3.11 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.

Parameters:

↔ ***wcs*** Coordinate transformation parameters.

↔ ***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.

↔ ***ctype*** Required 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).

6.11.4 Variable Documentation

6.11.4.1 const char * wcs_errmsg[ ]

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

6.12 wcsfix.h File Reference

```
#include "wcs.h"
```

Defines

- #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 UNITFIX 2

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

- #define CELFIX 3

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

- #define SPCFIX 4

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

- #define CYLFIX 5

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

- #define NWCSFIX 6

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

- #define cylfix_errmsg wcsfix_errmsg

  Deprecated.

Functions

- int wcsfix (int ctrl, const int naxis[ ], struct wcsprm ∗wcs, int stat[ ])

  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 unitfix (int ctrl, struct wcsprm ∗wcs)

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

- int celfix (struct wcsprm ∗wcs)

  Translate AIPS-convention celestial projection types.

- int spcfix (struct wcsprm ∗wcs)

  Translate AIPS-convention spectral types.

- int cylfix (const int naxis[ ], struct wcsprm ∗wcs)

  Fix malformed cylindrical projections.

Variables

- const char ∗ wcsfix_errmsg [ ]

  Status return messages.

6.12.1 Detailed Description

Routines in this suite identify and translate various forms of non-standard construct that are known to occur in FITS WCS headers. These range from the translation of non-standard values for standard WCS keywords, to the repair of malformed coordinate representations.

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 the 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() applies 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 and derive mjdobs from it if not already set. Alternatively, if mjdobs is set and dateobs isn't, then derive dateobs from it.
- unitfix(): translate some commonly used but non-standard unit strings in the **CUNIT**`ia` keyvalues, e.g. '**DEG**' -> '**deg**'.
- celfix(): translate AIPS-convention celestial projection types, **NCP** and **GLS**, in ctype[] as set from **CTYPE**`ia`.
- spcfix(): translate AIPS-convention spectral types, **'FREQ-LSR'**, **'FELO-HEL'**, etc., 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.

6.12.2 Define Documentation

6.12.2.1 #define CDFIX 0

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

6.12.2.2 #define DATFIX 1

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

6.12.2.3 #define UNITFIX 2

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

6.12.2.4 #define CELFIX 3

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

6.12.2.5 #define SPCFIX 4

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

6.12.2.6 #define CYLFIX 5

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

6.12.2.7 #define NWCSFIX 6

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

6.12.2.8 #define cylfix_errmsg wcsfix_errmsg

Deprecated

Added for backwards compatibility, use wcsfix_errmsg directly now instead.

6.12.3 Function Documentation

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

wcsfix() applies all of the corrections handled separately by datfix(), unitfix(), celfix(), spcfix() and cylfix().

Parameters:

← ***ctrl*** Do potentially unsafe translations of non-standard unit strings as described in the usage notes to wcsutrn().

← ***naxis*** Image axis lengths. If this array pointer is set to zero then cylfix() will not be invoked.

↔ ***wcs*** Coordinate transformation parameters.

→ ***stat*** Status returns from each of the functions. Use the preprocessor macros NWCSFIX to dimension this vector and CDFIX, DATFIX, UNITFIX, CELFIX, SPCFIX and CYLFIX to access its elements. A status value of -2 is set for functions that were not invoked.

Returns:

Status return value:

- 0: Success.
- 1: One or more of the translation functions returned an error.

6.12.3.2 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 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.

Parameters:

↔ ***wcs*** Coordinate transformation parameters.

Returns:

Status return value:

- -1: No change required (not an error).
- 0: Success.
- 1: Null wcsprm pointer passed.

6.12.3.3 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*) and derives **MJD-OBS** from it if not already set. Alternatively, if wcsprm::mjdobs is set and wcsprm::dateobs isn't, then **datfix**() derives wcsprm::dateobs from it. If both are set but disagree by more than half a day then status 5 is returned.

Parameters:

↔ ***wcs*** Coordinate transformation parameters. wcsprm::dateobs and/or wcsprm::mjdobs 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.

Notes:

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*.

6.12.3.4 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.

Parameters:

← ***ctrl*** Do potentially unsafe translations described in the usage notes to wcsutrn().

↔ ***wcs*** Coordinate transformation parameters.

Returns:

Status return value:

- -1: No change required (not an error).
- 0: Success.
- 1: Null wcsprm pointer passed.

6.12.3.5 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**. If the pv[] array was initially allocated by wcsini() then the array will be expanded if necessary. Otherwise, error 2 will be returned if two empty slots are not already available for use.

Parameters:

↔ ***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.

6.12.3.6 int spcfix (struct wcsprm * *wcs*)

spcfix() translates AIPS-convention spectral coordinate types, '{**FREQ**,**FELO**,**VELO**}-{**OBS**,**HEL**,**LSR**}' (e.g. **'FREQ-LSR'**, **'FELO-HEL'**, **'VELO-OBS'**) set in wcsprm::ctype[].

Parameters:

↔ ***wcs*** Coordinate transformation parameters. wcsprm::ctype[] and/or wcsprm::specsys may be changed.

Returns:

tatus 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.

6.12.3.7 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.

Parameters:

← ***naxis*** Image axis lengths.

↔ ***wcs*** Coordinate transformation parameters.

Returns:

tatus 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.

6.12.4 Variable Documentation

6.12.4.1 const char ∗ wcsfix_errmsg[ ]

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

6.13 wcshdr.h File Reference

```
#include "wcs.h"
```

Defines

- #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_CROTAia 0x00000001

  Bit mask for wcspih() and wcsbth() - accept **CROTA**`ia`*,* `i`**CROT**`na`*,* **TCROT**`na`*.*

- #define WCSHDR_EPOCHa 0x00000002

  Bit mask for wcspih() and wcsbth() - accept **EPOCH**a.

- #define WCSHDR_VELREFa 0x00000004

  Bit mask for wcspih() and wcsbth() - accept **VELREF**a.

- #define WCSHDR_CD00i00j 0x00000008

  Bit mask for wcspih() and wcsbth() - accept **CD00**i**00**j.

- #define WCSHDR_PC00i00j 0x00000010

  Bit mask for wcspih() and wcsbth() - accept **PC00**i**00**j.

- #define WCSHDR_PROJPn 0x00000020

  Bit mask for wcspih() and wcsbth() - accept **PROJP**n.

- #define WCSHDR_RADECSYS 0x00000040

  Bit mask for wcspih() and wcsbth() - accept **RADECSYS**.

- #define WCSHDR_VSOURCE 0x00000080

  Bit mask for wcspih() and wcsbth() - accept **VSOURCE**a.

- #define WCSHDR_DOBSn 0x00000100

  Bit mask for wcspih() and wcsbth() - accept **DOBS**n.

- #define WCSHDR_LONGKEY 0x00000200

  Bit mask for wcspih() and wcsbth() - accept long forms of the alternate binary table and pixel list WCS keywords.

- #define WCSHDR_CNAMn 0x00000400

  Bit mask for wcspih() and wcsbth() - accept i**CNAM**n*,* **TCNAM**n*,* i**CRDE**n*,* **TCRDE**n*,* i**CSYE**n*,* **TC-SYE**n.

- #define WCSHDR_AUXIMG 0x00000800

  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 0x00001000

  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 0x00010000

  Bit mask for wcsbth() - restrict to image header keywords only.

- #define WCSHDR_BIMGARR 0x00020000

  Bit mask for wcsbth() - restrict to binary table image array keywords only.

- #define WCSHDR_PIXLIST 0x00040000

  Bit mask for wcsbth() - restrict to pixel list keywords only.

- #define WCSHDO_none 0x00

Bit mask for wcshdo() - don't write any extensions.

- #define WCSHDO_all 0xFF

  Bit mask for wcshdo() - write all extensions.

- #define WCSHDO_safe 0x0F

  Bit mask for wcshdo() - write safe extensions only.

- #define WCSHDO_DOBSn 0x01

  Bit mask for wcshdo() - write **DOBS**`n`.

- #define WCSHDO_TPCn_ka 0x02

  Bit mask for wcshdo() - write **TPC**`n_ka`.

- #define WCSHDO_PVn_ma 0x04

  Bit mask for wcshdo() - write `i`**PV**`n_ma`, **TPV**`n_ma`, `i`**PS**`n_ma`, **TPS**`n_ma`.

- #define WCSHDO_CRPXna 0x08

  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 0x10

  Bit mask for wcshdo() - write `i`**CNAM**`na`, **TCNAM**`na`, `i`**CRDE**`na`, **TCRDE**`na`, `i`**CSYE**`na`, **TCSYE**`na`.

- #define WCSHDO_WCSNna 0x20

  Bit mask for wcshdo() - write **WCSN**`na` *instead of* **TWCS**`na`.

Functions

- int wcspih (char ∗header, int nkeyrec, int relax, int ctrl, int ∗nreject, int ∗nwcs, struct wcsprm ∗∗wcs)

  FITS WCS parser routine for image headers.

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

  FITS WCS parser routine for binary table and image headers.

- int wcstab (struct wcsprm ∗wcs)

  Tabular construction routine.

- int wcsidx (int nwcs, struct wcsprm ∗∗wcs, int alts[27])

  Index alternate coordinate representations.

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

  Index alternate coordinate representions.

- int wcsvfree (int ∗nwcs, struct wcsprm ∗∗wcs)

  Free the array of wcsprm structs.

- int wcshdo (int relax, struct wcsprm ∗wcs, int ∗nkeyrec, char ∗∗header)

  Write out a wcsprm struct as a FITS header.

Variables

- const char ∗ wcshdr_errmsg [ ]

  Status return messages.

6.13.1 Detailed Description

Routines in this suite are aimed at extracting WCS information from a FITS file. They 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 free'd 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.

6.13.2 Define Documentation

6.13.2.1 #define WCSHDR_none 0x00000000

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

Refer to wcsbth() note 5.

6.13.2.2 #define WCSHDR_all 0x000FFFFF

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

Refer to wcsbth() note 5.

6.13.2.3 #define WCSHDR_reject 0x10000000

Bit mask for the *relax* argument of wcspih() and wcsbth() - reject non-standard keywords.

Refer to wcsbth() note 5.

6.13.2.4 #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.

6.13.2.5 #define WCSHDR_EPOCHa 0x00000002

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

Refer to wcsbth() note 5.

6.13.2.6 #define WCSHDR_VELREFa 0x00000004

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

Refer to wcsbth() note 5.

6.13.2.7 #define WCSHDR_CD00i00j 0x00000008

Bit mask for the *relax* argument of wcspih() and wcsbth() - accept **CD00**`i`**00**`j`.

Refer to wcsbth() note 5.

6.13.2.8 #define WCSHDR_PC00i00j 0x00000010

Bit mask for the *relax* argument of wcspih() and wcsbth() - accept **PC00**`i`**00**`j`.

Refer to wcsbth() note 5.

6.13.2.9 #define WCSHDR_PROJPn 0x00000020

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

Refer to wcsbth() note 5.

6.13.2.10 #define WCSHDR_RADECSYS 0x00000040

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

Refer to wcsbth() note 5.

6.13.2.11 #define WCSHDR_VSOURCE 0x00000080

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

Refer to wcsbth() note 5.

6.13.2.12 #define WCSHDR_DOBSn 0x00000100

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

Refer to wcsbth() note 5.

6.13.2.13 #define WCSHDR_LONGKEY 0x00000200

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.

6.13.2.14 #define WCSHDR_CNAMn 0x00000400

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.

6.13.2.15 #define WCSHDR_AUXIMG 0x00000800

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.

6.13.2.16 #define WCSHDR_ALLIMG 0x00001000

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.

6.13.2.17 #define WCSHDR_IMGHEAD 0x00010000

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

6.13.2.18 #define WCSHDR_BIMGARR 0x00020000

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

6.13.2.19 #define WCSHDR_PIXLIST 0x00040000

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

6.13.2.20 #define WCSHDO_none 0x00

Bit mask for the *relax* argument of wcshdo() - don't write any extensions.

Refer to the notes for wcshdo().

6.13.2.21 #define WCSHDO_all 0xFF

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

Refer to the notes for wcshdo().

6.13.2.22 #define WCSHDO_safe 0x0F

Bit mask for the *relax* argument of wcshdo() - write only extensions that are considered safe.

Refer to the notes for wcshdo().

6.13.2.23 #define WCSHDO_DOBSn 0x01

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().

6.13.2.24 #define WCSHDO_TPCn_ka 0x02

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().

6.13.2.25 #define WCSHDO_PVn_ma 0x04

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().

6.13.2.26 #define WCSHDO_CRPXna 0x08

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().

6.13.2.27 #define WCSHDO_CNAMna 0x10

Bit mask for the *relax* argument of wcshdo() - write iCNAMna, TCNAMna, iCRDEna, TCRDEna, iCSYEna, TCSYEna, if less than eight characters instead of iCNAna, TCNAna, iCRDna, TCRDna, iCSYna, TCSYna.

Refer to the notes for wcshdo().

6.13.2.28 #define WCSHDO_WCSNna 0x20

Bit mask for the *relax* argument of wcshdo() - write WCSNna instead of TWCSna.

Refer to the notes for wcshdo().

6.13.3 Function Documentation

6.13.3.1 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, and III 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.

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.

Parameters:

↔ ***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.

← ***nkeyrec*** Number of keyrecords in header[].

← ***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.

← ***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.

The report is written to stderr.

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: 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 the basic keywords **'{DATE,MJD}-{OBS,AVG}'** and **'OBSGEO-{X,Y,Z}'**.

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.

→ ***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.

→ ***nwcs*** Number of coordinate representations found.

→ ***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:

Refer to wcsbth() notes 1, 2, 3, 5, 7, and 8.

6.13.3.2 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, and III 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:

↔ ***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.

← ***nkeyrec*** Number of keyrecords in header[].

← ***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.

← ***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.

The report is written to stderr.

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 the basic keywords **'{DATE,MJD}-{OBS,AVG}'** and **'OBSGEO-{X,Y,Z}'**.

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*** 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.

← ***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.

→ ***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.

→ ***nwcs*** Number of coordinate representations found.

→ ***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 and III that are not parameterised 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 keywords (that are not otherwise accepted). A message will optionally be printed on stderr, 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.

     Useful for testing conformance of a FITS header to the WCS standard.
   - WCSHDR_CROTAia: Accept **CROTA**`ia` (wcspih()), `i`**CROT**`na` (**wcsbth**()), **TCROT**`na` (**wcsbth**()).
   - WCSHDR_EPOCHa: Accept **EPOCH**`a`.
   - 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_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_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_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_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

  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.
- WCSHDR_CNAMn (**wcsbth**() only): Accept i**CNAM**n, i**CRDE**n, i**CSYE**n, **TCNAM**n, **TCRDE**n, and **TCSYE**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:

LATPOLEa	for **LATP**na
LONPOLEa	for **LONP**na
RESTFREQ	for **RFRQ**na
RESTFRQa	for **RFRQ**na
RESTWAVa	for **RWAV**na

whose keyvalues are actually used by WCSLIB, and also keywords that provide auxiliary information that is simply stored in the wcsprm struct:

EPOCH		... (No column-specific form.)
EPOCHa		... Only if WCSHDR_EPOCHa is set.
EQUINOXa	for **EQUI**na	
RADESYSa	for **RADE**na	
RADECSYS	for **RADE**na	... Only if WCSHDR_RADECSYS is set.
SPECSYSa	for **SPEC**na	
SSYSOBSa	for **SOBS**na	
SSYSSRCa	for **SSRC**na	
VELOSYSa	for **VSYS**na	
VELANGLa	for **VANG**na	
VELREF		... (No column-specific form.)
VELREFa		... Only if WCSHDR_VELREFa is set.
VSOURCEa	for **VSOU**na	... Only if WCSHDR_VSOURCE is set.
WCSNAMEa	for **WCSN**na	... Or **TWCS**na (see below).
ZSOURCEa	for **ZSOU**na	

DATE-AVG	for **DAVG**n
DATE-OBS	for **DOBS**n
MJD-AVG	for **MJDA**n
MJD-OBS	for **MJDOB**n
OBSGEO-X	for **OBSGX**n
OBSGEO-Y	for **OBSGY**n
OBSGEO-Z	for **OBSGZ**n

where the image-header keywords on the left provide default values for the column specific keywords on the right.

Keywords in the last group, such as **MJD-OBS**, apply to all alternate representations, so **MJD-OBS** would 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 the only potential difficulty in using WCSHDR_AUXIMG is that of erroneously inheriting one of these four 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, such as **CTYPE**ia, that are parameterized by axis number.

- 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 which 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

  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.

  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. In the AIPS convention **VELREF** 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.

A radio convention velocity is denoted by adding 256 to these, otherwise an optical velocity is indicated.

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* xargument, **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.

   In the absence of a formal mechanism for identifying the columns containing pixel coordinates (as opposed to pixel values or ancillary data recorded at the time the photon or neutrino was detected), 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.

wcsbth() simply collects all WCS keywords for a particular 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.

6.13.3.3 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 wcs.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 free'd 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:

↔ ***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 free'd automatically by wcsfree().

Returns:

Status return value:

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

6.13.3.4 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:

← ***nwcs*** Number of coordinate representations in the array.

← ***wcs*** Pointer to an array of wcsprm structs returned by wcspih() or wcsbth().

→ ***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.

6.13.3.5 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:

← ***nwcs*** Number of coordinate representations in the array.

← ***wcs*** Pointer to an array of wcsprm structs returned by wcsbth().

← ***type*** Select the type of coordinate representation:

- 0: binary table image arrays,
- 1: pixel lists.

→ ***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.

6.13.3.6 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:

↔ ***nwcs*** Number of coordinate representations found; set to 0 on return.

↔ ***wcs*** Pointer to the array of wcsprm structs; set to 0 on return.

Returns:

Status return value:

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

6.13.3.7 int wcshdo (int *relax*, 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**.
- Deprecated (e.g. **CROTA**`n`) or non-standard usage will be translated to standard (this is partially dependent on whether wcsfix() was applied).
- 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.
- 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.
- Additional keywords such as **WCSAXES**`a`, **CUNIT**`ia`, **LONPOLE**`a` and **LATPOLE**`a` may appear.
- The original keycomments will be lost, although **wcshdo**() tries hard to write meaningful comments.
- Keyword order may 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:

← ***relax*** Degree of permissiveness:

- 0: Recognize only FITS keywords defined by the published WCS standard.
- -1: Admit all informal extensions of the WCS standard.

Fine-grained control of the degree of permissiveness is also possible as explained in the notes below.

↔ ***wcs*** Pointer to a wcsprm struct containing coordinate transformation parameters. Will be initialized if necessary.

→ ***nkeyrec*** Number of FITS header keyrecords returned in the "header" array.

→ ***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 free'd by the user to avoid memory leaks.

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:

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

Notes:

wcshdo() interprets the *relax* argument as a vector of flag bits to provide fine-grained control over what non-standard WCS keywords to write. The flag bits are subject to change in future and should be set by using the preprocessor macros (see below) for the purpose.

- WCSHDO_none: Don't use any extensions.
- WCSHDO_all: Write all recognized extensions, equivalent to setting each flag bit.
- WCSHDO_safe: Write all extensions that are considered to be safe and recommended.
- WCSHDO_DOBSn: Write **DOBS**`n`, the column-specific analogue of **DATE-OBS** for use in binary tables and pixel lists. WCS Paper III introduced **DATE-AVG** and **DAVG**`n` but by an oversight **DOBS**`n` (the obvious analogy) was never formally defined by the standard. The alternative to using **DOBS**`n` is to write **DATE-OBS** which applies to the whole table. This usage is considered to be safe and is recommended.
- WCSHDO_TPCn_ka: WCS Paper I defined
  - **TP**`n_ka` and **TC**`n_ka` for pixel lists

  but WCS Paper II uses **TPC**`n_ka` in one example and subsequently the errata for the WCS papers legitimized the use of

  - **TPC**`n_ka` and **TCD**`n_ka` for pixel lists

  provided that the keyword does not exceed eight characters. This usage is considered to be safe and is recommended because of the non-mnemonic terseness of the shorter forms.
- WCSHDO_PVn_ma: WCS Paper I defined
  - `i`**V**`n_ma` and `i`**S**`n_ma` for bintables and
  - **TV**`n_ma` and **TS**`n_ma` for pixel lists

but WCS Paper II uses i**PV**n_ma and **TPV**n_ma in the examples and subsequently the errata for the WCS papers legitimized the use of

- i**PV**n_ma and i**PS**n_ma for bintables and
- **TPV**n_ma and **TPS**n_ma for pixel lists

provided that the keyword does not exceed eight characters. This usage is considered to be safe and is recommended because of the non-mnemonic terseness of the shorter forms.

- WCSHDO_CRPXna: For historical reasons WCS Paper I defined
  - j**CRPX**n, i**CDLT**n, i**CUNI**n, i**CTYP**n, and i**CRVL**n for bintables and
  - **TCRPX**n, **TCDLT**n, **TCUNI**n, **TCTYP**n, and **TCRVL**n for pixel lists

  for use without an alternate version specifier. However, because of the eight-character keyword constraint, in order to accommodate column numbers greater than 99 WCS Paper I also defined

  - j**CRP**na, i**CDE**na, i**CUN**na, i**CTY**na and i**CRV**na for bintables and
  - **TCRP**na, **TCDE**na, **TCUN**na, **TCTY**na and **TCRV**na for pixel lists

  for use with an alternate version specifier (the "a"). Like the PC, CD, PV, and PS keywords there is an obvious tendency to confuse these two forms for column numbers up to 99. It is very unlikely that any parser would reject keywords in the first set with a non-blank alternate version specifier so this usage is considered to be safe and is recommended.

- WCSHDO_CNAMna: WCS Papers I and III defined
  - i**CNA**na, i**CRD**na, and i**CSY**na for bintables and
  - **TCNA**na, **TCRD**na, and **TCSY**na for pixel lists

  By analogy with the above, the long forms would be

  - i**CNAM**na, i**CRDE**na, and i**CSYE**na for bintables and
  - **TCNAM**na, **TCRDE**na, and **TCSYE**na for pixel lists

  Note that these keywords provide auxiliary information only, none of them are needed to compute world coordinates. This usage is potentially unsafe and is not recommended at this time.

- WCSHDO_WCSNna: In light of wcsbth() note 4, write **WCSN**na instead of **TWCS**na for pixel lists. While wcsbth() treats **WCSN**na and **TWCS**na as equivalent, other parsers may not. Consequently, this usage is potentially unsafe and is not recommended at this time.

6.13.4 Variable Documentation

6.13.4.1 const char * wcshdr_errmsg[ ]

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

6.14 wcslib.h File Reference

```
#include "cel.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 "wcsfix.h"
#include "wcshdr.h"
#include "wcsmath.h"
#include "wcstrig.h"
#include "wcsunits.h"
#include "wcsutil.h"
```

6.14.1 Detailed Description

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

6.15 wcsmath.h File Reference

Defines

- #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.

6.15.1 Detailed Description

Definition of mathematical constants used by WCSLIB.

6.15.2 Define Documentation

6.15.2.1 #define PI 3.141592653589793238462643

6.15.2.2 #define D2R PI/180.0

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

6.15.2.3 #define R2D 180.0/PI

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

6.15.2.4 #define SQRT2 1.4142135623730950488

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

6.15.2.5 #define SQRT2INV 1.0/SQRT2

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

6.15.2.6 #define UNDEFINED 987654321.0e99

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

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

Macro used to test for an undefined value.

6.16 wcstrig.h File Reference

```
#include <math.h>
#include "wcsconfig.h"
```

Defines

- #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.

6.16.1 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.

6.16.2 Define Documentation

6.16.2.1 #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 $|v| == 1$.

6.16.3 Function Documentation

6.16.3.1 double cosd (double *angle*)

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

Parameters:

← ***angle*** [deg].

Returns:

Cosine of the angle.

6.16.3.2 double sind (double *angle*)

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

Parameters:

← ***angle*** [deg].

Returns:

Sine of the angle.

6.16.3.3 void sincosd (double *angle*, double ∗ *sin*, double ∗ *cos*)

sincosd() returns the sine and cosine of an angle given in degrees.

Parameters:

← ***angle*** [deg].
→ ***sin*** Sine of the angle.
→ ***cos*** Cosine of the angle.

Returns:

6.16.3.4 double tand (double *angle*)

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

Parameters:

← ***angle*** [deg].

Returns:

Tangent of the angle.

6.16.3.5 double acosd (double *x*)

acosd() returns the inverse cosine in degrees.

Parameters:

← ***x*** in the range [-1,1].

Returns:

Inverse cosine of x [deg].

6.16.3.6 double asind (double *y*)

asind() returns the inverse sine in degrees.

Parameters:

← ***y*** in the range [-1,1].

Returns:

Inverse sine of y [deg].

6.16.3.7 double atand (double *s*)

atand() returns the inverse tangent in degrees.

Parameters:

← ***s***

Returns:

Inverse tangent of s [deg].

6.16.3.8 double atan2d (double *y*, double *x*)

atan2d() returns the polar angle, β, in degrees, of polar coordinates (ρ, β) corresponding Cartesian coordinates (x, y). It is equivalent to the $\arg(x, y)$ function of WCS Paper II, though with transposed arguments.

Parameters:

← ***y*** Cartesian y-coordinate.

← ***x*** Cartesian x-coordinate.

Returns:

Polar angle of (x, y) [deg].

6.17 wcsunits.h File Reference

Defines

- #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.

Functions

- int wcsunits (const char have[ ], const char want[ ], double ∗scale, double ∗offset, double ∗power)

  FITS units specification conversion.

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

  Translation of non-standard unit specifications.

- int wcsulex (const char unitstr[ ], int ∗func, double ∗scale, double units[ ])

  FITS units specification parser.

Variables

- const char ∗ wcsunits_errmsg [ ]

  Status return messages.

- const char ∗ wcsunits_types [ ]

  Names of physical quantities.

- const char ∗ wcsunits_units [ ]

  Names of units.

6.17.1 Detailed Description

Routines in this suite deal with units specifications and conversions:

- wcsunits(): given two unit specifications, derive the conversion from one to the other.
- wcsutrn(): translates certain commonly used but non-standard unit strings. It is intended to be called before wcsulex() which only handles standard FITS units specifications.
- wcsulex(): parses a standard FITS units specification of arbitrary complexity, deriving the conversion to canonical units.

6.17.2 Define Documentation

6.17.2.1 #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.

6.17.2.2 #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.

6.17.2.3 #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.

6.17.2.4 #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.

6.17.2.5 #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.

6.17.2.6 #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.

6.17.2.7 #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.

6.17.2.8 #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.

6.17.2.9 #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.

6.17.2.10 #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.

6.17.2.11 #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.

6.17.2.12 #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.

6.17.2.13 #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.

6.17.2.14 #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.

6.17.2.15 #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.

6.17.2.16 #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.

6.17.2.17 #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.

6.17.2.18 #define WCSUNITS_NTYPE 17

Number of entries in the *units* array returned by wcsulex(), and the wcsunits_types[] and wcsunits_units[] global variables.

6.17.3 Function Documentation

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

wcsunits() derives the conversion from one system of units to another.

Parameters:

← ***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.

← ***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.

→ ***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;
```

Returns:

Status return value:

- 0: Success.
- 1-9: Status return from wcsulex().
- 10: Non-conformant unit specifications.
- 11: Non-conformant functions.

scale is zeroed on return if an error occurs.

6.17.3.2 int wcsutrn (int *ctrl*, char *unitstr*[ ])

wcsutrn() translates certain commonly used but non-standard unit strings, e.g. "DEG", "MHZ", "KELVIN", that are not recognized by wcsulex(), 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.

Parameters:

← ***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.

↔ ***unitstr*** Null-terminated character array containing the units specification to be translated.

Inline units specifications in the 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.

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:

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    angstrom
arcmin      arcmins, ARCMIN, ARCMINS
arcsec      arcsecs, ARCSEC, ARCSECS
beam        BEAM
byte        Byte
d           day, days, (D), DAY, DAYS
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 wcsulex() itself as an unofficial extension of the standard, but they are converted to the standard form here.

6.17.3.3 int wcsulex (const char *unitstr*[ ], int ∗ *func*, double ∗ *scale*, double *units*[ ])

wcsulex() 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.

Parameters:

← ***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.

→ ***func*** Special function type, see note 4:

- 0: None
- 1: log() ...base 10
- 2: ln() ...base e
- 3: exp()

→ ***scale*** Scale factor for the unit specification; multiply a value expressed in the given units by this factor to convert it to canonical units.

→ ***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.

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. **wcsulex**() 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.

6.17.4 Variable Documentation

6.17.4.1 const char * wcsunits_errmsg[ ]

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

6.17.4.2 const char * wcsunits_types[ ]

Names for physical quantities to match the units vector returned by **wcsulex**():

- 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

6.17.4.3 const char * wcsunits_units[ ]

Names for the units (SI) to match the units vector returned by **wcsulex**():

- 0: degree
- 1: steradian
- 2: Coulomb
- 3: mole
- 4: Kelvin
- 5: candela
- 6: kilogram
- 7: metre
- 8: second

The remainder are dimensionless.

6.18 wcsutil.h File Reference

Functions

- 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_allEq (int nvec, int nelem, const double ∗first)
  Test for equality of a particular vector element.
- 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.

6.18.1 Detailed Description

Simple utility functions used by WCSLIB. They are documented here solely as an aid to understanding the code. Thay are not intended for external use - the API may change without notice!

6.18.2 Function Documentation

6.18.2.1 void wcsutil_blank_fill (int *n*, char *c*[ ])

wcsutil_blank_fill() pads a character string with blanks starting with the terminating NULL character.

Used by the Fortran wrapper functions in translating C character strings into Fortran CHARACTER variables.

Parameters:

← ***n*** Length of the character array, c[].

↔ ***c*** The character string. It will not be null-terminated on return.

Returns:

6.18.2.2 void wcsutil_null_fill (int *n*, char *c*[ ])

wcsutil_null_fill() pads a character string with NULL characters.

Used mainly to make character strings intelligible in the GNU debugger - it prints the rubbish following the terminating NULL, obscuring the valid part of the string.

Parameters:

← ***n*** Number of characters.

↔ ***c*** The character string.

Returns:

6.18.2.3 int wcsutil_allEq (int *nvec*, int *nelem*, const double ∗ *first*)

wcsutil_allEq() tests for equality of a particular element in a set of vectors.

Parameters:

← ***nvec*** The number of vectors.

← ***nelem*** The length of each vector.

← ***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.

6.18.2.4 void wcsutil_setAll (int *nvec*, int *nelem*, double ∗ *first*)

wcsutil_setAll() sets the value of a particular element in a set of vectors.

Parameters:

← ***nvec*** The number of vectors.

← ***nelem*** The length of each vector.

↔ ***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:

6.18.2.5 void wcsutil_setAli (int *nvec*, int *nelem*, int ∗ *first*)

wcsutil_setAli() sets the value of a particular element in a set of vectors.

Parameters:

← ***nvec*** The number of vectors.

← ***nelem*** The length of each vector.

↔ ***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:

6.18.2.6 void wcsutil_setBit (int *nelem*, const int ∗ *sel*, int *bits*, int ∗ *array*)

wcsutil_setBit() sets bits in selected elements of an array.

Parameters:

← ***nelem*** Number of elements in the array.

← ***sel*** Address of a selection array of length nelem. May be specified as the null pointer in which case all elements are selected.

← ***bits*** Bit mask.

↔ ***array*** Address of the array of length nelem.

Returns:

7 WCSLIB 4.4 Page Documentation

7.1 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 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
- "Mapping on the HEALPix Grid" (HPX), Calabretta, M.R., & Roukema, B.F. 2007, MNRAS, 381, 865

Reprints of all published papers may be obtained from NASA's Astrophysics Data System (ADS), http://adsabs.harvard.edu/. Reprints of Papers I, II (+HPX) and III are available from http://www.atnf.csiro.au/~mcalabre/. 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.

7.2 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/~mcalabre/WCS/. The remainder of this manual describes its use.
- **wcstools**, developed by Doug Mink, may be obtained from http://tdc-www.harvard.edu/software/wcstools/.
- **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 is also provided; go to the above site and then look at the section entitled "FITS-WCS Plotting Demo".

Python wrappers to WCSLIB are provided by

- The **Kapteyn Package** http://www.astro.rug.nl/software/kapteyn/ by Hans Terlouw and Martin Vogelaar.
- **pywcs**, http://stsdas.stsci.edu/astrolib/pywcs/ by Michael Droettboom.

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.

Recommended WCS-aware FITS image viewers:

- Bill Joye's **DS9** (http://hea-www.harvard.edu/RD/ds9/), and
- **Fv** by Pan Chai (http://heasarc.gsfc.nasa.gov/ftools/fv/)

both handle 2-D images.

Currently (2009/09/08) 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 out-of-date by the time you read it, best do a web search):

- Fedora (RPM), https://admin.fedoraproject.org/pkgdb/packages/name/wcslib
- Fresh Ports (RPM), http://www.freshports.org/astro/wcslib/
- Gentoo, http://packages.gentoo.org/package/sci-astronomy/wcslib
- RPM (general) http://v2.www.rpmseek.com/cat/Libraries.html?hl=com&cx=591:W

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.

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/.

7.3 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 wcsutrn()).
- 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 wcsulex()).
  - 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.
  - 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.

- *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.

7.4 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 and tabprm, 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 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. wcsini(), 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 and tabprm) 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.

7.5 Memory management

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

- wcsini() 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 linini() for certain of these). Note that wcsini() 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.

- linini(): 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 wcsini() 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.

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.

wcsini() maintains a record of memory it has allocated and this is used by wcsfree() which wcsini() 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 wcsini() 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 linini(), linfree(), and linset() and to tabini(), tabfree(), and tabset().

A memory leak will result if a wcsprm, linprm or tabprm struct goes out of scope before the memory has been *free'd*, either by the relevant routine, wcsfree(), linfree() or tabfree(), 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 or tabprm 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() or tabcpy().

7.6 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(), and tabs2x(), 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) 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.

7.6.1 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.

7.6.2 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.

7.7 Thread-safety

With the following exceptions WCSLIB 4.4 is thread-safe:

- The C code generated by Flex is not re-entrant. Flex does have the capacity for producing re-entrant scanners but they have a different API. This may be handled by a compile-time option in future but in the meantime calls to the header parsers should be serialized via a mutex.

- The low-level functions wcsnpv() and wcsnps() are not thread-safe but within the library itself they are only used by the Flex scanners wcspih() and wcsbth(). They would rarely need to be used by application programmers.

7.8 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 vital 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.

  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.

- 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. *twcssub* tests the extraction of a coordinate description for a subimage from a wcsprm struct by wcssub().

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

- Low level:

  tlin, *tlog*, *tprj1*, *tsph*, *tspc*, *tspc*, 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.

7.9 WCSLIB Fortran wrappers

The Fortran subdirectory contains wrappers, written in C, that allow Fortran programs to use WCSLIB.

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. These must be used in conjunction with wrappers on the routines provided to manage the structs in C, for example `WCSINI`, `WCSSUB`, `WCSCOPY`, `WCSFREE`, and `WCSPRT` which wrap wcsini(), wcssub(), wcscopy(), wcsfree(), and wcsprt().

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 wherever 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.

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 `WCSINI`, 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.

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)

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

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

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

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

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

      etc.

*     Initialize.
      STATUS = WCSSET (WCS)

*     Find the "longitude" axis.
      STATUS = WCSGET (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.

Note that the data type of the third argument to the *PUT and *GET routines may differ 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. Compilers (e.g. g77) may warn of inconsistent usage of this argument but this can (usually) be safely ignored.

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

7.10 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/~mcalabre/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.

7.11 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 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 cels2x_errmsg Added for backwards compatibility, use cel_errmsg directly now instead.

Global FITSHDR_CARD Added for backwards compatibility, use *FITSHDR_KEYREC* instead.

Global linini_errmsg Added for backwards compatibility, use lin_errmsg directly now 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 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 linp2x_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 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 prjs2x_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 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 spcs2x_errmsg Added for backwards compatibility, use spc_errmsg directly now instead.

Global tabini_errmsg Added for backwards compatibility, use tab_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 tabprt_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 tabs2x_errmsg Added for backwards compatibility, use tab_errmsg directly now instead.

Global wcsini_errmsg Added for backwards compatibility, use wcs_errmsg directly now instead.

Global wcssub_errmsg Added for backwards compatibility, use wcs_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 wcsprt_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 wcsp2s_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 wcsmix_errmsg Added for backwards compatibility, use wcs_errmsg directly now instead.

Global cylfix_errmsg Added for backwards compatibility, use wcsfix_errmsg directly now instead.

Index