CCLRC / Rutherford Appleton Laboratory
Particle Physics & Astronomy Research Council
Starlink Project
Starlink User Note 211.27

SUN/211.27

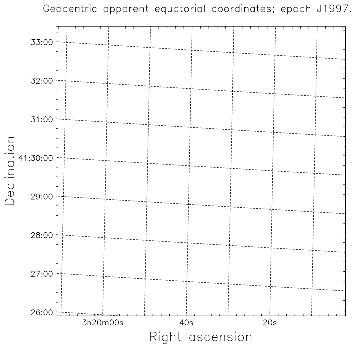
R.F. Warren-Smith & D.S. Berry

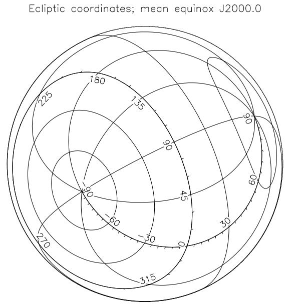
29th April 2013

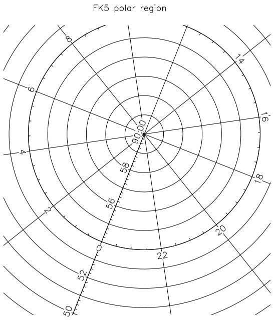
AST
A Library for Handling World Coordinate Systems in Astronomy

V7.3

Programmer's Guide (C Version)







Abstract

The AST library provides a comprehensive range of facilities for attaching world coordinate systems to astronomical data, for retrieving and interpreting that information in a variety of formats, including FITS-WCS, and for generating graphical output based on it.

This programmer's manual should be of interest to anyone writing astronomical applications which need to manipulate coordinate system data, especially celestial or spectral coordinate systems. AST is portable and environment-independent.

Contents

AST
A Library for Handling World Coordinate Systems in Astronomy

V7.3

This is the C version of this document.
For the Fortran version, please see SUN/210.

1 Introduction

Welcome to the AST library. If you are writing software for astronomy and need to use celestial coordinates (*e.g.* RA and Dec), spectral coordinates (*e.g.* wavelength, frequency, *etc.*), or other coordinate system information, then this library should be of interest. It provides solutions for most of the problems you will meet and allows you to write robust and flexible software. It is able to read and write WCS information in a variety of formats, including FITS-WCS.

1.1 What Problems Does AST Tackle?

Here are some of the main problems you may face when handling world coordinate system (WCS) information and the solutions that AST provides:

1. **The Variety of Coordinate Systems**
   Astronomers use a wide range of differing coordinate systems to describe positions within a variety of physical domains. For instance, there are a large number of celestial coordinate systems in use within astronomy to describe positions on the sky. Understanding these, and knowing how to convert coordinates between them, can require considerable expertise. It can also be difficult to decide which of them your software should support. The same applies to coordinate systems describing other domains, such as position within an electromagnetic spectrum.

   Solution. AST has built-in knowledge of many coordinate systems and allows you to convert freely between them without specialist knowledge. This avoids the need to embed details of specific coordinate systems in your software. You also benefit automatically when new coordinate systems are added to AST.

2. **Storing and Retrieving WCS Information**

   Storing coordinate system information in astronomical datasets and retrieving it later can present a considerable challenge. Typically, it requires knowledge of rather complex conventions (*e.g.* FITS) which are low-level, often mis-interpreted and may be subject to change. Exchanging information with other software systems is further complicated by the number of different conventions in use.

   Solution. AST combines a unifying high-level description of WCS information with the ability to save and restore this using a variety of formats. Details of the formats, which include FITS, are handled internally by AST. This frees you from the need to understand them or embed the details in your software. Again, you benefit automatically when new formats are added to AST.

3. **Generating Graphical Output**

   Producing graphical displays involving curvilinear coordinate systems, such as celestial coordinate grids, can be complicated. Particular difficulties arise when handling large areas of sky, the polar regions and discontinuous (*e.g.* segmented) sky projections. Even just numbering and labelling curvilinear axes is rarely straightforward.

   Solution. AST provides plotting facilities especially designed for use with curvilinear coordinate systems. These include the plotting of axes and complete labelled coordinate grids. A large number of options are provided for tailoring the output to your specific needs. Three dimensional coordinate grids can also be produced.

4. **Aligning Data from Different Sources**

   One of the main uses of coordinate systems is to facilitate the inter-comparison of data from different sources. A typical use might be to plot (say) radio contours over an optical image. In practice, however, different celestial coordinate systems may have been used, making accurate alignment far from simple.

   Solution AST provides a one-step method of aligning datasets, searching for all possible intermediate coordinate systems. This makes it simple to directly inter-relate the pixel coordinates of different datasets.

5. **Handling Different Types of Coordinate System**

   Not all coordinate systems used in astronomy are celestial ones, so if you are writing general-purpose software such as (say) a display tool, you may also need to handle axes representing wavelength, distance, time or whatever else comes along. Obviously, you would prefer not to handle each one as a special case.

   Solution AST uses the same flexible high-level model to describe all types of coordinate system. This allows you to write software that handles different kinds of coordinate axis without introducing special cases.

1.2 Other Design Objectives

As well as its scientific objectives, the AST library's design includes a number of technical criteria intended to make it applicable to as wide a range of projects as possible. The main considerations are described here:

1. **Minimum Software Dependencies.** The AST library depends on no other other software[1].

2. **Environment Independence.** AST is designed so that it can operate in a variety of "programming environments" and is not tied to any particular one. To allow this, it uses simple, flexible interfaces to obtain the following services:

   - **Data Storage.** Data I/O operations are based on text and/or FITS headers. This makes it easy to interface to a wide variety of astronomical data formats in a machine-independent way.
   - **Graphics.** Graphical output is produced *via* a simple generic graphics interface, which may easily be re-implemented over different graphics systems. AST provides a default implementation based on the widely-used PGPLOT graphics system (SUN/15).
   - **Error Handling.** Error messages are written to standard error by default, but go through a simple generic interface similar to that used for graphics (above). This permits error message delivery *via* other routes when necessary (*e.g.* in a graphical interface).

3. **Multiple Language Support.** AST has been designed to be called from more than one language. Both C and Fortran interfaces are available (see SUN/210 for the Fortran version) and use from C++ is also straightforward if the C interface is included using:

   ```
   extern "C" {
   #include "ast.h"
   }
   ```

   A JNI interface (known as "JNIAST" - see http://www.starlink.ac.uk/jniast/) has also been developed by Starlink which allows AST to be used from Java.

4. **Object Oriented Design.** AST uses "object oriented" techniques internally in order to provide a flexible and easily-extended programming model. A fairly traditional calling interface is provided, however, so that the library's facilities are easily accessible to programmers using C and Fortran.

5. **Portability.** AST is implemented entirely in ANSI standard C and, when called *via* its C interface, makes no explicit use of any machine-dependent facilities.

   The Fortran interface is, unavoidably, machine dependent. However, the potential for problems has been minimised by encapsulating the interface layer in a compact set of C macros which facilitate its transfer to other platforms. No Fortran compiler is needed to build the library.

   Currently, AST is supported by Starlink on PC Linux, Sun Solaris and Tru64 Unix (formerly DEC UNIX) platforms.

---

[1] It comes with bundled copies of the IAU SOFA and Starlink PAL libraries which are built at the same time as the other AST internal libraries. Alternatively, external PAL and SOFA libraries may be used by specifying the "`--with-external_pal`" option when configuring AST

1.3 What Does “AST” Stand For?

The library name “AST” stands for “ASTrometry Library”. The name arose when it was thought that knowledge of “astrometry” (*i.e.* celestial coordinate systems) would form the bulk of the library. In fact, it turns out that astrometry forms only a minor component, but the name AST has stuck.

2 Overview of AST Concepts

This section presents a brief overview of AST concepts. It is intended as a basic orientation course before you move on to the more technical considerations in subsequent sections.

2.1 Relationships Between Coordinate Systems

The relationships between coordinate systems are represented in AST by Objects called Mappings. A Mapping does not represent a coordinate system itself, but merely the process by which you move from one coordinate system to another related one.

A convenient picture of a Mapping is as a "black box" (Figure 1) into which you can feed sets of coordinates. For each set you feed in, the Mapping returns a corresponding set of transformed coordinates. Since each set of coordinates represents a point in a coordinate space, the Mapping acts to inter-relate corresponding positions in the two spaces, although what these spaces represent is unspecified. Notice that a Mapping need not have the same number of input and output coordinates. That is, the two coordinate spaces which it inter-relates need not have the same number of dimensions.

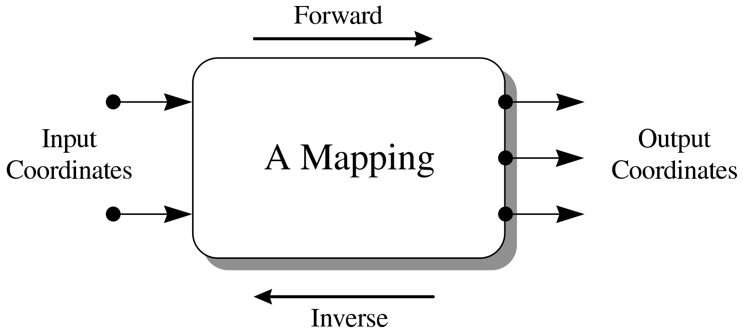


Figure 1: A Mapping viewed as a "black box" for transforming coordinates.

In many cases, the transformation can, in principle, be performed in either direction: either from the *input* coordinate space to the *output,* or *vice versa.* The first of these is termed the *forward* transformation and the other the *inverse* transformation.

Further reading: For a more complete discussion of Mappings, see §5.

2.2 Mappings Available

The basic concept of a Mapping (§2.1) is rather generic and obviously it is necessary to have specific Mappings that implement specific relationships between coordinate systems. AST provides a range of these, to perform transformations such as the following and, where appropriate, their inverses:

- Conversions between various celestial coordinate systems (the SlaMap).
- Conversions between various spectral coordinate systems (the SpecMap and GrismMap).

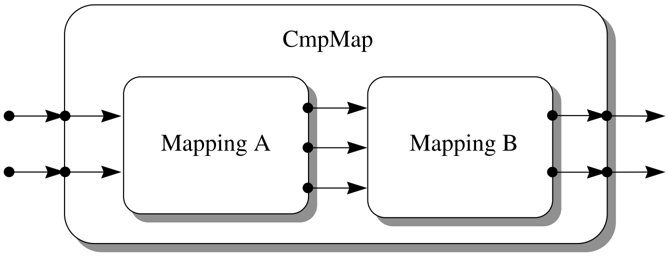


Figure 2: A CmpMap (compound Mapping) composed of two component Mappings joined in series. The output coordinates of the first Mapping feed into the input coordinates of the second one, so that the whole entity behaves like a single Mapping.

- Conversions between various time systems (the TimeMap).
- Conversion between 2-dimensional spherical celestial coordinates (longitude and latitude) and a 3-dimensional vectorial positions (the SphMap).
- Various projections of the celestial sphere on to 2-dimensional coordinate spaces—*i.e.* map projections (the DssMap and WcsMap).
- Permutation, introduction and elimination of coordinates (the PermMap).
- Various linear coordinate transformations (the MatrixMap, WinMap, ShiftMap and ZoomMap).
- General N-dimensional polynomial transformations (the PolyMap).
- Lookup tables (the LutMap).
- General-purpose transformations expressed using arithmetic operations and functions similar to those available in C (the MathMap).
- Transformations for internal use within a program, based on private transformation functions which you write yourself in C (the IntraMap).

Further reading: For a more complete description of each of the Mappings mentioned above, see its entry in Appendix D. In addition, see the discussion of the PermMap in §5.10, the UnitMap in §5.9 and the IntraMap in §20. The ZoomMap is used as an example throughout §4.

2.3 Compound Mappings

The Mappings described in §2.2 provide a set of basic building blocks from which more complex Mappings may be constructed. The key to doing this is a type of Mapping called a CmpMap, or compound Mapping. A CmpMap's role is, in principle, very simple: it allows any other pair of Mappings to be joined together into a single entity which behaves as if it were a single Mapping. A CmpMap is therefore a container for another pair of Mappings.

A pair of Mappings may be combined using a CmpMap in either of two ways. The first of these, *in series,* is illustrated in Figure 2. Here, the transformations implemented by each component Mapping are performed one after the other, with the output from the first Mapping feeding into

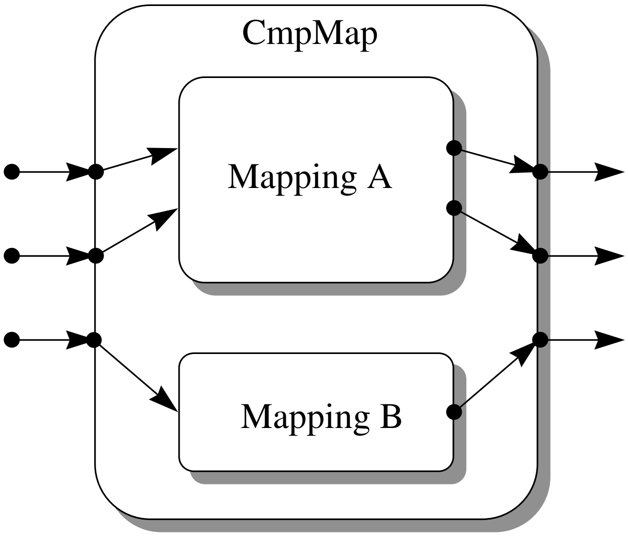


Figure 3: A CmpMap composed of two Mappings joined in parallel. Each component Mapping acts on a complementary subset of the input and output coordinates.

the second. The second way, *in parallel,* is shown in Figure 3. In this case, each Mapping acts on a complementary subset of the input and output coordinates.[2]

The CmpMap forms the key to building arbitrarily complex Mappings because it is itself a form of Mapping. This means that a CmpMap may contain other CmpMaps as components (*e.g.* Figure 4). This nesting of CmpMaps can be repeated indefinitely, so that complex Mappings may be built in a hierarchical manner out of simper ones. This gives AST great flexibility in the coordinate transformations it can describe.

Further reading: For a more complete description of CmpMaps, see §6. Also see the CmpMap entry in Appendix D.

2.4 Representing Coordinate Systems

While Mappings (§2.1) represent the relationships between coordinate systems in AST, the coordinate systems themselves are represented by Objects called Frames (Figure 5). A Frame is similar in concept to the frame you might draw around a graph. It contains information about the labels which appear on the axes, the axis units, a title, knowledge of how to format the coordinate values on each axis, *etc.* An AST Frame is not, however, restricted to two dimensions and may have any number of axes.

A basic Frame may be used to represent a Cartesian coordinate system by setting values for its *attributes* (all AST Objects have values associated with them called attributes, which may be set and enquired). Usually, this would involve setting appropriate axis labels and units, for example. Functions are provided for use with Frames to perform operations such as formatting coordinate values as text, calculating distances between points, interchanging axes, *etc.*

---

[2] A pair of Mappings can be combined in a third way using a TranMap. A TranMap allows the forward transformation of one Mapping to be combined with the inverse transformation of another to produce a single Mapping.

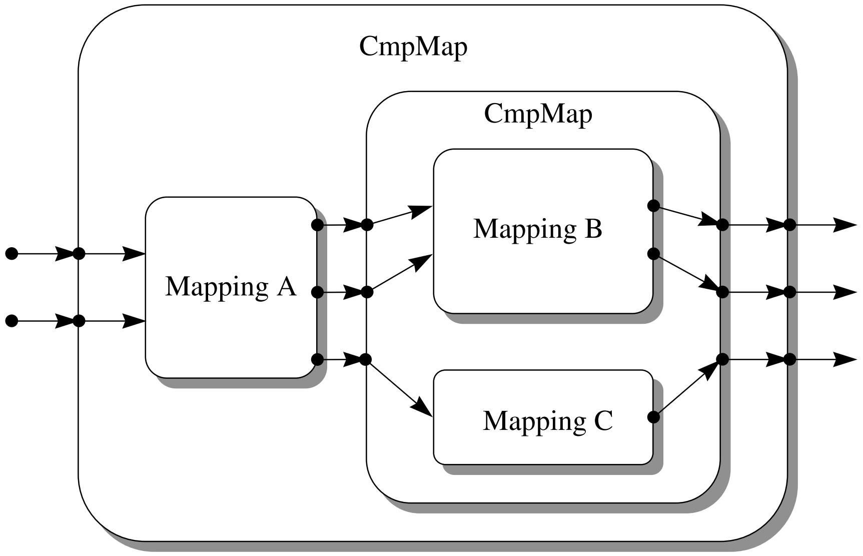


Figure 4: CmpMaps (compound Mappings) may be nested in order to construct complex Mappings out of simpler building blocks.

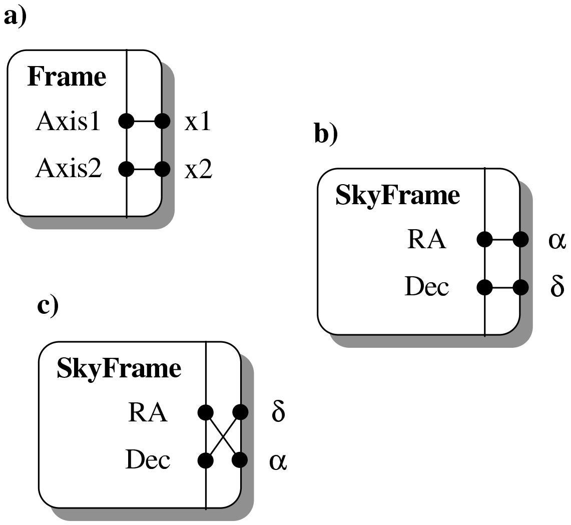


Figure 5: (a) A basic Frame is used to represent a Cartesian coordinate system, here 2-dimensional. (b) A SkyFrame represents a (spherical) celestial coordinate system. (c) The axis order of any Frame may be permuted to match the coordinate space it describes.

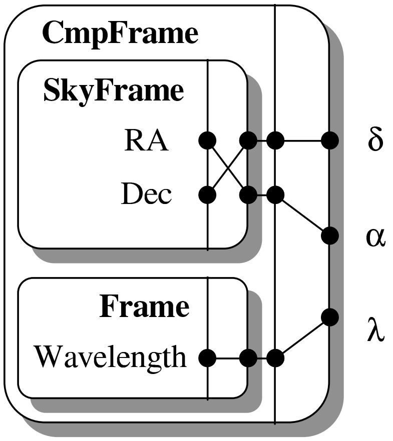


Figure 6: A CmpFrame (compound Frame) formed by combining two simpler Frames. Note how the special relationship which exists between the RA and Dec axes is preserved within this data structure. As with compound Mappings (Figure 4), CmpFrames may be nested in order to build more complex Frames.

There are several more specialised forms of Frame, which provide the additional functionality required when handling coordinates within some specific physical domain. This ranges from tasks such as formatting axis values, to complex tasks such as determining the transformation between any pair of related coordinate systems. For instance, the SkyFrame (Figure 5b,c), represents celestial coordinate systems, the SpecFrame represents spectral coordinate systems, and the TimeFrame represents time coordinate systems. All these provide a wide range of different systems for describing positions within their associated physical domain, and these may be selected by setting appropriate attributes.

As with compound Mappings (§2.3), it is possible to merge two Frames together to form a compound Frame, or CmpFrame, in which both sets of axes are combined. One could, for example, have celestial coordinates on two axes and an unrelated coordinate (wavelength, perhaps) on a third (Figure 6). Knowledge of the relationships between the axes is preserved internally by the process of constructing the CmpFrame which represents them.

Further reading: For a more complete description of Frames see §7, for SkyFrames see §8 and for SpecFrames see §9. Also see the Frame, SkyFrame, SpecFrame, TimeFrame and CmpFrame entries in Appendix D.

2.5 Networks of Coordinate Systems

Mappings and Frames may be connected together to form networks called FrameSets, which are used to represent sets of inter-related coordinate systems (Figure 7). A FrameSet may be extended by adding a new Frame to it, together with an associated Mapping which relates the new coordinate system to one which is already present. This process ensures that there is always exactly one path, *via* Mappings, between any pair of Frames. A function is provided for identifying this path and returning the complete Mapping.

One of the Frames in a FrameSet is termed its *base* Frame. This underlies the FrameSet's purpose, which is to calibrate datasets and other entities by attaching coordinate systems to

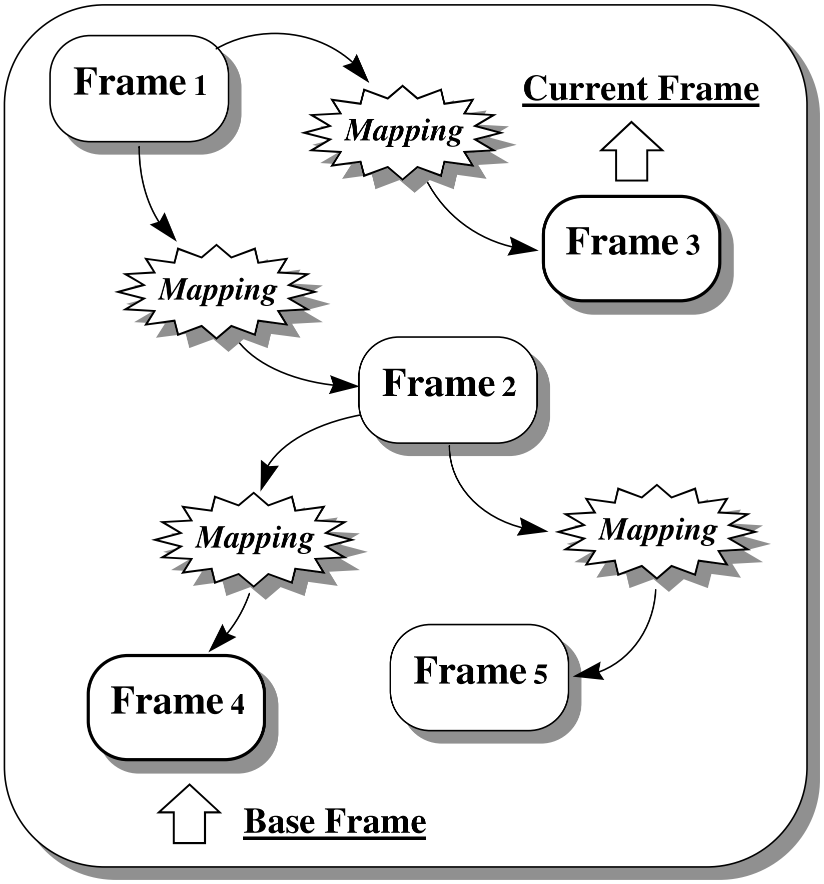


Figure 7: A FrameSet is a network of Frames inter-connected by Mappings such that there is exactly one conversion path, *via* Mappings, between any pair of Frames.

them. In this context, the base Frame represents the "native" coordinate system (for example, the pixel coordinates of an image). Similarly, one Frame is termed the *current* Frame and represents the "currently-selected" coordinates. It might, typically, be a celestial or spectral coordinate system and would be used during interactions with a user, as when plotting axes on a graph or producing a table of results. Other Frames within the FrameSet represent a library of alternative coordinate systems which a software user can select by making them current.

Further reading: For a more complete description of FrameSets, see §13 and §14. Also see the FrameSet entry in Appendix D.

2.6 Input/Output Facilities

AST allows you to convert any kind of Object into a stream of text which contains a full description of that Object. This text may be written out by one program and read back in by another, thus allowing the original Object to be reconstructed.

The filter which converts Objects into text and back again is itself a kind of Object, called a Channel. A Channel provides a number of options for controlling the information content of the text, such as the addition of comments for human interpretation. It is also possible to intercept the text being processed by a Channel so that it may be redirected to/from any chosen external data store, such as a text file, an astronomical dataset, or a network connection.

The text format used by the basic Channel class is peculiar to the AST library - no other software will understand it. However, more specialised forms of Channel are provided which use text formats more widely understood.

To further facilitate the storage of coordinate system information in astronomical datasets, a more specialised form of Channel called a FitsChan is provided. Instead of using free-format text, a FitsChan converts AST Objects to and from FITS header cards. It also allows the information to be encoded in the FITS cards in a number of ways (called *encodings*), so that WCS information from a variety of sources can be handled.

Another sub-class of Channel, called XmlChan, is a specialised form of Channel that stores the text in the form of XML markup. Currently, two markup formats are provided by the XmlChan class, one is closely related to the text format produced by the basic Channel class (currently, no schema or DTD is available describing this format). The other is a subset of an early draft of the IVOA Space-Time-Coordinates XML (STC-X) schema (V1.20) described at http://www.ivoa.net/Documents/WD/STC/STC-20050225.html [3]. The version of STC-X that has been adopted by the IVOA differs in several significant respects from V1.20, and therefore this XmlChan format is of historical interest only.

Finally, the StcsChan class provides facilities for reading and writing IVOA STC-S region descriptions. STC-S (see http://www.ivoa.net/Documents/latest/STC-S.html) is a linear string syntax that allows simple specification of STC metadata. AST supports a subset of the STC-S specification, allowing an STC-S description of a region within an AST-supported astronomical coordinate system to be converted into an equivalent AST Region object, and vice-versa.

Further reading: For a more complete description of Channels see §15 and for FitsChans see §16 and §17. Also see the Channel and FitsChan entries in Appendix D and the Encoding entry in Appendix C.

---

[3] XML documents which use only the subset of the STC schema supported by AST can be read by the XmlChan class to produce corresponding AST objects (subclasses of the Stc class). However, the reverse is not possible. That is, AST objects can not currently be written out in the form of STC documents.

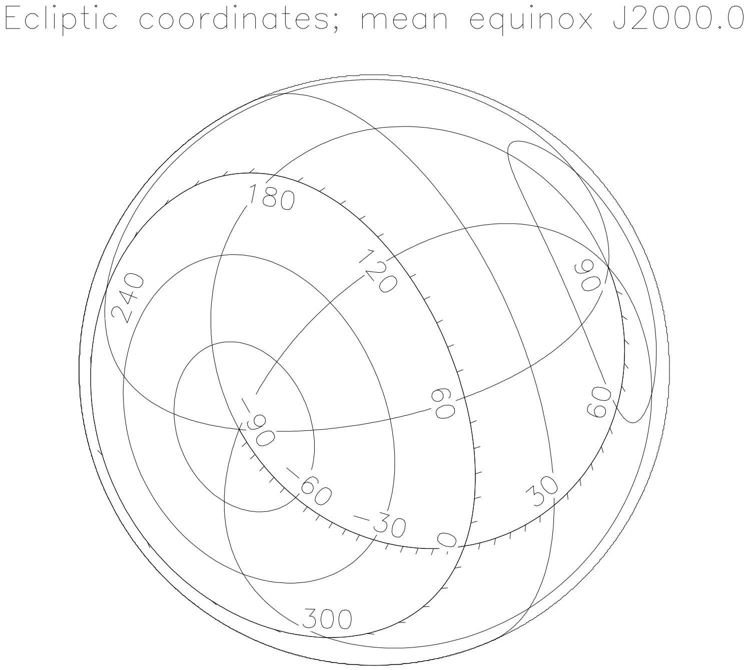


Figure 8: A labelled coordinate grid for an all-sky zenithal equal area projection in ecliptic coordinates. This was composed and drawn *via* a Plot using a single function call.

2.7 Producing Graphical Output

Two dimensional graphical output is supported by a specialised form of FrameSet called a Plot, whose base Frame corresponds with the native coordinates of the underlying graphics system. Plotting operations are specified in *physical coordinates* which correspond with the Plot's current Frame. Typically, this might be a celestial coordinate system.

Three dimensional plotting is also supported, via the Plot3D class - sub-class of Plot.

Operations, such as drawing lines, are automatically transformed from physical to graphical coordinates before plotting, using an adaptive algorithm which ensures smooth curves (because the transformation is usually non-linear). "Missing" coordinates (*e.g.* graphical coordinates which do not project on to the celestial sphere), discontinuities and generalised clipping are all consistently handled. It is possible, for example, to plot in equatorial coordinates and clip in galactic coordinates. The usual plotting operations are provided (text, markers), but a geodesic curve replaces the primitive straight line element. There is also a separate function for drawing axis lines, since these are normally not geodesics.

In addition to drawing coordinate grids over an area of the sky, another common use of the Plot class is to produce line plots such as flux against wavelength, displacement again time, *etc.* For these situations the current Frame of the Plot would be a compound Frame (CmpFrame) containing a pair of 1-dimensional Frames - the first representing the X axis quantity (wavelength, time, etc), and the second representing the Y axis quantity (flux, displacement, etc). The Plot class includes an option for axes to be plotted logarithmically.

Perhaps the most useful graphics function available is for drawing fully annotated coordinate grids (*e.g.* Figure 8). This uses a general algorithm which does not depend on knowledge of the coordinates being represented, so can also handle programmer-defined coordinate systems. Grids for all-sky projections, including polar regions, can be drawn and most aspects of the output (colour, line style, *etc.*) can be adjusted by setting appropriate Plot attributes.

Further reading: For a more complete description of Plots and how to produce graphical output, see §21. Also see the Plot entry in Appendix D.

3 How To…

For those of you with a plane to catch, this section provides some instant templates and recipes for performing the most commonly-required operations using AST, but without going into detail. The examples given (sort of) follow on from each other, so you should be able to construct a variety of programs by piecing them together. Note that some of them appear longer than they actually are, because we have included plenty of comments and a few options that you probably won't need.

If any of this material has you completely baffled, then you may want to read the introduction to AST programming concepts in §4 first. Otherwise, references to more detailed reading are given after each example, just in case they don't quite do what you want.

3.1 …Obtain and Install AST

The AST library is available both as a stand-alone package and also as part of the Starlink Software Collection[4]. If your site has the Starlink Software Collection installed then AST should already be available.

If not, you can download the AST library by itself from http://www.starlink.ac.uk/ast/.

3.2 …Structure an AST Program

An AST program normally has the following structure:

```
/* Include the interface to the AST library. */
#include "ast.h"

/* Main program (or could be any function). */
main () {
   <normal C declarations and statements>

/* Enclose the parts which use AST between the astBegin and astEnd macros. */
   astBegin;
   <C statements which use AST>
   astEnd;

   <maybe more C statements>
}
```

The use of astBegin and astEnd is optional, but has the effect of tidying up after you have finished using AST, so is normally recommended. For more details of this, see §4.10. For details of how to access the "ast.h" header file, see §22.1.

3.3 …Build an AST Program

To build a simple AST program that doesn't use graphics, use:

---

[4] The Starlink Software Collection can be downloaded from http://www.starlink.ac.uk/Download/.

```
cc program.c -L/star/lib -I/star/include ‘ast_link‘ -o program
```

To build a program which uses PGPLOT for graphics, use:

```
cc program.c -L/star/lib ‘ast_link -pgplot‘ -o program
```

For more details about accessing the “ast.h” header file, see §22.1. For more details about linking programs, see §22.2 and the description of the “ast_link” command in Appendix E.

3.4 ...Read a WCS Calibration from a Dataset

Precisely how you extract world coordinate system (WCS) information from a dataset obviously depends on what type of dataset it is. Usually, however, you should be able to obtain a set of FITS header cards which contain the WCS information (and probably much more besides). Suppose that “cards” is a pointer to a string containing a complete set of concatenated FITS header cards (such as produced by the CFITSIO function fits_hdr2str). Then proceed as follows:

```
fitsfile *fptr;
AstFitsChan *fitschan;
AstFrameSet *wcsinfo;
char *header;
int nkeys, status;

...

/* Obtain all the cards in the header concatenated into a single dynamically
   allocated null-terminated character string. Note, we do not exclude
   any cards since we may later modify the WCS information within the
   header and consequently want to write the entire header out again. */
   if( fits_hdr2str( fptr, 0, NULL, 0, &header, &nkeys, &status ) )
          printf(" Error getting header\n");
   ...

/* Header obtained succesfully... */
   } else {

/* Create a FitsChan and fill it with FITS header cards. */
      fitschan = astFitsChan( NULL, NULL, "" );
      astPutCards( fitschan, header );

/* Free the memory holding the concatenated header cards. */
      header = free( header );

/* Read WCS information from the FitsChan. */
      wcsinfo = astRead( fitschan );

      ...
```

The result should be a pointer, "wcsinfo", to a FrameSet which contains the WCS information. This pointer can now be used to perform many useful tasks, some of which are illustrated in the following recipes.

Some datasets which do not easily yield FITS header cards may require a different approach, possibly involving use of a Channel or XmlChan (§15) rather than a FitsChan. In the case of the Starlink NDF data format, for example, all the above may be replaced by a single call to the function ndfGtwcs—see SUN/33. The whole process can probably be encapsulated in a similar way for most data systems, whether they use FITS header cards or not.

For more details about reading WCS information from datasets, see §17.3 and §17.4. For a more general description of FitsChans and their use with FITS header cards, see §16 and §17. For more details about FrameSets, see §13 and §14.

3.5 . . . Validate WCS Information

Once you have read WCS information from a dataset, as in §3.4, you may wish to check that you have been successful. The following will detect and classify the things that might possibly go wrong:

```
#include <string.h>

...

if ( !astOK ) {
   <an error occurred (a message will have been issued)>
} else if ( wcsinfo == AST__NULL ) {
   <there was no WCS information present>
} else if ( strcmp( astGetC( wcsinfo, "Class" ), "FrameSet" ) ) {
   <something unexpected was read (i.e. not a FrameSet)>
} else {
   <WCS information was read OK>
}
```

For more information about detecting errors in AST functions, see §4.15. For details of how to validate input data read by AST, see §15.6 and §17.4.

3.6 . . . Display AST Data

If you have a pointer to any AST Object, you can display the data stored in that Object in textual form as follows:

```
astShow( wcsinfo );
```

Here, we have used a pointer to the FrameSet which we read earlier (§3.4). The result is written to the program's standard output stream. This can be very useful during debugging.

For more details about using astShow, see §4.4. For information about interpreting the output, also see §15.8.

3.7 . . . Convert Between Pixel and World Coordinates

You may use a pointer to a FrameSet, such as we read in §3.4, to transform a set of points between the pixel coordinates of an image and the associated world coordinates. If you are working in two dimensions, proceed as follows:

```
double xpixel[ N ], ypixel[ N ];
double xworld[ N ], yworld[ N ];

...

astTran2( wcsinfo, N, xpixel, ypixel, 1, xworld, yworld );
```

Here, N is the number of points to be transformed, "xpixel" and "ypixel" hold the pixel coordinates, and "xworld" and "yworld" receive the returned world coordinates.[5] To transform in the opposite direction, interchange the two pairs of arrays (so that the world coordinates are given as input) and change the fifth argument of astTran2 to zero.

To transform points in one dimension, use astTran1. In any other number of dimensions (or if the number of dimensions is initially unknown), use astTranN or astTranP. These functions are described in Appendix B.

For more information about transforming coordinates, see §4.8 and §13.6. For details of how to handle missing coordinates, see §5.8.

3.8 . . . Test if a WCS is a Celestial Coordinate System

The world coordinate system (WCS) currently associated with an image may often be a celestial coordinate system, but this need not necessarily be the case. For instance, instead of right ascension and declination, an image might have a WCS with axes representing wavelength and slit position, or maybe just plain old pixels.

If you have obtained a WCS calibration for an image, as in §3.4, in the form of a pointer "wcsinfo" to a FrameSet, then you may determine if the current coordinate system is a celestial one or not, as follows:

```
AstFrame *frame;
int issky;

...

/* Obtain a pointer to the current Frame and determine if it is a
   SkyFrame. */
frame = astGetFrame( wcsinfo, AST__CURRENT );
issky = astIsASkyFrame( frame );
frame = astAnnul( frame );
```

This will set "issky" to 1 if the WCS is a celestial coordinate system, and to zero otherwise.

---

[5] By pixel coordinates, we mean a coordinate system in which the first pixel in the image is centred on (1,1) and each pixel is a unit square. Note that the world coordinates will not necessarily be celestial coordinates, but if they are, then they will be in radians.

3.9 . . . Test if a WCS is a Spectral Coordinate System

Testing for a spectral coordinate system is basically the same as testing for a celestial coordinate system (see the previous section). The one difference is that you use the astIsASpecFrame function in place of the astIsASkyFrame function.

3.10 . . . Format Coordinates for Display

Once you have converted pixel coordinates into world coordinates (§3.7), you may want to format them as text before displaying them. Typically, this would convert from (say) radians into something more comprehensible. Using the FrameSet pointer "wcsinfo" obtained in §3.4 and a pair of world coordinates "xw" and "yw" (*e.g.* see §3.7), you could proceed as follows:

```
#include <stdio.h>
const char *xtext, *ytext;
double xw, yw;

...

xtext = astFormat( wcsinfo, 1, xw );
ytext = astFormat( wcsinfo, 2, yw );

(void) printf( "Position = %s, %s\n", xtext, ytext );
```

Here, the second argument to astFormat is the axis number.

With celestial coordinates, this will usually result in sexagesimal notation, such as "12:34:56.7". However, the same method may be applied to any type of coordinates and appropriate formatting will be employed.

For more information about formatting coordinate values and how to control the style of formatting used, see §7.6 and §8.6. If necessary, also see §7.7 for details of how to "normalise" a set of coordinates so that they lie within the standard range (*e.g.* 0 to 24 hours for right ascension and ±90° for declination).

3.11 . . . Display Coordinates as they are Transformed

In addition to formatting coordinates as part of a program's output, you may also want to examine coordinate values while debugging your program. To save time, you can "eavesdrop" on the coordinate values being processed every time they are transformed. For example, when using the FrameSet pointer "wcsinfo" obtained in §3.4 to transform coordinates (§3.7), you could inspect the coordinate values as follows:

```
astSet( wcsinfo, "Report=1" );
astTran2( wcsinfo, N, xpixel, ypixel, 1, xworld, yworld );
```

By setting the FrameSet's Report attribute to 1, coordinate transformations are automatically displayed on the program's standard output stream, appropriately formatted, for example:

```
(42.1087, 20.2717) --> (2:06:03.0, 34:22:39)
(43.0197, 21.1705) --> (2:08:20.6, 35:31:24)
(43.9295, 22.0716) --> (2:10:38.1, 36:40:09)
(44.8382, 22.9753) --> (2:12:55.6, 37:48:55)
(45.7459, 23.8814) --> (2:15:13.1, 38:57:40)
(46.6528, 24.7901) --> (2:17:30.6, 40:06:25)
(47.5589, 25.7013) --> (2:19:48.1, 41:15:11)
(48.4644, 26.6149) --> (2:22:05.6, 42:23:56)
(49.3695, 27.5311) --> (2:24:23.1, 43:32:41)
(50.2742, 28.4499) --> (2:26:40.6, 44:41:27)
```

For a complete description of the Report attribute, see its entry in Appendix C. For further details of how to set and enquire attribute values, see §4.6 and §4.5.

3.12 ...Read Coordinates Entered by a User

In addition to writing out coordinate values generated by your program (§3.10), you may also need to accept coordinates entered by a user, or perhaps read from a file. In this case, you will probably want to allow "free-format" input, so that the user has some flexibility in the format that can be used. You will probably also want to detect any typing errors.

Let's assume that you want to read a number of lines of text, each containing the world coordinates of a single point, and to split each line into individual numerical coordinate values. Using the FrameSet pointer "wcsinfo" obtained earlier (§3.4), you could proceed as follows:

```
#include <stdio.h>
char *t;
char text[ MAXCHARS + 2 ];
double coord[ 10 ];
int iaxis, n, naxes;

...

/* Obtain the number of coordinate axes (if not already known). */
naxes = astGetI( wcsinfo, "Naxes" );

/* Loop to read each line of input text, in this case from the
   standard input stream (your programming environment will probably
   provide a better way of reading text than this). Set the pointer
   "t" to the start of each line read. */
while ( t = fgets( text, MAXCHARS + 2, stdin ) ) {

/* Attempt to read a coordinate for each axis. */
   for ( iaxis = 1; iaxis <= naxes; iaxis++ ) {
      n = astUnformat( wcsinfo, iaxis, t, &coord[ iaxis - 1 ] );

/* If nothing was read and this is not the first axis or the
   end-of-string, try stepping over a separator and reading again. */
   if ( !n && ( iaxis > 1 ) && *t )
      n = astUnformat( wcsinfo, iaxis, ++t, &coord[ iaxis - 1 ] );
/* Quit if nothing was read, otherwise move on to the next coordinate. */
      if ( !n ) break;
      t += n;
   }

/* Test for the possible errors that may occur... */

/* Error detected by AST (a message will have been issued). */
   if ( !astOK ) {
      break;

/* Error in input data at character t[n]. */
   } else if ( *t || !n ) {
      <handle the error, or report your own message here>
      break;

   } else {
      <coordinates were read OK>
   }
}
```

This algorithm has the advantage of accepting free-format input in whatever style is appropriate for the world coordinates in use (under the control of the FrameSet whose pointer you provide). For example, wavelength values might be read as floating point numbers (*e.g.* “1.047” or “4787”), whereas celestial positions could be given in sexagesimal format (*e.g.* “12:34:56” or “12 34.5”) and would be converted into radians. Individual coordinate values may be separated by white space and/or any non-ambiguous separator character, such as a comma.

For more information on reading coordinate values using the astUnformat function, see §7.8. For details of how sexagesimal formats are handled, and the forms of input that may be used for celestial coordinates, see §8.7.

3.13 . . . Create a New WCS Calibration

This section describes how to add a WCS calibration to a data set which you are creating from scratch, rather than modifying an existing data set.

In most common cases, the simplest way to create a new WCS calibration from scratch is probably to create a set of strings describing the required calibration in terms of the keywords used by the FITS WCS standard, and then convert these strings into an AST FrameSet describing the calibration. This FrameSet can then be used for many other purposes, or simply stored in the data set.

The full FITS-WCS standard is quite involved, currently running to four separate papers, but the basic kernel is quite simple, involving the following keywords (all of which end with an integer axis index, indicated below by $< i >$):

CRPIX¡i¿

hold the pixel coordinates at a reference point

CRVAL¡i¿

hold the corresponding WCS coordinates at the reference point

CTYPE¡i¿

name the quantity represented by the WCS axes, together with the projection algorithm used to convert the scaled and rotated pixel coordinates to WCS coordinates.

CD¡i¿_¡j¿

a set of keywords which specify the elements of a matrix. This matrix scales pixel offsets from the reference point into the offsets required as input by the projection algorithm specified by the CTYPE keywords. This matrix specifies the scale and rotation of the image. If there is no rotation the off-diagonal elements of the matrix (*e.g.* CD1_2 and CD2_1) can be omitted.

As an example consider the common case of a simple 2D image of the sky in which north is parallel to the second pixel axis and east parallel to the (negative) first pixel axis. The image scale is 1.2 arc-seconds per pixel on both axes, and the image is presumed to have been obtained with a tangent plane projection. Furthermore, it is known that pixel coordinates (100.5,98.4) correspond to an RA of 11:00:10 and a Dec. of -23:26:02. A suitable set of FITS-WCS header cards could be:

```
CTYPE1  = 'RA---TAN'        / Axis 1 represents RA with a tan projection
CTYPE2  = 'DEC--TAN'        / Axis 2 represents Dec with a tan projection
CRPIX1  = 100.5             / Pixel coordinates of reference point
CRPIX2  = 98.4              / Pixel coordinates of reference point
CRVAL1  = 165.04167         / Degrees equivalent of "11:00:10" hours
CRVAL2  = -23.433889        / Decimal equivalent of "-23:26:02" degrees
CD1_1   = -0.0003333333     / Decimal degrees equivalent of -1.2 arc-seconds
CD2_2   = 0.0003333333      / Decimal degrees equivalent of 1.2 arc-seconds
```

Notes:

- a FITS header card begins with the keyword name starting at column 1, has an equals sign in column 9, and the keyword value in columns 11 to 80.
- string values must be enclosed in single quotes.
- celestial longitude and latitude must both be specified in decimal degrees.
- the CD1_1 value is negative to indicate that RA increases as the first pixel axis decreases.
- the (RA,Dec) coordinates will be taken as ICRS coordinates. For FK5 you should add:

  ```
  RADESYS = 'FK5'
  EQUINOX = 2005.6
  ```

  The EQUINOX value defaults to J2000.0 if omitted. FK4 can also be used in place of FK5, in which case EQUINOX defaults to B1950.0.

Once you have created these FITS-WCS header card strings, you should store them in a FitsChan and then read the corresponding FrameSet from the FitsChan. How to do this is described in §3.4.

Having created the WCS calibration, you may want to store it in a data file. How to do this is described in §3.15).[6]

If the required WCS calibration cannot be described as a set of FITS-WCS headers, then a different approach is necessary. In this case, you should first create a Frame describing pixel coordinates, and store this Frame in a new FrameSet. You should then create a new Frame describing the world coordinate system. This Frame may be a specific subclass of Frame such as a SkyFrame for celestial coordinates, a SpecFrame for spectral coordinates, a Timeframe for time coordinates, or a CmpFrame for a combination of different coordinates. You also need to create a suitable Mapping which transforms pixel coordinates into world coordinates. AST provides many different types of Mappings, all of which can be combined together in arbitrary fashions to create more complicated Mappings. The WCS Frame should then be added into the FrameSet, using the Mapping to connect the WCS Frame with the pixel Frame.

3.14 . . . Modify a WCS Calibration

The usual reason for wishing to modify the WCS calibration associated with a dataset is that the data have been geometrically transformed in some way (here, we will assume a 2-dimensional image dataset). This causes the image features (stars, galaxies, *etc.*) to move with respect to the grid of pixels which they occupy, so that any coordinate systems previously associated with the image become invalid.

To correct for this, it is necessary to set up a Mapping which expresses the positions of image features in the new data grid in terms of their positions in the old grid. In both cases, the grid coordinates we use will have the first pixel centred at (1,1) with each pixel being a unit square.

AST allows you to correct for any type of geometrical transformation in this way, so long as a suitable Mapping to describe it can be constructed. For purposes of illustration, we will assume here that the new image coordinates "xnew" and "ynew" can be expressed in terms of the old coordinates "xold" and "yold" as follows:

```
double xnew, xold, ynew, yold;
double m[ 4 ], z[ 2 ];

...

xnew = xold * m[ 0 ] + yold * m[ 1 ] + z[ 0 ];
ynew = xold * m[ 2 ] + yold * m[ 3 ] + z[ 1 ];
```

where "m" is a 2×2 transformation matrix and "z" represents a shift of origin. This is therefore a general linear coordinate transformation which can represent displacement, rotation, magnification and shear.

In AST, it can be represented by concatenating two Mappings. The first is a MatrixMap, which implements the matrix multiplication. The second is a WinMap, which linearly transforms one coordinate window on to another, but will be used here simply to implement the shift of origin (alternatively, a ShiftMap could have been used in place of a WinMap). These Mappings may be constructed and concatenated as follows:

---

[6] If you are writing the WCS calibration to a FITS file you obviously have the choice of storing the FITS-WCS cards directly.

```
AstCmpMap *newmap;
AstMatrixMap *matrixmap;
AstWinMap *winmap;

...

/* The MatrixMap may be constructed directly from the matrix "m". */
matrixmap = astMatrixMap( 2, 2, 0, m, "" );

/* For the WinMap, we set up the coordinates of the corners of a unit
   square (window) and then the same square shifted by the required
   amount. */
{
   double ina[] = { 0.0, 0.0 };
   double inb[] = { 1.0, 1.0 };
   double outa[] = {       z[ 0 ],       z[ 1 ] };
   double outb[] = { 1.0 + z[ 0 ], 1.0 + z[ 1 ] };

/* The WinMap will then implement this shift. */
   winmap = astWinMap( 2, ina, inb, outa, outb, "" );
}

/* Join the two Mappings together, so that they are applied one after
   the other. */
newmap = astCmpMap( matrixmap, winmap, 1, "" );
```

You might, of course, create any other form of Mapping depending on the type of geometrical transformation involved. For an overview of the Mappings provided by AST, see §2.2, and for a description of the capabilities of each class of Mapping, see its entry in Appendix D. For an overview of how individual Mappings may be combined, see §2.3 (§6 gives more details).

Assuming you have obtained a WCS calibration for your original image in the form of a pointer to a FrameSet, "wcsinfo1" (§3.4), the Mapping created above may be used to produce a calibration for the new image as follows:

```
AstFrameSet *wcsinfo1, *wcsinfo2;

...

/* If necessary, make a copy of the WCS calibration, since we are
   about to alter it. */
wcsinfo2 = astCopy( wcsinfo1 );

/* Re-map the base Frame so that it refers to the new data grid
   instead of the old one. */
astRemapFrame( wcsinfo2, AST__BASE, newmap );
```

This will produce a pointer, "wcsinfo2", to a new FrameSet in which all the coordinate systems associated with your original image are modified so that they are correctly registered with the new image instead.

For more information about re-mapping the Frames within a FrameSet, see §14.4. Also see §14.5 for a similar example to the above, applicable to the case of reducing the size of an image by binning.

3.15 . . . Write a Modified WCS Calibration to a Dataset

If you have modified the WCS calibration associated with a dataset, such as in the example above (§3.14), then you will need to write the modified version out along with any new data.

In the same way as when reading a WCS calibration (§3.4), how you do this will depend on your data system, but we will assume that you wish to generate a set of FITS header cards that can be stored with the data. You should usually make preparations for doing this when you first read the WCS calibration from your input dataset by modifying the example given in §3.4 as follows:

```
AstFitsChan *fitschan1;
AstFrameSet *wcsinfo1;
const char *encode;

...

/* Create an input FitsChan and fill it with FITS header cards. Note,
   if you have all the header cards in a single string, use astPutCards in
   place of astPutFits. */
fitschan1 = astFitsChan( NULL, NULL, "" );
for ( icard = 0; icard < ncard; icard++ ) astPutFits( fitschan1, cards[ icard ], 0 );

/* Note which encoding has been used for the WCS information. */
encode = astGetC( fitschan1, "Encoding" );

/* Rewind the input FitsChan and read the WCS information from it. */
astClear( fitschan1, "Card" );
wcsinfo1 = astRead( fitschan1 );
```

Note how we have added an enquiry to determine how the WCS information is encoded in the input FITS cards, storing a pointer to the resulting string in the "encode" variable. This must be done **before** actually reading the WCS calibration.

(**N.B.** *If you will be making extensive use of astGetC in your program, then you should allocate a buffer and make a copy of this string, because the pointer returned by astGetC will only remain valid for 50 invocations of the function, and you will need to use the Encoding value again later on.)*

Once you have produced a modified WCS calibration for the output dataset (*e.g.* §3.14), in the form of a FrameSet identified by the pointer "wcsinfo2", you can produce a new FitsChan containing the output FITS header cards as follows:

```
AstFitsChan *fitschan2;
AstFrameSet *wcsinfo2;

...

/* Make a copy of the input FitsChan, AFTER the WCS information has
   been read from it. This will propagate all the input FITS header
   cards, apart from those describing the input WCS calibration. */
fitschan2 = astCopy( fitschan1 );
```

```
/* If necessary, make modifications to the cards in "fitschan2"
   (e.g. you might need to change NAXIS1, NAXIS2, etc., to account for
   a change in image size). You probably only need to do this if your
   data system does not provide these facilities itself. */
<details not shown - see below>

/* Alternatively, if your data system handles the propagation of FITS
   header cards to the output dataset for you, then simply create an
   empty FitsChan to contain the output WCS information alone.
fitschan2 = astFitsChan( NULL, NULL, "" );
*/

/* Rewind the new FitsChan (if necessary) and attempt to write the
   output WCS information to it using the same encoding method as the
   input dataset. */
astSet( fitschan2, "Card=1, Encoding=%s", encode );
if ( !astWrite( fitschan2, wcsinfo2 ) ) {

/* If this didn't work (the WCS FrameSet has become too complex), then
   use the native AST encoding instead. */
   astSet( fitschan2, "Encoding=NATIVE" );
   (void) astWrite( fitschan2, wcsinfo2 );
}
```

For details of how to modify the contents of the output FitsChan in other ways, such as by adding, over-writing or deleting header cards, see §16.4, §16.9, §16.8 and §16.13.

Once you have assembled the output FITS cards, you may retrieve them from the FitsChan that contains them as follows:

```
#include <stdio.h>
char card[ 81 ];

...

astClear( fitschan2, "Card" );
while ( astFindFits( fitschan2, "%f", card, 1 ) ) (void) printf( "%s\n", card );
```

Here, we have simply written each card to the standard output stream, but you would obviously replace this with a function invocation to store the cards in your output dataset.

For data systems that do not use FITS header cards, a different approach may be needed, possibly involving use of a Channel or XmlChan (§15) rather than a FitsChan. In the case of the Starlink NDF data format, for example, all of the above may be replaced by a single call to the function ndfPtwcs—see SUN/33. The whole process can probably be encapsulated in a similar way for most data systems, whether they use FITS header cards or not.

For an overview of how to propagate WCS information through data processing steps, see §17.6. For more information about writing WCS information to FitsChans, see §16.5 and §17.7. For information about the options for encoding WCS information in FITS header cards, see §16.1, §17.1, and the description of the Encoding attribute in Appendix C. For a complete understanding of FitsChans and their use with FITS header cards, you should read §16 and §17.

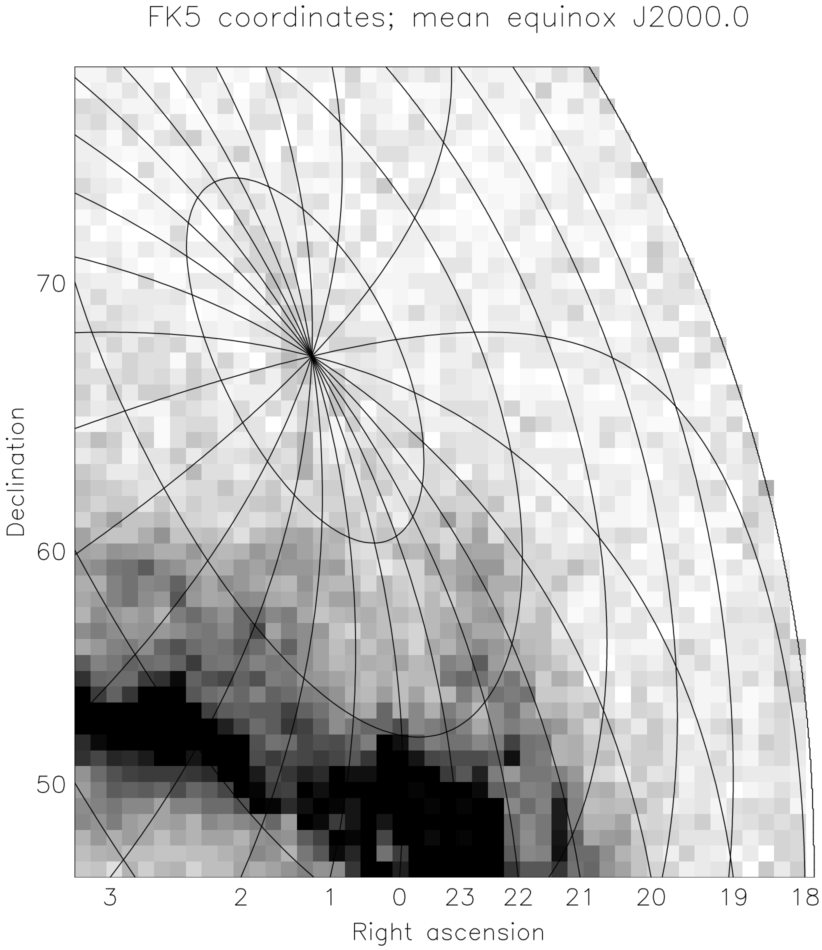


Figure 9: An example of a displayed image with a coordinate grid plotted over it.

3.16 . . . Display a Graphical Coordinate Grid

A common requirement when displaying image data is to plot an associated coordinate grid (*e.g.* Figure 9) over the displayed image. The use of AST in such circumstances is independent of the underlying graphics system, so starting up the graphics system, setting up a coordinate system, displaying the image, and closing down afterwards can all be done using the graphics functions you would normally use.

However, displaying an image at a precise location can be a little fiddly with some graphics systems, and obviously the grid drawn by AST will not be accurately registered with the image unless this is done correctly. In the following template, we therefore illustrate both steps, basing the image display on the C interface to the PGPLOT graphics package.[7] Plotting a coordinate grid with AST then becomes a relatively minor part of what is almost a complete graphics program.

Once again, we assume that a pointer, "wcsinfo", to a suitable FrameSet associated with the image has already been obtained (§3.4).

---

[7] An interface is provided with AST that allows it to use PGPLOT (SUN/15) for its graphics, although interfaces to other graphics systems may also be written.

```
#include "cpgplot.h"
AstPlot *plot;
const float *data;
float hi, lo, scale, x1, x2, xleft, xright, xscale;
float y1, y2, ybottom, yscale, ytop;
int nx, ny;

...

/* Access the image data, which we assume has dimension sizes "nx" and
   "ny", and will be accessed via the "data" pointer.  Also derive
   limits for scaling it, which we assign to the variables "hi" and
   "lo". */
<this stage depends on your data system, so is not shown>

/* Open PGPLOT using the device given by environment variable
   PGPLOT_DEV and check for success. */
if( cpgbeg( 0, " ", 1, 1 ) == 1 ) {

/* Clear the screen and ensure equal scales on both axes. */
   cpgpage();
   cpgwnad( 0.0f, 1.0f, 0.0f, 1.0f );

/* Obtain the extent of the plotting area (not strictly necessary for
   PGPLOT, but possibly for other graphics systems). From this, derive
   the display scale in graphics units per pixel so that the image
   will fit within the display area. */
   cpgqwin( &x1, &x2, &y1, &y2 );
   xscale = ( x2 - x1 ) / nx;
   yscale = ( y2 - y1 ) / ny;
   scale = ( xscale < yscale ) ? xscale : yscale;

/* Calculate the extent of the area in graphics units that the image
   will occupy, so as to centre it within the display area. */
   xleft   = 0.5f * ( x1 + x2 - nx * scale );
   xright  = 0.5f * ( x1 + x2 + nx * scale );
   ybottom = 0.5f * ( y1 + y2 - ny * scale );
   ytop    = 0.5f * ( y1 + y2 + ny * scale );

/* Set up a PGPLOT coordinate transformation matrix and display the
   image data as a grey scale map (these details are specific to
   PGPLOT). */
   {
      float tr[] = { xleft - 0.5f * scale, scale, 0.0f,
                     ybottom - 0.5f * scale, 0.0f, scale };
      cpggray( data, nx, ny, 1, nx, 1, ny, hi, lo, tr );
   }

/* BEGINNING OF AST BIT */
/* ==================== */
/* Store the locations of the bottom left and top right corners of the
   region used to display the image, in graphics coordinates. */
   {
      float gbox[] = { xleft, ybottom, xright, ytop };
/* Similarly, store the locations of the image's bottom left and top
   right corners, in pixel coordinates -- with the first pixel centred
   at (1,1). */
      double pbox[] = { 0.5, 0.5, nx + 0.5, ny + 0.5 };

/* Create a Plot, based on the FrameSet associated with the
   image. This attaches the Plot to the graphics surface so that it
   matches the displayed image. Specify that a complete set of grid
   lines should be drawn (rather than just coordinate axes). */
      plot = astPlot( wcsinfo, gbox, pbox, "Grid=1" );
   }

/* Optionally, we can now set other Plot attributes to control the
   appearance of the grid. The values assigned here use the
   colour/font indices defined by the underlying graphics system. */
   astSet( plot, "Colour(grid)=2, Font(textlab)=3" );

/* Use the Plot to draw the coordinate grid. */
   astGrid( plot );

   <maybe some more AST graphics here>

/* Annul the Plot when finished (or use the astBegin/astEnd technique
   shown earlier). */
   plot = astAnnul( plot );

/* END OF AST BIT */
/* ============== */

/* Close down the graphics system. */
   cpgend();
}
```

Note that once you have set up a Plot which is aligned with a displayed image, you may also use it to generate further graphical output of your own, specified in the image's world coordinate system (such as markers to represent astronomical objects, annotation, *etc.*). There is also a range of Plot attributes which gives control over most aspects of the output's appearance. For details of the facilities available, see §21 and the description of the Plot class in Appendix D.

For details of how to build a graphics program which uses PGPLOT, see §3.3 and the description of the ast_link command in Appendix E.

3.17 . . . Switch to Plot a Different Celestial Coordinate Grid

Once you have set up a Plot to draw a coordinate grid (§3.16), it is a simple matter to change things so that the grid represents a different celestial coordinate system. For example, after creating the Plot with astPlot, you could use:

```
astSet( plot, "System=Galactic" );
```

or:

```
astSet( plot, "System=FK5, Equinox=J2010" );
```

and any axes and/or grid drawn subsequently would represent the new celestial coordinate system you specified. Note, however, that this will only work if the original grid represented celestial coordinates of some kind (see §3.8 for how to determine if this is the case[8]). If it did not, you will get an error message.

For more information about the celestial coordinate systems available, see the descriptions of the System, Equinox and Epoch attributes in Appendix C.

3.18 ... Give a User Control Over the Appearance of a Plot

The idea of using a Plot's attributes to control the appearance of the graphical output it produces (§3.16 and §3.17) can easily be extended to allow the user of a program complete control over such matters.

For instance, if the file "plot.config" contains a series of plotting options in the form of Plot attribute assignments (see below for an example), then we could create a Plot and implement these assignments before producing the graphical output as follows:

```
#include <stdio.h>
#define MAXCHARS 120
FILE *stream;
char line[ MAXCHARS + 2 ];
int base;

...

/* Create a Plot and define the default appearance of the graphical
   output it will produce. */
plot = astPlot( wcsinfo, gbox, pbox,
                "Grid=1, Colour(grid)=2, Font(textlab)=3" );

/* Obtain the value of any Plot attributes we want to preserve. */
base = astGetI( plot, "Base" );

/* Open the plot configuration file, if it exists. Read each line of
   text and use it to set new Plot attribute values. Close the file
   when done. */
if ( stream = fopen( "plot.config", "r" ) ) {
   while ( fgets( line, MAXCHARS + 2, stream ) ) astSet( plot, "%s", line );
   close( stream );
}

/* Restore any attribute values we are preserving. */
astSetI( plot, "Base", base );

/* Produce the graphical output (e.g.). */
astGrid( plot );
```

---

[8] Note that the methods applied to a FrameSet may be used equally well with a Plot.

Notice that we take care that the Plot's Base attribute is preserved so that the user cannot change it. This is because graphical output will not be produced successfully if the base Frame does not describe the plotting surface to which we attached the Plot when we created it.

The arrangement shown above allows the contents of the "plot.config" file to control most aspects of the graphical output produced (including the coordinate system used; the colour, line style, thickness and font used for each component; the positioning of axes and tick marks; the precision, format and positioning of labels; *etc.*) *via* assignments of the form:

```
System=Galactic, Equinox = 2001
Border = 1, Colour( border ) = 1
Colour( grid ) = 2
DrawAxes = 1
Colour( axes ) = 3
Digits = 8
Labelling = Interior
```

For a more sophisticated interface, you could obviously perform pre-processing on this input—for example, to translate words like "red", "green" and "blue" into colour indices, to permit comments and blank lines, *etc.*

For a full list of the attributes that may be used to control the appearance of graphical output, see the description of the Plot class in Appendix D. For a complete description of each individual attribute (*e.g.* those above), see the attribute's entry in Appendix C.

4 An AST Object Primer

The AST library deals throughout with entities called Objects and a basic understanding of how to handle these is needed before you can use the library effectively. If you are already familiar with an object-oriented language, such as C++, few of the concepts should seem new to you. Be aware, however, that AST is designed to be used *via* fairly conventional C and Fortran interfaces, so some things have to be done a little differently.

If you are not already familiar with object-oriented programming, then don't worry—we will not emphasise this aspect more than is necessary and will not assume any background knowledge. Instead, this section concentrates on presenting all the fundamental information you will need, explaining how AST Objects behave and how to manipulate them from conventional C programs.

If you like to read documents from cover to cover, then you can consider this section as an introduction to the programming techniques used in the rest of the document. Otherwise, you may prefer to skim through it on a first reading and return to it later as reference material.

4.1 AST Objects

An AST Object is an entity which is used to store information and Objects come in various kinds, called *classes,* according to the sort of information they hold. Throughout this section, we will make use of a simple Object belonging to the "ZoomMap" class to illustrate many of the basic concepts.

A ZoomMap is an Object that contains a recipe for converting coordinates between two hypothetical coordinate systems. It does this by multiplying all the coordinate values by a constant called the *Zoom factor.* A ZoomMap is a very simple Object which exists mainly for use in examples. It allows us to illustrate the ways in which Objects are manipulated and to introduce the concept of a Mapping—a recipe for converting coordinates—which is fundamental to the way the AST library works.

4.2 Object Creation and Pointers

Let us first consider how to create a ZoomMap. This is done very simply as follows:

```
#include "ast.h"
AstZoomMap *zoommap;

...

zoommap = astZoomMap( 2, 5.0, "" )
```

The first step is to include the header file "ast.h" which declares the interface to the AST library. We then declare a pointer of type AstZoomMap∗ to receive the result and invoke the function astZoomMap to create the ZoomMap. The pattern is the same for all other classes of AST Object—you simply prefix "ast" to the class name to obtain the function that creates the Object and prefix "Ast" to obtain the type of the returned pointer.

These functions are called *constructor functions,* or simply *constructors* (you can find an individual description of all AST functions in Appendix B) and the arguments passed to the constructor are used to initialise the new Object. In this case, we specify 2 as the number of coordinates (*i.e.* we are going to work in a 2-dimensional space) and 5.0 as the Zoom factor to be applied. Note that this is a C double value. We will return to the final argument, an empty string, shortly (§4.6).

The value returned by the constructor is termed an *Object pointer* or, in this case, a *ZoomMap pointer* and is used to refer to the Object. You perform all subsequent operations on the Object by passing this pointer to other AST functions.

4.3 The Object Hierarchy

Now that we have created our first ZoomMap, let us examine how it relates to other kinds of Object before investigating what we can do with it.

We have so far indicated that a ZoomMap is a kind of Object and have also mentioned that it is a kind of Mapping as well. These statements can be represented very simply using the following hierarchy:

```
Object
   Mapping
      ZoomMap
```

which is a way of stating that a ZoomMap is a special class of Mapping, while a Mapping, in turn, is a special class of Object. This is exactly like saying that an Oak is a special form of Tree, while a Tree, in turn, is a special form of Plant. This may seem almost trivial, but before you turn to read something less dull, be assured that it is a very important idea to keep in mind in what follows.

If we look at some of the other Objects used by the AST library, we can see how these are all related in a similar way (don't worry about what they do at this stage):

```
Object
   Mapping
      Frame
         FrameSet
            Plot
      UnitMap
      ZoomMap
   Channel
      FitsChan
      XmlChan
```

Notice that there are several different types of Mapping available (*i.e.* there are classes of Object indented beneath the "Mapping" heading) and, in addition, other types of Object which are not Mappings—Channels for instance (which are at the same hierarchical level as Mappings).

The most specialised Object we have shown here is the Plot (which we will not discuss in detail until §21). As you can see, a Plot is a FrameSet... and a Frame... and a Mapping... and, like everything else, ultimately an Object.

What this means is that you can use a Plot not only for its own specialised behaviour, but also whenever any of these other less-specialised classes of Object is called for. The general rule is that an Object of a particular class may substitute for any of the classes appearing above it in this hierarchy. The Object is then said to *inherit* the behaviour of these higher classes. We can therefore use our ZoomMap whenever a ZoomMap, a Mapping or an Object is called for.

Sometimes, this can lead to some spectacular short-cuts by avoiding the need to break large Objects down in order to access their components. With some practice and a little lateral thinking you should soon be able to spot opportunities for this.

You can find the full *class hierarchy*, as this is called, for the AST library in Appendix A and you may need to refer to it occasionally until you are familiar with the classes you need to use.

4.4 Displaying Objects

Let us now return to the ZoomMap that we created earlier (§4.2) and examine what it's made of. There is a function for doing this, called astShow, which is provided mainly for looking at Objects while you are debugging programs.

If you consult the description of astShow in Appendix B, you will find that it takes a pointer to an Object (of type AstObject$*$) as its argument. Although we have only a ZoomMap pointer available, this is not a problem. If you refer to the brief class hierarchy described above (§4.3), you will see that a ZoomMap is an Object, albeit a specialised one, so it inherits the properties of all Objects and can be substituted wherever an Object is required. We can therefore pass our ZoomMap pointer directly to astShow, as follows:

```
astShow( zoommap );
```

The output from this will appear on the standard output stream and should look like the following:

```
Begin ZoomMap
   Nin = 2
IsA Mapping
   Zoom = 5
End ZoomMap
```

Here, the "Begin" and "End" lines mark the beginning and end of the ZoomMap, while the values 2 and 5 are simply the values we supplied to initialise it (§4.2). These have been given simple names to make them easy to refer to.

The line in the middle which says "IsA Mapping" is a dividing line between the two values. It indicates that the "Nin" value is a property shared by all Mappings, so the ZoomMap has inherited this from its *parent class* (Mapping). The "Zoom" value, however, is specific to a ZoomMap and isn't shared by other kinds of Mappings.

4.5 Getting Attribute Values

We saw above (§4.4) how to display the internal values of an Object, but what about accessing these values from a program? Not all internal Object values are accessible in this way, but many are. Those that are, are called *attributes*. A description of all the attributes used by the AST library can be found in Appendix C.

Attributes come in several data types (character string, integer, boolean and floating point) and there is a standard way of obtaining their values. As an example, consider obtaining the value of the Nin attribute for the ZoomMap created earlier. This could be done as follows:

```
int nin;

...

nin = astGetI( zoommap, "Nin" );
```

Here, the function astGetI is used to extract the attribute value by giving it the ZoomMap pointer and the attribute name (attribute names are not case sensitive, but we have used consistent capitalisation in this document in order to identify them). Remember to use the "ast.h" header file to include the function prototype.

If we had wanted the value of the Zoom attribute, we would probably have used astGetD instead, this being a double version of the same function, for example:

```
double zoom;

...

zoom = astGetD( zoommap, "Zoom" );
```

However, we could equally well have read the Nin value as double, or the Zoom value as an integer, or whatever we wanted.

The data type you want returned is specified simply by replacing the final character of the astGetX function name with C (character string), D (double), F (float), I (int) or L (long). If possible, the value is converted to the type you want. If not, an error message will result. Note that all floating point values are stored internally as double, and all integer values as int. Boolean values are also stored as integers, but only take the values 1 and 0 (for true/false).

4.6 Setting Attribute Values

Some attribute values are read-only and cannot be altered after an Object has been created. The Nin attribute of a ZoomMap (describing the number of coordinates) is like this. It is defined when the ZoomMap is created, but cannot then be altered.

Other attributes, however, can be modified whenever you want. A ZoomMap's Zoom attribute is like this. If we wanted to change it, this could be done simply as follows:

```
astSetD( zoommap, "Zoom", 99.6 );
```

which sets the value to 99.6. As when getting an attribute value (§4.5), you have a choice of which data type you will use to supply the new value. For instance, you could use an integer value, as in:

```
astSetI( zoommap, "Zoom", 99 );
```

and the necessary data conversion would occur. You specify the data type you want to supply simply by replacing the final character of the astSetX function name with C (character string), D (double), F (float), I (int) or L (long). Setting a boolean attribute to any non-zero integer causes it to take the value 1.

An alternative way of setting attribute values for Objects is to use the astSet function (*i.e.* with no final character specifying a data type). In this case, you supply the attribute values in a character string. The big advantage of this method is that you can assign values to several attributes at once, separating them with commas. This also reads more naturally in programs. For example:

```
astSet( zoommap, "Zoom=99.6, Report=1" );
```

would set values for both the Zoom attribute and the Report attribute (about which more shortly—§4.8). You don't really have to worry about data types with this method, as any character representation will do. Note, when using astSet, a literal comma may be included in an attribute value by enclosed the value in quotation marks:

```
 astSet( skyframe, 'SkyRef="12:13:32,-23:12:44"' );
```

Another attractive feature of astSet is that you can build the character string which contains the attribute settings in the same way as when using the C run time library "printf" function. This is most useful when the values you want to set are held in other variables. For example:

```
double zoom = 99.6;
int report = 1;

...

astSet( zoommap, "Zoom=%g, Report=%d", zoom, report );
```

would replace the "%" conversion specifications by the values supplied as additional arguments. Any number of additional arguments may be supplied and the formatting rules are exactly the same as for the C "printf" family of functions. This is a very flexible technique, but does contain one pitfall:

> **Pitfall.** The default precision used by "printf" (and astSet) for floating point values is only 6 decimal digits, corresponding approximately to float on most machines, whereas the AST library stores such values internally as doubles. You should be careful to specify a larger precision (such as DBL_DIG, as defined in <float.h>) when necessary. For example:
>
> ```
> #include <float.h>
>
> ...
>
> astSet( zoommap, "Zoom=%.*g", DBL_DIG, double_value );
> ```

Substituted strings may contain commas and this is a useful way of assigning such strings as attribute values without the comma being interpreted as an assignment separator, for example:

```
astSet( object, "Attribute=%s", "A string, containing a comma" );
```

This is equivalent to using astSetC and one of these two methods should always be used when assigning string attribute values which might potentially contain a comma (*e.g.* strings obtained from an external source). However, you should not attempt to use astSet to substitute strings that contain newline characters, since these are used internally as separators between adjacent attribute assignments.

Finally, a very convenient way of setting attribute values is to do so at the same time as you create an Object. Every Object constructor function has a final character string argument which allows you to do this. Although you can simply supply an empty string, it is an ideal opportunity to initialise the Object to have just the attributes you want. For example, we might have created our original ZoomMap with:

```
zoommap = astZoomMap( 2, 5.0, "Report=1" );
```

and it would then start life with its Report attribute set to 1. The "printf"-style substitution described above may also be used here.

4.7 Testing, Clearing and Defaulting Attributes

You can use the astGetX family of functions (§4.5) to get a value for any Object attribute at any time, regardless of whether a value has previously been set for it. If no value has been set, the AST library will generate a suitable default value.

Often, the default value of an attribute will not simply be trivial (zero or blank) but may involve considerable processing to calculate. Wherever possible, defaults are designed to be real-life, sensible values that convey information about the state of the Object. In particular, they may often be based on the values of other attributes, so their values may change in response to changes in these other attributes. The ZoomMap class that we have studied so far is a little too simple to show this behaviour, but we will meet it later on.

An attribute that returns a default value in this way is said to be *un-set.* Conversely, once an explicit value has been assigned to an attribute, it becomes *set* and will always return precisely that value, never a default.

The distinction between set and un-set attributes is important and affects the behaviour of several key routines in the AST library. You can test if an attribute is set using the function astTest, which returns a boolean (integer) result, as in:

```
if ( astTest( zoommap, "Report" ) ) {
   <the Report attribute is set>
}
```

Once an attribute is set, you can return it to its un-set state using astClear. The effect is as if it had never been set in the first place. For example:

```
astClear( zoommap, "Report" );
```

would ensure that the default value of the Report attribute is used subsequently.

4.8 Transforming Coordinates

We now have the necessary apparatus to start using our ZoomMap to show what it is really for. Here, we will also encounter a routine that is a little more fussy about the type of pointer it will accept.

The purpose of a ZoomMap is to multiply coordinates by a constant zoom factor. To witness this in action, we will first set the Report attribute for our ZoomMap to a non-zero value:

```
astSet( zoommap, "Report=1" );
```

This boolean (integer) attribute, which is present in all Mappings (and a ZoomMap is a Mapping), causes the automatic display of all coordinate values that the Mapping converts. It is not a good idea to leave this feature turned on in a finished program, but it can save a lot of work during debugging.

Our next step is to set up some coordinates for the ZoomMap to work on, using two arrays "xin" and "yin", and two arrays to receive the transformed coordinates, "xout" and "yout". Note that these are arrays of double, as are all coordinate data processed by the AST library:

```
double xin[ 10 ] = { 0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0 };
double yin[ 10 ] = { 0.0, 2.0, 4.0, 6.0, 8.0, 10.0, 12.0, 14.0, 16.0, 18.0 };
double xout[ 10 ];
double yout[ 10 ];
```

We will now use the function astTran2 to transform the input coordinates. This is the most commonly-used (2-dimensional) coordinate transformation function. If you look at its description in Appendix B, you will see that it requires a pointer to a Mapping, so we cannot supply just any old Object pointer, as we could with the functions discussed previously. If we passed it a pointer to an inappropriate Object, an error message would result.

Fortunately, a ZoomMap is a Mapping (Appendix A), so we can use it with astTran2 to transform our coordinates, as follows:

```
astTran2( zoommap, 10, xin, yin, 1, xout, yout );
```

Here, 10 is the number of points we want to transform and the fifth argument value of 1 indicates that we want to transform in the *forward* direction (from input to output).

Because our ZoomMap's Report attribute is set to 1, this will cause the effects of the ZoomMap on the coordinates to be displayed on the standard output stream:

```
(0, 0) --> (0, 0)
(1, 2) --> (5, 10)
(2, 4) --> (10, 20)
(3, 6) --> (15, 30)
(4, 8) --> (20, 40)
(5, 10) --> (25, 50)
(6, 12) --> (30, 60)
(7, 14) --> (35, 70)
(8, 16) --> (40, 80)
(9, 18) --> (45, 90)
```

This shows the coordinate values of each point both before and after the ZoomMap is applied. You can see that each coordinate value has been multiplied by the factor 5 determined by the Zoom attribute value. The transformed coordinates are now stored in the "xout" and "yout" arrays.

If we wanted to transform in the opposite direction, we need simply change the fifth argument of astTran2 from 1 to 0. We can also feed the output coordinates from the above back into the function:

```
astTran2( zoommap, 10, xout, yout, 0, xin, yin );
```

The output would then look like:

```
(0, 0) --> (0, 0)
(5, 10) --> (1, 2)
(10, 20) --> (2, 4)
(15, 30) --> (3, 6)
(20, 40) --> (4, 8)
(25, 50) --> (5, 10)
(30, 60) --> (6, 12)
(35, 70) --> (7, 14)
(40, 80) --> (8, 16)
(45, 90) --> (9, 18)
```

This is termed the *inverse* transformation (we have converted from output to input) and you can see that the original coordinates have been recovered by dividing by the Zoom factor.

4.9 Managing Object Pointers

So far, we have looked at creating Objects and using them in various simple ways but have not yet considered how to get rid of them again.

Every Object consumes various computer resources (principally memory) and should be disposed of when it is no longer required, so as to free up these resources. One way of doing this (not necessarily the best—§4.10) is to *annul* each Object pointer once you have finished with it, using astAnnul. For example:

```
zoommap = astAnnul( zoommap );
```

This indicates that you have finished with the pointer. Since astAnnul always returns the null value AST__NULL (as defined in "ast.h"), the recommended way of using it, as here, is to assign the returned value to the pointer being annulled. This ensures that any attempt to use the pointer again will generate an error message.

In general, this process may not delete the Object, because there may still be other pointers associated with it. However, each Object maintains a count of the number of pointers associated with it and will be deleted if you annul the final pointer. Using astAnnul consistently will therefore ensure that all Objects are disposed of at the correct time. You can determine how many pointers are associated with an Object by examining its (read-only) RefCount attribute.

4.10 AST Pointer Contexts—Begin and End

The use of astAnnul (§4.9) is not completely foolproof, however. Consider the following:

```
astShow( astZoomMap( 2, 5.0, "" ) );
```

This creates a ZoomMap and displays it on standard output (§4.4). Using function invocations as arguments to other functions in this way is very convenient because it avoids the need for intermediate pointer variables. However, the pointer generated by astZoomMap is still active, and since we have not stored its value, we cannot use astAnnul to annul it. The ZoomMap will therefore stay around until the end of the program.

A simple way to avoid this problem is to enclose all use of AST functions between invocations of astBegin and astEnd, for example:

```
astBegin;
astShow( astZoomMap( 2, 5.0, "" ) );
astEnd;
```

When the expansion of astEnd (which is a macro) executes, every Object pointer created since the previous use of astBegin (also a macro) is automatically annulled and any Objects left without pointers are deleted. This provides a simple solution to managing Objects and their pointers, and allows you to create Objects very freely without needing to keep detailed track of each one. Because this is so convenient, we implicitly assume that astBegin and astEnd are used in most of the examples given in this document. Pointer management is not generally shown explicitly unless it is particularly relevant to the point being illustrated.

If necessary, astBegin and astEnd may be nested, like blocks delimited by "{. . . }" in C, to define a series of AST pointer contexts. Each use of astEnd will then annul only those Object pointers created since the matching use of astBegin.

4.11 Exporting, Importing and Exempting AST Pointers

The astExport function allows you to export particular pointers from one AST context (§4.10) to the next outer one, as follows:

```
astExport( zoommap );
```

This would identify the pointer stored in "zoommap" as being required after the end of the current AST context. It causes any pointers nominated in this way to survive the next use of astEnd (but only one such use) unscathed, so that they are available to the next outer context. This facility is not needed often, but is invaluable when the purpose of your astBegin. . . astEnd block is basically to generate an Object pointer. Without this, there is no way of getting that pointer out.

The astImport routine can be used in a similar manner to import a pointer into the current context, so that it is deleted when the current context is closed using astEnd.

Sometimes, you may also want to exempt a pointer from all the effects of AST contexts. You should not need to do this often, but it will prove essential if you ever need to write a library of functions that stores AST pointers as part of its own internal data. Without some form of exemption, the caller of your routines could cause the pointers you have stored to be annulled—thus corrupting your internal data—simply by using astEnd. To avoid this, you should use astExempt on each pointer that you store, for example:

```
astExempt( zoommap );
```

This will prevent the pointer being affected by any subsequent use of astEnd. Of course, it then becomes your responsibility to annul this pointer (using astAnnul) when it is no longer required.

4.12 AST Objects within Multi-threaded Applications

When the AST library is built from source, the build process checks to see if the POSIX threads library ("`pthreads`") is available. If so, appropriate `pthreads` calls are inserted into the AST source code to ensure that AST is thread-safe, and the AST__THREADSAFE macro (defined in the "ast.h" header file) is set to "`1`". If the `pthreads` library cannot be found when AST is built, a working version of the AST library will still be created, but it will not be thread-safe. In this case the AST__THREADSAFE macro will be set to "`0`" in ast.h. The rest of this section assumes that the thread-safe version of AST is being used.

Note, some AST functions call externally specified functions (*e.g.* the source and sink functions used by the Channel class or the graphics primitives functions used by the Plot class). AST does not know whether such functions are thread-safe or not. For this reason, invocations of these functions within a multi-threaded environment are serialised using a mutex in order to avoid two or more threads executing an external function simultaneously.

If an application uses more than one thread, the possibility arises that an Object created by one thread may be accessed by another thread, potentially simultaneously. If any of the threads modifies any aspect of the Object, this could lead to serious problems within the other threads. For this reason, some restrictions are placed on how Objects can be used in a multi-threaded application.

4.12.1 Locking AST Objects for Exclusive Use

The basic restriction is that a thread can only access Objects that it has previously locked for its own exclusive use. If a thread attempts to access any Object that it has not locked, an error is reported.

The astAnnul function is the one exception to this restriction. Pointers for Objects not currently locked by the calling thread can be annulled succesfully using astAnnul. This means that a thread that has finished with an Object pointer can unlock the Object by passing the pointer to astUnlock (so that other threads can use the Object via their own cloned pointers), and can then annul the pointer using astAnnul. Note, however, that an error will be reported by astAnnul if the supplied pointer has been locked by another thread using astLock.

When an Object is created, it is initially locked by the calling thread. Therefore a thread does not need to lock an Object explicitly if it was created in the same thread.

If the Object pointer is then passed to another thread, the first thread must unlock the Object using astUnlock and the second thread must then lock it using astLock.

If a thread attempts to lock an Object that is already locked by another thread, it can choose to report an error immediately or to wait until the Object is available.

The astThread function can be used to determine whether an Object is locked by the running thread, locked by another thread, or unlocked.

If two or more threads need simultaneous access to an Object, a deep copy of the Object should be taken for each thread, using astCopy, and then the copies should be unlocked and passed to the othe threads, which should then lock them. Note, if a thread modifies the Object, the modification will have no effect on the other threads, because the Object copies are independent of each other.

4.12.2 AST Pointer Contexts

Each thread maintains its own set of nested AST contexts, so when astEnd is called, only Objects that are locked by the current thread will be annulled.

If an Object is unlocked by a thread using astUnlock, it is exempted from context handling so that subsequent invocations of astEnd will not cause it to be annulled (this is similar to using astExempt on the Object). When the Object is subsequently locked by another thread using astLock, it will be imported into the context that was active when astLock was called.

4.13 Copying Objects

The AST library makes extensive use of pointers, not only for accessing Objects directly, but also as a means of storing Objects inside other Objects (a number of classes of Object are designed to hold collections of other Objects). Rather than copy an Object in its entirety, a pointer to the interior Object is simply stored in the enclosing Object.

This means that Objects may frequently not be completely independent of each other because, for instance, they both contain pointers to the same sub-Object. In this situation, changing one Object (say assigning an attribute value) may affect the other one *via* the common Object.

It is difficult to describe all cases where this may happen, so you should always be alert to the possibility. Fortunately, there is a simple solution. If you require two Objects to be independent, then simply use astCopy to make a copy of one, *e.g:*

```
AstZoomMap *zoommap1, *zoommap2;

...

zoommap2 = astCopy( zoommap1 );
```

This process will create a true copy of any Object and return a pointer to the copy. This copy will not contain any pointers to any component of the original Object (everything is duplicated), so you can then modify it safely, without fear of affecting either the original or any other Object.

4.14 C Pointer Types

At this point it is necessary to confess to a small amount of deception. So far, we have been passing Object pointers to AST functions in order to perform operations on those Objects. In fact, however, what we were using were not true C functions at all, but merely macros which invoke a related set of hidden functions with essentially the same arguments. In practical terms, this makes very little difference to how you use the functions, as we will continue to call them.[9]

The reason for this deception has to do with the rules for data typing in C. Recall that most AST functions can be used to process Objects from a range of different classes (§4.3). In C, this means passing different pointer types to the same function and most C compilers will not permit this (at least, not without grumbling) because it usually indicates a programming error. In AST, however, it is perfectly safe if done properly. Some way is therefore needed of circumventing the normal compiler checking.

The normal way of doing this in C is with a cast. This approach quickly becomes cumbersome, however, so we have adopted the strategy of wrapping each function in a macro which applies the appropriate cast for you. This means that you can pass pointers of any type to any AST function. For example, in passing a ZoomMap pointer to astShow:

```
AstZoomMap *zoommap;

...

zoommap = astZoomMap( 2, 5.0, "" );
astShow( zoommap );
```

we are exploiting this mechanism to avoid a compiler warning, because the notional type of astShow's parameter is AstObject∗ (not AstZoomMap∗).

We must still guard against programming errors, however, so every pointer's type is checked by the enclosing macro immediately before any AST function executes. This allows pointer mismatches (in the more liberal AST sense—*i.e.* taking account of the class hierarchy, rather than the stricter C sense) to be detected at run-time and a suitable error message will be reported. This message should also identify the line where the error occurs.

A similar strategy is used when pointers are returned by AST functions (*i.e.* as the function result). In this case the pointer is cast to void∗, although we retain the notional pointer type in the function's documentation (*e.g.* Appendix B). This allows you to assign function results to pointer variables without using an explicit cast. For example, the astRead function returns an Object pointer, but might be used to read (say) a ZoomMap as follows:

```
AstChannel *channel;
AstZoomMap *zoommap;

...

zoommap = astRead( channel );
```

---

[9] About the only difference is that you cannot store a pointer to an AST "function" in a variable and use the variable's value to invoke that function again later.

Strictly, there is a C pointer mis-match here, but it is ignored because the operation makes perfect sense to AST.

There is an important exception to this, however, in that constructor functions always return strongly-typed pointers. What we mean by this is that the returned pointer is never implicitly cast to void∗. You must therefore match pointer types when you initially create an Object using its constructor, such as in the following:

```
AstZoomMap *zoommap;

...

zoommap = astZoomMap( 2, 5.0, "" );
```

If the variable receiving the pointer is of a different type, an appropriate cast should be used, as in:

```
AstMapping *mapping;

...

mapping = (AstMapping *) astZoomMap( 2, 5.0, "" );
```

This is an encouragement for you to declare your pointer types consistently, since this is of great benefit to anyone trying to understand your software.

Finally, we should also make one more small confession—AST pointers are not really pointers at all. Although they behave like pointers, the actual "values" stored are not the addresses of C data structures. This means that you cannot de-reference an AST pointer to examine the data within (although you can use astShow instead—§4.4). This is necessary so that AST pointers can be made unique even although several of them might reference the same Object.

4.15 Error Detection

If an error occurs in an AST function (for example, if you supply an invalid argument, such as a pointer to the wrong class of Object), an error message will be written to the standard error stream and the function will immediately return.

To indicate than an error has occurred, an AST *error status* value is used. This integer value is stored internally by AST and is initially clear (*i.e.* set to zero[10] to indicate no error). If an error occurs, it becomes set to a different *error value*, which allows you to detect the error, as follows:

```
zoommap = astZoomMap( 2, 5.0, "Title=My ZoomMap" );
if ( !astOK ) {
   <an error has occurred>
}
```

---

[10] We will assume throughout that the "OK" value is zero, as it currently is. However, a different value could, in principle, be used if the environment in which AST is running requires it. This is why a simple interface is provided to isolate you from the actual value of the error status.

The macro astOK is used to test whether the AST error status is still OK. In this example it would not be, because we have attempted to set a value for the Title attribute of a ZoomMap and a ZoomMap does not have such an attribute. The actual value of the AST error status can be obtained using the astStatus macro, as follows:

```
int status;

...


status = astStatus;
```

A consequence of the AST error status being set is that almost all AST functions will subsequently cease to function and will instead simply return without action. This means that you do not need to use astOK to check for errors very frequently. Instead, you can usually simply invoke a succession of AST functions. If an error occurs in any of them, the following ones will do nothing and you can check for the error at the end, for example:

```
astFunctionA( ... );
astFunctionB( ... );
astFunctionC( ... );
if ( !astOK ) {
   <an error has occurred>
}
```

There are, however, a few functions which do not adhere to this general rule and which will attempt to execute if the AST error status is set. These functions, such as astAnnul, are concerned with cleaning up and recovering resources. For example, in the following:

```
zoommap = astZoomMap( 2, 5.0, "" );

astFunctionX( ... );
astFunctionY( ... );
astFunctionZ( ... );

zoommap = astAnnul( zoommap );
if ( !astOK ) {
   <an error has occurred>
}
```

astAnnul will execute normally in order to recover the resources associated with the ZoomMap that was created earlier, regardless of whether an error has occurred in any of the intermediate functions. Functions which behave in this way are noted in the relevant descriptions in Appendix B.

If a serious error occurs, you will probably want to abort your program, but sometimes you may want to recover and carry on. Because very few AST functions will execute once the AST error status has been set, you must first clear this status by using the astClearStatus macro, as follows:

```
astClearStatus;
```

This will restore the AST error status to its OK value, so that AST functions execute normally again.

Occasionally, you may also need to set the AST error status to an explicit error value (see §15.14 for an example). This is done using astSetStatus and can be used to communicate to AST that an error has occurred in some other item of software, for example:

```
int new_status;

...

astSetStatus( new_status );
```

The effect is that most AST routines will subsequently return without action, just as if an error had occurred within the AST library itself.

4.16 Sharing the Error Status

In some software, it is usual to maintain a single integer error status variable which is accessed by each function as it executes. If an error occurs, this status variable is set and other functions can detect this and take appropriate action.

If you use AST in such a situation, it can be awkward to have a separate internal error status used by AST functions alone. To remedy this, AST is capable of sharing the error status variable used by any other software, so long as they use the same conventions (*i.e.* a C int with the same "OK" value). To enable this facility, you should pass the address of your status variable to astWatch, as follows:

```
int my_status;
int *old_address;

...

old_address = astWatch( &my_status );
```

Henceforth, instead of using its own internal error status variable, AST will use the one you supply, so that it can detect errors flagged by other parts of your software. The address of the original error status variable is returned by astWatch, so you can restore the original behaviour later if necessary.

Note that this facility is not available *via* the Fortran interface to the AST library.

5 Inter-Relating Coordinate Systems (Mappings)

In §4 we used the ZoomMap as an example of a Mapping. We saw how it could be used to transform coordinates from its input to its output and back again (§4.8). We also saw how its behaviour could be controlled by setting various attributes, such as the Zoom factor and the Report attribute that made it display coordinate values as it transformed them.

In this section, we will look at Mappings a bit more thoroughly and explore the behaviour which is common to all the Mappings provided by AST. This is good background for what follows, because many of the Objects we discuss later will also turn out to be Mappings in various disguises.

5.1 The Mapping Class

Before we start, it is worth taking a quick look at the Mapping class as a whole and some of the sub-classes it contains:

```
Mapping
   CmpMap
   DssMap
   GrismMap
   IntraMap
   LutMap
   MathMap
   MatrixMap
   PermMap
   PolyMap
   SlaMap
   SpecMap
   TimeMap
   UnitMap
   WcsMap
   ZoomMap

   Frame
      <various types of Frame>
```

The Frame sub-class has been separated out here because it is covered in detail in §7. We start by looking at the parent class, Mapping.

AST does not provide a function to create a basic Mapping (*i.e.* the astMapping constructor does not exist). This is because the Mapping class itself is "virtual" and basic Mappings are of no use in themselves. The Mapping class serves simply to contain the various specialised Mappings that exist. However, it provides more than just a convenient heading for them because it bestows all classes of Mapping with common properties (*e.g.* attributes) and behaviour. By examining the Mapping class, we are therefore examining the things that all other Mappings have in common.

5.2 The Mapping Model

The concept of a Mapping was illustrated in Figure 1. It is a black box which you can supply with a set of coordinate values in return for a set of transformed coordinates. The two sets are termed *input* and *output* coordinates. You can also go back the other way and transform output coordinates back into input coordinates, as we saw in §4.8.

5.3 Input and Output Coordinate Numbers

In general, the number of coordinates you feed into a Mapping to represent a single point need not be the same as the number that comes out. Often these numbers will be the same, and often they will both equal 2 (because 2-dimensional coordinate systems are common), but this needn't necessarily be the case.

The number of coordinates required to specify an input point is represented by the integer attribute Nin and the number required to specify an output point is represented by Nout. These are read-only attributes common to all Mappings. Generally, their values are fixed when a Mapping is created.

In §4.2, we saw how the Nin attribute for a ZoomMap was initialised by the call to the constructor function astZoomMap which created it. In this case, the Nout attribute was not needed and it implicitly took the same value as Nin, but we could have enquired about its value had we wanted, as follows:

```
#include "ast.h"
AstZoomMap *zoommap;
int nout;

...

nout = astGetI( zoommap, "Nout" );
```

5.4 Forward and Inverse Transformations

We stated earlier that a Mapping may be used to transform coordinates either from input to output, or *vice versa.* These are termed its *forward* and *inverse* transformations.

This statement was not quite accurate, however, because in general Mappings are only **potentially** capable of working in both directions. In practice, coordinate transformation may only be feasible in one direction or the other because some functions are not easily inverted (they may be multi-valued, for instance). Allowance must be made for this, so each Mapping has two read-only boolean (integer) attributes, TranForward and TranInverse, which indicate whether each transformation is available.

A transformation is available if the corresponding attribute is non-zero, otherwise it is not.[11] If you enquire about the value of these attributes, a value of 0 or 1 is returned. Attempting to use a Mapping to apply a transformation which is not available will result in an error.

---

[11] Most of the Mappings provided by the AST library work in both directions, although the LutMap can behave otherwise.

5.5 Inverting Mappings

An important attribute, common to all Mappings, is the Invert flag. This is a boolean (integer) attribute that can be assigned a new value at any time. If it is non-zero, it has the effect of interchanging the Mapping's input and output coordinates and the Mapping is then said to be *inverted.* By default, the Invert attribute is zero.

There is no magic in this. There is no fancy arithmetic involved in inverting mathematical functions, for instance. The Invert flag is simply a switch that interchanges a Mapping's input and output ports. If it is non-zero, the Mapping's Nin and Nout attributes are swapped, its TranForward and TranInverse attributes are swapped, and when you ask for what was once the forward transformation you get the inverse transformation instead (and *vice versa*). When you return the Invert attribute to zero, or clear it, the Mapping returns to its original behaviour.

Often, the actual value of the Invert attribute is unimportant and you simply wish to invert its boolean sense, so that what was the Mapping's input becomes its output and *vice versa.* This is most easily accomplished using astInvert, as follows:

```
AstMapping *mapping;

...

astInvert( mapping );
```

If the Mapping you have happens to be the wrong way around, astInvert allows you to correct the problem.

5.6 Finding the Rate of Change of a Mapping Output

The astRate function can be used to find the rate of change of any Mapping output with respect to any Mapping input, at a given input position. The method used produces good accuracy (typically a relative error of 10E-10 or less) but may require the Mapping to be evaluated 100 or more times. An estimate of the second derivative is also produced by this function.

5.7 Reporting Coordinate Transformations

We have already seen (§4.8) how the boolean (integer) Report attribute of a Mapping works. If it is non-zero, the operation of transforming a set of coordinates will result in a report being written to standard output. This will display the coordinate values before and after transformation. It can save considerable time during program development by eliminating the need to add loops and output statements to your program.

In a finished program, however, you should be careful that the Report attribute is not set to a non-zero value unless you want to see the output (there may often be rather a lot of this!). To help prevent unwanted output being produced by accident, the Report attribute is unusual in that its value is not preserved when a Mapping is copied using astCopy (§4.13). Instead, it reverts to its default of zero (*i.e.* un-set) in the copy. It also reverts to zero when a Mapping is written out, *e.g.* to a file using a Channel (§15).

5.8 Handling Missing (Bad) Coordinate Values

Even when coordinates can, in principle, be transformed in either direction by a Mapping, there may still be instances where specific coordinate values cannot be handled. For example, the Mapping may be mathematically intractable (*e.g.* singular) in certain places, or it may map a subset of one space on to another, so that some points in one space are not represented in the other. Sky projections often show this behaviour, since it is quite common to project only half of the celestial sphere on to two dimensions, omitting points on the opposite side of the sky. There are many other examples.

To indicate when coordinates cannot be transformed, for whatever reason, AST substitutes a special output coordinate value given by the macro AST__BAD (as defined in the "ast.h" header file). Before making use of coordinates generated by any of the AST transformation functions, therefore, you may need to check for the presence of this value.

Because coordinates with the value AST__BAD can be generated in this way, all other AST functions are also capable of recognising this value and handling it appropriately. The coordinate transformation functions do this by propagating any missing input coordinate information through to their output. This means that if you supply coordinates with the value AST__BAD, the returned coordinates are also likely to contain this value. Here, for example, is what happens if you use a ZoomMap (with Zoom factor 5) to transform such a set of coordinates:

```
(0, 0) --> (0, 0)
(<bad>, 2) --> (<bad>, 10)
(2, 4) --> (10, 20)
(3, 6) --> (15, 30)
(4, <bad>) --> (20, <bad>)
(5, 10) --> (25, 50)
(<bad>, <bad>) --> (<bad>, <bad>)
(7, 14) --> (35, 70)
(8, 16) --> (40, 80)
(9, 18) --> (45, 90)
```

The AST__BAD value is represented by the string "<bad>". This is a case of "garbage in, garbage out" but at least it's consistent garbage that you can recognise!

Note how the presence of the AST__BAD value in one input dimension does not necessarily result in the loss of information for all output dimensions. Sometimes, such loss will be unavoidable, but in general an attempt is made to preserve information as far as possible. The exact behaviour will depend on the Mapping involved.

5.9 Example—the UnitMap

The UnitMap is the simplest of Mappings. It is a null Mapping. Its purpose is simply to copy coordinate values, unaltered, from its input to its output and *vice versa.*

A UnitMap has no additional attributes beyond those of a basic Mapping. Its Nin and Nout attributes are always equal and are specified by the first argument supplied to its constructor. For example:

```
AstUnitMap *unitmap;

...

unitmap = astUnitMap( 2, "" );
```

will create a UnitMap that copies 2-dimensional coordinates. Inverting a UnitMap has no effect beyond changing the value of its Invert attribute.

The main use of a UnitMap is to allow a Mapping to be supplied when one is required (as an argument to a function, for example) but you wish it to leave coordinate values unchanged.

5.10 Example—the PermMap

The PermMap is a rather more complicated Mapping than we have met previously. Its purpose is to change the order, or number, of coordinates. It is also able to substitute fixed values for coordinates.

To illustrate its action, suppose our input coordinates are denoted by (x_1, x_2, x_3, x_4) in a 4-dimensional space and suppose our output coordinates are to be (x_4, x_1, x_2, x_3). Our PermMap, therefore, should rotate the coordinate values by one position.

To create such a PermMap, we first set up two integer arrays. One of these, "outperm", controls the selection of input coordinates for use in the output and the other, "inperm", controls selection of output coordinates for use in the input:

```
int outperm[ 4 ] = { 4, 1, 2, 3 };
int inperm[ 4 ] = { 2, 3, 4, 1 };
```

Note that the numbers we store in these arrays are the indices of the coordinates that we want to select. We have chosen these so that the forward and inverse transformations will perform complementary permutations on the coordinates.

The PermMap is then created by passing these arrays to its constructor, as follows:

```
AstPermMap *permmap;

...

permmap = astPermMap( 4, inperm, 4, outperm, NULL, "" );
```

Note that we specify the number of input and output coordinates separately, but set both to 4 in this example. The resulting PermMap would have the following effect when used to transform coordinates:

```
Forward:
   (1, 2, 3, 4) --> (4, 1, 2, 3)
   (2, 4, 6, 8) --> (8, 2, 4, 6)
   (3, 6, 9, 12) --> (12, 3, 6, 9)
   (4, 8, 12, 16) --> (16, 4, 8, 12)
```

```
      (5, 10, 15, 20) --> (20, 5, 10, 15)

   Inverse:
      (4, 1, 2, 3) --> (1, 2, 3, 4)
      (8, 2, 4, 6) --> (2, 4, 6, 8)
      (12, 3, 6, 9) --> (3, 6, 9, 12)
      (16, 4, 8, 12) --> (4, 8, 12, 16)
      (20, 5, 10, 15) --> (5, 10, 15, 20)
```

If the number of input and output coordinates are unequal so, also, will be the size of the "outperm" and "inperm" arrays. This means, however, that we cannot fill them with coordinate indices so that they perform complementary permutations, because one transformation will lose information (discard a coordinate) that the other cannot recover. To give an example, consider the following:

```
int outperm[ 3 ] = { 4, 3, 2 };
int inperm[ 4 ] = { -1, 3, 2, 1 };
double con[ 1 ] = { 99.004 };
```

In this case, the forward transformation will change (x_1, x_2, x_3, x_4) into (x_4, x_3, x_2) and will discard x_1. The inverse transformation restores the original coordinate order, but has no value to assign to the first coordinate. In this case, the number entered in the "inperm" array is -1.

This negative value indicates that the coordinate value should be obtained by addressing the first element of the "con" array (*i.e.* element zero). This array, ignored in the previous example, may then be used to supply a value for the missing coordinate.

The constructor function:

```
permmap = astPermMap( 4, inperm, 3, outperm, con, "" );
```

will then create a PermMap with the following effect when used to transform coordinates:

```
Forward:
   (1, 2, 3, 4) --> (4, 3, 2)
   (2, 4, 6, 8) --> (8, 6, 4)
   (3, 6, 9, 12) --> (12, 9, 6)
   (4, 8, 12, 16) --> (16, 12, 8)
   (5, 10, 15, 20) --> (20, 15, 10)

Inverse:
   (4, 3, 2) --> (99.004, 2, 3, 4)
   (8, 6, 4) --> (99.004, 4, 6, 8)
   (12, 9, 6) --> (99.004, 6, 9, 12)
   (16, 12, 8) --> (99.004, 8, 12, 16)
   (20, 15, 10) --> (99.004, 10, 15, 20)
```

The "con" array may contain more than one value if necessary and may be addressed by both the "inperm" and "outperm" arrays using coordinate indices -1, -2, -3, *etc.* to refer to the first, second, third, *etc.* elements.

If there is no suitable replacement value that can be supplied *via* the "con" array, a value of zero may be entered into the "outperm" and/or "inperm" arrays. This causes the value AST__BAD to be used for the affected coordinate (as defined in the "ast.h" header file), thus indicating a missing coordinate value (§5.8).

The principle use for a PermMap lies in matching a coordinate system to a data array where there is a choice of storage order for the data. PermMaps are also useful for discarding unwanted coordinates so as to reduce the number of dimensions, such as when selecting a "slice" from a multi-dimensional array.

6 Compound Mappings (CmpMaps)

We now turn to a rather special form of Mapping, the CmpMap. The Mappings we have considered so far have been atomic, in the sense that they perform pre-defined elementary transformations. A CmpMap, however, is a compound Mapping. In essence, it is a framework for containing other Mappings and its purpose is to allow those Mappings to work together in various combinations while appearing as a single Object. A CmpMap's behaviour is therefore not pre-defined, but is determined by the other Mappings it contains.

6.1 Combining Mappings in Series

Consider a simple example based on two 2-dimensional coordinate systems. Suppose that to convert from one to the other we must swap the coordinate order and multiply both coordinates by 5, so that the coordinates (x_1, x_2) transform into $(5x_2, 5x_1)$. This can be done in two stages:

1. Apply a PermMap (§5.10) to swap the coordinate order.
2. Apply a ZoomMap (§4.8) to multiply both coordinate values by the constant 5.

The PermMap and ZoomMap are then said to operate *in series,* because they are applied sequentially (*c.f.* Figure 2). We can create a CmpMap that applies these Mappings in series as follows:

```
#include "ast.h"
AstCmpMap *cmpmap;
AstPermMap *permmap;
AstZoomMap *zoommap;

...

/* Create the individual Mappings. */
{
   int inperm[ 2 ] = { 2, 1 };
   int outperm[ 2 ] = { 2, 1 };
   permmap = astPermMap( 2, inperm, 2, outperm, NULL, "" );
}
zoommap = astZoomMap( 2, 5.0, "" )

/* Combine them in series. */
cmpmap = astCmpMap( permmap, zoommap, 1, "" );

/* Annul the individual Mapping pointers. */
permmap = astAnnul( permmap );
zoommap = astAnnul( zoommap );
```

Here, the third argument (1) of the constructor function astCmpMap indicates "in series".

When used to transform coordinates in the forward direction, the resulting CmpMap will apply the first component Mapping (the PermMap) and then the second one (the ZoomMap). When transforming in the inverse direction, it will apply the second one (in the inverse direction) and then the first one (also in the inverse direction). In general, although not in this particular example, the order in which the two component Mappings are supplied is significant. Clearly, also, the Nout attribute (number of output coordinates) for the first Mapping must equal the Nin attribute (number of input coordinates) for the second one.

6.2 Combining Mappings in Parallel

Connecting two Mappings in series (§6.1) is not the only way of combining them. The alternative, *in parallel,* involves applying the two Mappings at once but on different subsets of the coordinate values.

Consider, for example, a set of 3-dimensional coordinates and suppose we wish to transform them by swapping the first two coordinate values and multiplying the final one by 5, so that (x_1, x_2, x_3) transforms into $(x_2, x_1, 5x_3)$. Again, we can perform each of these steps individually using exactly the same PermMap and ZoomMap as used earlier (§6.1). In this case, however, these individual Mappings are applied in parallel (*c.f.* Figure 3).

Creating a CmpMap for this purpose is also very simple:

```
cmpmap = astCmpMap( permmap, zoommap, 0, "" );
```

The only difference is that the third argument of astCmpMap is now zero, meaning “in parallel”.

As before, the order in which the two component Mappings are supplied is significant. The first one acts on the lower-numbered input coordinate values (however many it needs) and produces the lower-numbered output coordinates, while the second Mapping acts on the higher-numbered input coordinates (however many remain) and generates the remaining higher-numbered output coordinates. When the CmpMap transforms coordinates in the inverse direction, both component Mappings are applied to the same coordinates, but in the inverse direction.

Note that the Nin and Nout attributes of the component Mappings (*i.e.* the numbers of input and output coordinates) will sum to give the Nin and Nout attributes of the overall CmpMap.

6.3 The Component Mappings

A CmpMap does not store copies of its component Mappings, but simply holds pointers to them. In the example above (§6.1), we were free to annul the individual Mapping pointers after creating the CmpMap because the pointers held internally by the CmpMap increased the reference count (RefCount attribute) of each component Mapping by one. The individual components are therefore not deleted by astAnnul, but retained until the CmpMap itself is deleted and annuls the pointers it holds. Consistent use of astAnnul (§4.9) and/or pointer contexts (§4.10) will therefore ensure that all Objects are deleted at the appropriate time.

Note that access to a CmpMap’s component Mappings is not generally available unless pointers to them are retained when the CmpMap is created. If such pointers are retained, then subsequent modifications to the individual components can be used to indirectly modify the behaviour of the overall CmpMap.

There is an important exception to this, however, because a CmpMap retains a copy of the initial Invert flag settings of each of its components and uses these in order to ignore any subsequent external changes. This means that you may invert either component Mapping before inserting it into a CmpMap and need not worry if you un-invert it again later. The CmpMap's behaviour will not be affected by the later action.

6.4 Creating More Complex Mappings

Because a CmpMap is itself a Mapping, any existing CmpMap can substitute (§4.3) as a component Mapping when constructing a new CmpMap using astCmpMap. This has the effect of nesting one CmpMap inside another and opens up many new possibilities. For example, combining three Mappings in series can be accomplished as follows:

```
AstMapping *map1, *map2, *map3;

...

cmpmap = astCmpMap( map1, astCmpMap( map2, map3, 1, "" ), 1, "" );
```

The way in which the individual component Mappings are grouped within the nested CmpMaps is not usually important.

A similar technique can be used to combine multiple Mappings in parallel and, of course, mixed series and parallel combinations are also possible (Figure 4). There is no built-in limit to how many CmpMaps may be nested in this way, so this mechanism provides an indefinitely extensible method of building complex Mappings out of the elemental building blocks provided by AST.

In practice, you might not need to construct such complex CmpMaps yourself very frequently, but they will often be returned by AST routines. Nested CmpMaps underlie the library's entire ability to represent a wide range of different coordinate transformations.

6.5 Example—Transforming Between Two Calibrated Images

Consider, as a practical example of CmpMaps, two images of the sky. Suppose that for each image we have a Mapping which converts from pixel coordinates to a standard celestial coordinate system, say FK5 (J2000.0). If we wish to inter-compare these images, we can do so by using this celestial coordinate system to align them. That is, we first convert from pixel coordinates in the first image into FK5 coordinates and we then convert from FK5 coordinates into pixel coordinates in the second image.

If "mapa" and "mapb" are pointers to our two original Mappings, we could form a CmpMap which transforms directly between the pixel coordinates of the first and second images by combining these Mappings, as follows:

```
AstCmpMap *alignmap;
AstMapping *mapa, *mapb;

...

astInvert( mapb );
alignmap = astCmpMap( mapa, mapb, 1, "" );
astInvert( mapb );
```

Here, we have used astInvert (§5.5) to invert "mapb" before inserting it into the CmpMap because, as supplied, it converted in the wrong direction. Afterwards, we invert it again to return it to its original state. The CmpMap, however, will ignore this subsequent change (§6.3).

The forward transformation of the resulting CmpMap will now transform from pixel coordinates in the first image to pixel coordinates in the second image, while its inverse transformation will convert in the opposite direction.

6.6 Over-Complex Compound Mappings

While a CmpMap provides a very flexible way of constructing arbitrarily complex Mappings (§6.4), it unfortunately also provides an opportunity for representing simple Mappings in complex ways. Sometimes, unnecessary complexity can be difficult to avoid but can obscure important simplifications.

Consider the example above (§6.5), in which we inter-related two images of the sky *via* a CmpMap. If the two images turned out to be simply offset from each other by a shift along each pixel axis, then this approach would align them correctly, but it would be inefficient. This is because it would introduce unnecessary and expensive transformations to and from an intermediate celestial coordinate system, whereas a simple shift of pixel origin would suffice.

Recognising that a simpler and more efficient solution exists obviously requires a little more than simply joining two Mappings end-to-end. We must also determine whether the resulting CmpMap is more complex than it needs to be, *i.e.* contains redundant information. If it is, we then need a way to simplify it.

The problem is not always just one of efficiency, however. Sometimes we may also need to know something about the actual form a Mapping takes—*i.e.* the nature of the operations it performs. Unnecessary complexity can obscure this, but such complexity can easily accumulate during normal data processing.

For example, a Mapping that transforms pixel coordinates into positions on the sky might be repeatedly modified as changes are made to the shape and size of the image. Typically, on each occasion, another Mapping will be concatenated to reflect what has happened to the image. This could soon make it difficult to discern the overall nature of the transformation from the complex CmpMap that accumulates. If only shifts of origin were involved on each occasion, however, they could be combined into a single shift which could be represented much more simply.

Suppose we now wanted to represent our image's celestial coordinate calibration using FITS conventions (§17). This requires AST to determine whether the Mapping which relates pixel coordinate to sky positions conforms to the FITS model (for example, whether it is equivalent to applying a single set of shifts and scale factors followed by a map projection). Clearly, there is an important use here for some means of simplifying the internal structure of a CmpMap.

6.7 Simplifying Compound Mappings

The ability to simplify compound Mappings is provided by the astSimplify function. This function encapsulates a number of heuristics for converting Mappings, or combinations of Mappings within a CmpMap, into simpler, equivalent ones. When applied to a CmpMap, astSimplify tries to reduce the number of individual Mappings within it by merging neighbouring component

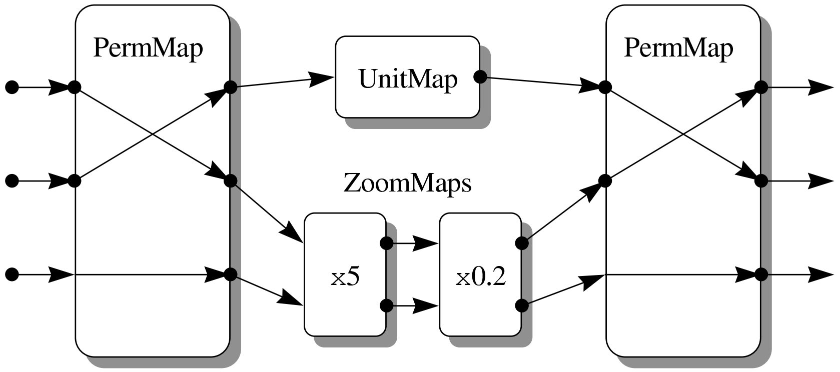


Figure 10: An over-complex compound Mapping, consisting of PermMaps, ZoomMaps and a UnitMap, which can be simplified to become a single UnitMap. The enclosing nested CmpMaps have been omitted for clarity.

Mappings together. It will do this with both series and parallel combinations of Mappings, or both, and will handle CmpMaps nested to any depth (§6.4).

To illustrate how astSimplify works, consider the combination of Mappings shown in Figure 10. If this were contained in a CmpMap, it could be simplified as follows:

```
AstMapping *simpler;

...

simpler = astSimplify( cmpmap );
```

In this case, the result would be a simple 3-dimensional UnitMap (the identity Mapping). To reach this conclusion, astSimplify will have made a number of deductions, roughly as follows:

1. The two 2-dimensional ZoomMaps in series are equivalent to a single ZoomMap with a combined Zoom factor of unity. This, in turn, is equivalent to a 2-dimensional UnitMap.
2. This UnitMap in parallel with the other 1-dimensional UnitMap is equivalent to a single 3-dimensional UnitMap. This UnitMap, sandwiched between any other pair of Mappings, can then be eliminated.
3. The remaining two PermMaps in series are equivalent to a single 3-dimensional PermMap. When these are combined, the resulting PermMap is found to be equivalent to a 3-dimensional UnitMap.

This example is a little contrived, but illustrates how astSimplify can deal with even quite complicated compound Mappings through a series of incremental simplifications. Where possible, this will result in either a simpler compound Mapping or, if feasible, an atomic (non-compound) Mapping, as here. If no simplification is possible, astSimplify will just return a pointer to the original Mapping.

Although astSimplify cannot identify every simplification that is theoretically possible, sufficient rules are included to deal with the most common and important cases.

7 Representing Coordinate Systems (Frames)

An AST Frame is an Object that is used to represent a coordinate system. Contrast this with a Mapping (§5), which is used to describe how to convert between coordinate systems. The two concepts are complementary and we will see how they work together in §13.

In this section we will discuss only basic Frames, which represent Cartesian coordinate systems. More specialised types of Frame (*e.g.* the SkyFrame, which represents celestial coordinate systems, and the SpecFrame, which represents spectral coordinate systems) are covered later (§8 and §9) and, naturally, inherit the properties and behaviour of the simple Frames discussed here.

7.1 The Frame Model

The best way to think about a Frame is like the frame that you would plot around a graph. In two dimensions, you would have an "x" and a "y" axis, a title on the graph and labels on the axes, together with an indication of the physical units being plotted. The values marked along each axis would be formatted in a human-readable way. The frame around a graph therefore defines a coordinate space within which you can locate points, draw lines, calculate distances, *etc.*

An AST Frame works in much the same way, embodying all of these concepts and a few more. It also allows any number of axes, which means that a Frame can represent coordinate systems with any number of dimensions. You specify how many when you create it.

Remember that the basic Frame we are considering here is completely general. It knows nothing of celestial coordinates, for example, and all its axes are equivalent. It can be adapted to describe any general purpose Cartesian coordinate system by setting its attributes, such as its Title and axis Labels, *etc.* to appropriate values.

7.2 Creating a Frame

Creating a Frame is straightforward and follows the usual pattern:

```
#include "ast.h"
astFrame *frame;

...

frame = astFrame( 2, "" );
```

The first argument of the astFrame constructor function specifies the number of axes which the Frame should have.

7.3 Using a Frame as a Mapping

We should briefly point out that the Frame we created above (§7.2) is also a Mapping (§5.1) and therefore inherits the properties and behaviour common to other Mappings.

One way to see this is to set the Frame's Report attribute (inherited from the Mapping class) to a non-zero value and pass the Frame pointer to a coordinate transformation function, such as astTran2.

```
double xin[ 5 ] = { 0.0, 1.0, 2.0, 3.0, 4.0, 5.0 };
double yin[ 5 ] = { 0.0, 2.0, 4.0, 6.0, 8.0, 10.0 };
double xout[ 5 ];
double yout[ 5 ];

...

astSet( frame, "Report=1" );
astTran2( frame, 5, xin, yin, 1, xout, yout );
```

The resulting output might then look like this:

```
(1, 2) --> (1, 2)
(2, 4) --> (2, 4)
(3, 6) --> (3, 6)
(4, 8) --> (4, 8)
(5, 10) --> (5, 10)
```

This is not very exciting because a Frame implements an identity transformation just like a UnitMap (§5.9). However, it illustrates that a Frame can be used as a Mapping and that its Nin and Nout attributes are both equal to the number of Frame axes.

When we consider more specialised Frames (*e.g.* §13), we will see that using them as Mappings can be very useful indeed.

7.4 Frame Axis Attributes

Frames have a number of attributes which can take multiple values, one for each axis. These separate values are identified by appending the axis number in parentheses to the attribute name. For example, the Label(1) attribute is a character string containing the label which appears on the first axis.

Axis attributes are accessed in the same way as all other attributes (§4.5, §4.6 and §4.7). For example, the Label on the second axis might be obtained as follows:

```
const char *label;

...

label = astGetC( frame, "Label(2)" );
```

Other attribute access functions (astSetX, astTest and astClear) may also be applied to axis attributes in the same way.

If the axis number is stored in a program variable, then its value must be formatted to generate a suitable attribute name before using this to access the attribute itself. For example, the following will print out the Label value for each axis of a Frame:

```
#include <stdio.h>
char name[ 18 ];
int iaxis, naxes;

...

naxes = astGetI( frame, "Naxes" );
for ( iaxis = 1; iaxis <= naxes; iaxis++ ) {
   (void) sprintf( name, "Label(%d)", iaxis );
   label = astGetC( frame, name );
   (void) printf( "Label %2d: %s\n", iaxis, label );
}
```

Note the use of the Naxes attribute to determine the number of Frame axes.

The output from this might look like the following:

```
Label  1: Axis 1
Label  2: Axis 2
```

In this case, the Frame's default axis Labels have been revealed as rather un-exciting. Normally, you would set much more useful values, typically when you create the Frame—perhaps something like:

```
frame = astFrame( 2, "Label(1)=Offset from centre of field,"
                     "Unit(1) =mm,"
                     "Label(2)=Transmission coefficient,"
                     "Unit(2) =%" );
```

Here, we have also set the (character string) Unit attribute for each axis to describe the physical units represented on that axis. All the attribute assignments have been combined into a single string, separated by commas.

7.5 Frame Attributes

We will now briefly outline the various attributes associated with a Frame (this is, of course, in addition to those inherited from the Mapping class). We will not delve too deeply into the details of each attribute, for which you should consult the appropriate description in Appendix C. Instead, we aim simply to sketch the range of facilities available:

Naxes
A read-only integer giving the number of Frame axes.

Title
A string describing the coordinate system which the Frame represents.

Label(axis)
A label string for each axis.

Unit(axis)
A string describing the physical units on each axis. You can choose whether to make this attribute "active" or "passive" (using astSetActiveUnit). If active, its value will be taken into account when finding the Mapping between two Frames (*e.g.* a scaling of 0.001 would be used to connect two axis with units of "km" and "m"). If passive, its value is ignored. Its use is described in more detail in §7.14.

Symbol(axis)
A string containing a "short form" symbol (*e.g.* like "X" or "Y") used to represent the quantity plotted on each axis.

Digits/Digits(axis)
The preferred number of digits of precision to be used when formatting values for display on each axis.

Format(axis)
A string containing a *format specifier* which determines exactly how values should be formatted for display on each axis (§7.6). If this attribute is unset, the formatting is based on the Digits value, otherwise the Format string over-rides the Digits value.

Direction(axis)
A boolean (integer) value which indicates in which direction each axis should be plotted. If it is non-zero (the default), the axis should be plotted in the conventional direction—*i.e.* increasing to the right for the abscissa and increasing upwards for the ordinate. If it is zero, the axis should be plotted in reverse. This attribute is provided as a hint only and programs are free to ignore it if they wish.

Domain
A character string which identifies the *physical domain* to which the Frame's coordinate system applies. The primary purpose of this attribute is to prevent unwanted conversions from occurring between coordinate systems which are not related. Its use is described in more detail in §7.12.

System
A character string which identifies the specific coordinate system used to describe positions within the physical domain represented by the Frame. For a simple Frame, this attribute currently has a fixed value of "Cartesian", but could in principle be extended to include options such as "Polar", "Cylindrical", *etc.* More specialised Frames such as the SkyFrame, TimeFrame and SpecFrame, redefine the allowed values to be appropriate to the domain which they describe. For instance, the SkyFrame allows values such as "FK4" and "Galactic", and the SpecFrame allows values such as "frequency" and "wavelength".

Epoch
This value is used to qualify a coordinate system by giving the moment in time when the coordinates are correct. Usually, this will be the date of observation. The Epoch value is important in cases where coordinates systems move with respect to each other over time. An example of two such coordinate systems are the FK4 and FK5 celestial coordinate systems.

ObsLon
Specifies the longitude of the observer (assumed to be on the surface of the

earth). The basic Frame class does not use this value, but specialised subclasses may. For instance, the SpecFrame class uses it to calculate the relative velocity of the observer and the centre of the earth for use in converting between standards of rest.

ObsLat

Specifies the latitude of the observer. Use in conjunction with ObsLon.

There are also some further Frame attributes, not described above, which are important when Frames are used as templates to search for other Frames. Their use goes beyond the present discussion.

7.6 Formatting Axis Values

The coordinate values associated with each axis of a Frame are stored (*e.g.* within your program) as double values. The Frame class therefore provides a function, astFormat, to convert these values into formatted strings for display:

```
const char *string
double value;

...

string = astFormat( frame, iaxis, value );
```

Here, the astFormat function is passed a Frame pointer, the number of an axis ("iaxis") and a double precision value to format ("value"). It returns a pointer to character string containing the formatted value.

By default, the formatting applied will be determined by the Frame's Digits attribute and will normally display results with seven digits of precision (corresponding approximately to the C "float" data type on many machines). Setting a different Digits value, however, allows you to adjust the precision as necessary to suit the accuracy of the coordinate data you are processing. If finer control is needed, it is also possible to set a Digits value for each individual axis by appending an axis number to the attribute name (*e.g.* "Digits(2)"). If this is done, it over-rides the effect of the Frame's main Digits value for that axis.

Even finer control is possible by setting the (character string) Format attribute for a Frame axis. The string given should contain a C *format specifier* which explicitly determines how the values on that axis should be formatted. This will over-ride the effects of any Digits value[12]. Any valid "printf" format specifier may be used so long as it consumes exactly one double value.

When setting Format values, remember that the "%" which appears in the format specifier may need to be doubled to "%%" if you are using a function (such as astSet) which interprets "printf" format specifiers itself.

It is recommended that you use astFormat whenever you display formatted coordinate values, even although you could format them yourself using "sprintf". This is because it puts the Frame

---

[12] The exception to this rule is that if the Format value includes a precision of ".*", then Digits will be used to determine the actual precision used.

in control of formatting. When you start to handle more elaborate Frames (representing, say, celestial coordinates), you will need different formatting methods. This approach delivers them without any change to your software.

You should also consider regularly using the astNorm function, described below (§7.7), for any values that will be made visible to the user of your software.

7.7 Normalising Frame Coordinates

The function astNorm is provided to cope with the fact that some coordinate systems do not extend indefinitely in all directions. Some may have boundaries, outside which coordinates are meaningless, while others wrap around on themselves, so that after a certain distance you return to the beginning again (coordinate systems based on circles and spheres, for instance). A basic Frame has no such complications, but other more specialised Frames (such as SkyFrames, representing the celestial sphere—§8) do.

The role played by astNorm is to *normalise* any arbitrary set of coordinates by converting them into a set which is "within bounds", interpreted according to the particular Frame in question. For example, on the celestial sphere, a right ascension value of 24 hours or more can have a suitable multiple of 24 hours subtracted without affecting its meaning and astNorm would perform this task. Similarly, negative values of right ascension would have a multiple of 24 hours added, so that the result lies in the range zero to 24 hours. The coordinates in question are modified in place by astNorm, as follows:

```
double point[ 2 ];

...

astNorm( frame, point );
```

If the coordinates supplied are initially OK, as they would always be with a basic Frame, then they are returned unchanged.

Because the main purpose of astNorm is to convert coordinates into the preferred range for human consumption, its use is almost always appropriate immediately before formatting coordinate values for display using astFormat (§7.6). Even if the Frame in question does not restrict the range of coordinates, so that astNorm does nothing, using it will allow you to process other more specialised Frames, where normalisation is important, without changing your software.

7.8 Reading Formatted Axis Values

The process of converting a formatted coordinate value for a Frame axis, such as might be produced by astFormat (§7.6), back into a numerical (double) value ready for processing is performed by astUnformat. However, although this process is essentially the inverse of that performed by astFormat, there are a number of additional difficulties that must be addressed in practice.

The main use for astUnformat is in reading formatted coordinate values which have been entered by the user of a program, or read from a file. As such, we can rarely assume that the values are neatly formatted in the way that astFormat would produce. Instead, it is usually desirable to allow considerable flexibility in the form of input that can be accommodated, so as to permit "free-format" data input by the user. In addition, we may need to extract individual coordinate values embedded in other textual data.

Underlying these requirements is the root difficulty that the textual format used to represent a coordinate value will depend on the class of Frame we are considering. For example, for a basic Frame, astUnformat may have to read a value like "1.25e-6", whereas for a more specialised Frame representing celestial coordinates it may have to handle a value like "-07d 49m 13s". Of course, the format might also depend on which axis is being considered.

Ideally, we would like to write software that can handle any kind of Frame. However, this makes it a little more difficult to analyse textual input data to extract individual coordinate values, since we cannot make assumptions about how the values are formatted. It would not be safe, for example, simply to assume that the values being read are separated by white space. This is not just because they might be separated by some other character, but also because celestial coordinate values might themselves contain spaces. In fact, to be completely safe, we cannot make any assumptions about how a formatted coordinate value is separated from the surrounding text, except that it should be separated in some way which is not ambiguous.

This is the very basic assumption upon which astUnformat works. It is invoked as follows:

```
int n;

...

n = astUnformat( frame, iaxis, string, &value );
```

It is supplied with a Frame pointer ("frame"), the number of an axis ("iaxis") and a character string to be read ("string"). If it succeeds in reading a value, astUnformat returns the resulting coordinate to the address supplied *via* the final argument ("&value"). The returned function value indicates how many characters were read from the string in order to obtain this result.

The string is read as follows:

1. Any white space at the start is skipped over.

2. Further characters are considered, one at a time, until the next character no longer matches any of the acceptable forms of input (given the characters that precede it). The longest sequence of characters which matches is then considered "read".

3. If a suitable sequence of characters was read successfully, it is converted into a coordinate value which is returned. Any white space following this sequence is then skipped over and the total number of characters consumed is returned as the function value.

4. If the sequence of characters read is empty, or insufficient to define a coordinate value, then the string does not contain a value to read. In this case, the read is aborted and astUnformat returns a function value of zero and no coordinate value. However, it returns without error.

Note that failing to read a coordinate value does not constitute an error, at least so far as astUnformat is concerned. However, an error can occur if the sequence of characters read appears to have the correct form but cannot be converted into a valid coordinate value. Typically, this will be because it violates some constraint, such as a limit on the value of one of its fields. The resulting error message will give details.

For any given Frame axis, astUnformat does not necessarily always use the same algorithm for converting the sequence of characters it reads into a coordinate value. This is because some forms of input (particularly free-format input) can be ambiguous and might be interpreted in several ways depending on the context. For example, the celestial longitude "12:34:56.7" could represent an angle in degrees or a right ascension in hours. To decide which to use, astUnformat may examine the Frame's attributes and, in particular, the appropriate Format(axis) string which is used by astFormat when formatting coordinate values (§7.6). This is done in order that astFormat and astUnformat should complement each other—so that formatting a value and then un-formatting it will yield the original value, subject to any rounding error.

To give a simple (but crucially incomplete!) example, consider reading a value for the axis of a basic Frame, as follows:

```
n = astUnformat( frame, iaxis, " 1.5e6   -99.0", &value );
```

astUnformat will skip over the initial space in the string supplied and then examine each successive character. It will accept the sequence "1.5e6" as input, but reject the space which follows because it does not form part of the format of a floating point number. It will then convert the characters "1.5e6" into a coordinate value and skip over the three spaces which follow them. The returned function value will therefore be 9, equal to the total number of characters consumed. This result may be used to address the string during a subsequent read, so as to commence reading at the start of "-99.0".

Most importantly, however, note that if the user of a program mistakenly enters the string " 1.5r6…" instead of " 1.5e6…", a coordinate value of 1.5 and a function result of 4 will be returned, because the "r" would prematurely terminate the attempt to read the value. Because this sort of mistake does not automatically result in an error but can produce incorrect results, it is **vital** to check the returned function value to ensure that the expected number of characters have been read.[13] For example, if the string is expected to contain exactly one value, and nothing else, then the following would suffice:

```
n = astUnformat( frame, iaxis, string, &value );
if ( astOK ) {
   if ( string[ n ] || !n ) {
      <error in input data>
   } else {
      <value read correctly>
   }
}
```

If astUnformat does not detect an error itself, we check that it has read to the end-of-string and consumed at least one character (which traps the case of a zero-length input string). If this reveals an error, the value of "n" indicates where it occurred.

---

[13] Anyone who seriously uses the C run time library "scanf" function will know about the need for this check!

Another common requirement is to obtain a position by reading a list of coordinates from a string which contains one value for each axis of a Frame. We assume that the values are separated in some unambiguous manner, perhaps using white space and/or some unspecified single-character separator. The choice of separator is up to the data supplier, who must choose it so as not to conflict with the format of the coordinate values, but our software does not need to know what it is. The following is a template algorithm for reading data in this form:

```
const char *s;
double values[ 10 ];

...

/* Initialise a string pointer. */
s = string;

/* Obtain the number of Frame axes and loop through them. */
naxes = astGetI( frame, "Naxes" );
for ( iaxis = 1; iaxis <= naxes; iaxis++ ) {

/* Attempt to read a value for this axis. */
   n = astUnformat( frame, iaxis, s, &values[ iaxis - 1 ] );

/* If nothing was read and this is not the first axis or the
   end-of-string, try stepping over a separator and reading again. */
   if ( !n && ( iaxis > 1 ) && *s )
      n = astUnformat( frame, iaxis, ++s, &values[ iaxis - 1 ] );

/* Quit if nothing was read, otherwise move on to the next value. */
   if ( !n ) break;
   s += n;
}

/* Check for possible errors. */
if ( astOK ) {
   if ( *s || !n ) {
      <error in input data>
   } else {
      <values read correctly>
   }
}
```

In this case, "s" will point to the location of any input error.

Note that this algorithm is insensitive to the precise format of the data and will therefore work with any class of Frame and any reasonably unambiguous input data. For example, here is a range of suitable input data for a 3-dimensional basic Frame:

```
1 2.5 3
3.1,3.2,3.3
1.5, 2.6, -9.9e2
-1.1+0.4-1.8
    .1/.2/.3
 44.0 ; 55.1 -14
```

7.9 Permuting Frame Axes

Once a Frame has been created, it is not possible to change the number of axes it contains, but it is possible to change the order in which these axes occur. To do so, an integer *permutation array* is filled with the numbers of the axes so as to specify the new order, *e.g:*

```
int perm[ 2 ] = { 2, 1 };
```

In this case, the axes of a 2-dimensional Frame could be interchanged by passing this permutation array to the astPermAxes function. That is, an (x_1, x_2) coordinate system would be changed into an (x_2, x_1) coordinate system by:

```
astPermAxes( frame, perm );
```

If the axes are permuted more than once, the effects are cumulative. You are, of course, not restricted to Frames with only two axes.

7.10 Selecting Frame Axes

An alternative to changing the number of Frame axes, which is not allowed, is to create a new Frame by selecting axes from an existing one. The method of doing this is very similar to the way astPermAxes is used (§7.9), in that we supply an integer array filled with the numbers of the axes we want, in their new order. In this case, however, the number of array elements need not equal the number of Frame axes.

For example, we could select axes 3 and 2 (in that order) from a 3-dimensional Frame as follows:

```
astFrame *frame1, *frame2;
astMapping *mapping;
int pick[ 2 ] = { 3, 2 };

...

frame2 = astPickAxes( frame1, 2, pick, &mapping );
```

This would return a pointer to a 2-dimensional Frame ("frame2") which contains the information associated with axes 3 and 2, in that order, from the original Frame ("frame1"). The original Frame is not altered by this process. Beware, however, that the axis information may still be shared by both Frames, so if you wish to alter either of them independently you may first need to use astCopy (§4.13) to make an independent copy.

In addition to the new Frame pointer, astPickAxes will also return a pointer to a new Mapping *via* its fourth argument (you may supply a NULL pointer as an argument if you do not want this Mapping). This Mapping will inter-relate the two Frames. By this we mean that its forward transformation will convert coordinates originally in the coordinate system represented by "frame1" into that represented by "frame2", while its inverse transformation will convert in the opposite direction. In this particular case, the Mapping would be a PermMap (§5.10) and would implement the following transformations:

```
Forward:
   (1, 2, 3) --> (3, 2)
   (2, 4, 6) --> (6, 4)
   (3, 6, 9) --> (9, 6)
   (4, 8, 12) --> (12, 8)
   (5, 10, 15) --> (15, 10)

Inverse:
   (3, 2) --> (<bad>, 2, 3)
   (6, 4) --> (<bad>, 4, 6)
   (9, 6) --> (<bad>, 6, 9)
   (12, 8) --> (<bad>, 8, 12)
   (15, 10) --> (<bad>, 10, 15)
```

This is our first introduction to the idea of inter-relating pairs of Frames *via* a Mapping, but this will assume a central role later on.

Note that when using astPickAxes, it is also possible to request more axes than there were in the original Frame. This will involve selecting axes from the original Frame that do not exist. To do this, the corresponding axis number (in the "pick" array) should be set to zero and the effect is to introduce an additional new axis which is not derived from the original Frame. This axis will have default values for all its attributes. You will need to do this because astPickAxes does not allow you to select any of the original axes more than once.[14]

7.11 Calculating Distances, Angles and Offsets

Some complementary functions are provided for use with Frames to allow you to perform geometric operations without needing to know the nature of the coordinate system represented by the Frame.

Functions can be used to find the distance between two points, and to offset a specified distance along a line joining two points, *etc.* In essence, these define the metric of the coordinate space which the Frame represents. In the case of a basic Frame, this is a Cartesian metric.

The first of these functions, astDistance, returns a double distance value when supplied with the Frame coordinates of two points. For example:

```
double dist;
double point1[ 2 ] = { 0.0, 0.0 };
double point2[ 2 ] = { 1.0, 1.0 };

...

dist = astDistance( frame, point1, point2 );
```

This calculates the distance between the origin (0,0) and a point at position (1,1). In this case, the result, as you would expect, is $\sqrt{2}$. However, this is only true for the Cartesian coordinate

---

[14] It will probably not be obvious why this restriction is necessary, but consider creating a Frame with one longitude axis and two latitude axes. Which latitude axis should be associated with the longitude axis?

system which a basic Frame represents. In general, astDistance will calculate the geodesic distance between the two points, so that with a more specialised Frame (such as a SkyFrame, representing the celestial sphere) a great-circle distance might be returned.

The astOffset function is really the inverse of astDistance. Given two points in a Frame, it calculates the coordinates of a third point which is offset a specified distance away from the first point along the geodesic joining it to the second one. For example:

```
double point1[ 2 ] = { 0.0, 0.0 };
double point2[ 2 ] = { 1.0, 1.0 };
double point3[ 2 ];

...

astOffset( frame, point1. point2, 0.5, point3 );
```

This would fill the "point3" array with the coordinates of a point which is offset 0.5 units away from the origin (0,0) in the direction of the position (1,1). Again, this is a simple result in a Cartesian Frame, as varying the offset will trace out a straight line. On the celestial sphere, however (*e.g.* using a SkyFrame), it would trace out a great circle.

The functions astAxDistance and astAxOffset are similar to astDistance and astOffset, except that the curves which they use as "straight lines" are not geodesics, but curves parallel to a specified axis[15]. One reason for using these functions is to deal with the cyclic ambiguity of longitude and latitude axes.

The astOffset2 function is similar to astOffset, but instead of using the geodesic which passes through two positions, it uses the geodesic which passes at a given position angle through the starting position.

Position angles are always measured from the positive direction of the second Frame axis to the required line, with positive angles being in the same sense as rotation from the positive direction of the second axis to the positive direction of the first Frame axis. This definition applies to all classes of Frame, including SkyFrame. The default ordering of axes in a SkyFrame makes the second axis equivalent to north, and so the definition of position angle given above corresponds to the normal astronomical usage, "from north, through east". However, it should be remembered that it is possible to permute the axes of a SkyFrame (or indeed any Frame), so that north becomes axis 1. In this case, an AST "position angle" would be the angle "from east, through north". Always take the axis ordering into account when deriving an astronomical position angle from an AST position angle.

Within a Cartesian coordinate system, the position angle of a geodesic (*i.e.* a straight line) is constant along its entire length, but this is not necessarily true of other coordinate systems. Within a spherical coordinate system, for instance, the position angle of a geodesic will vary along its length (except for the special cases of a meridian and the equator). In addition to returning the required offset position, the astOffset2 function returns the position angle of the geodesic at the offset position. This is useful if you want to trace out a path which involves turning through specified angles. For instance, tracing out a rectangle in which each side is a geodesic involves turning through 90 degrees at the corners. To do this, use astOffset2 to

---

[15] For instance, a line of constant Declination is not a geodesic

calculate the position of each corner, and then add (or subtract) 90 degrees from the position angle returned by astOffset2.

The astAngle function calculates the angle subtended by two points, at a third point. If used with a 2-dimensional Frame the returned angle is signed to indicate the sense of rotation (clockwise or anti-clockwise) in taking the "shortest route" from the first point to the second. If the Frame has more than 2 axes, the result is un-signed and is always in the range zero to π.

The astAxAngle function is similar to astAngle, but the "reference direction", from which angles are measured, is a specified axis.

The astResolve function resolves a given displacement within a Frame into two components, parallel and perpendicular to a given reference direction.

The displacement is specified by two positions within the Frame; the starting and ending positions. The reference direction is defined by the geodesic curve passing through the starting position and a third specified position. The lengths of the two components are returned, together with the position on the reference geodesic which is closest to the third supplied point.

7.12 The Domain Attribute

The Domain attribute is one of the most important properties of a Frame, although the concept it expresses can sometimes seem a little subtle. We will introduce it here, but its true value will probably not become apparent until later (§14.2).

To understand the need for the Domain attribute, consider using different Frames to represent the following different coordinate systems associated with a CCD image:

1. A coordinate system based on pixel numbers.
2. Positions on the CCD chip, measured in μm.
3. Positions in the focal plane of the telescope, measured in mm.
4. A celestial coordinate system, measured in radians.

If we had two such CCD images, we might legitimately want to align them pixel-for-pixel (*i.e.* using the coordinate system based on pixel numbers) in order to, say, divide by a flat-field exposure. We might similarly consider aligning them using any of the other coordinate systems so as to achieve different results. For example, we might consider merging separate images from a CCD mosaic by using focal plane positions.

It would obviously not be legitimate, however, to directly compare positions in one image measured in pixels with positions in the other measured in mm, nor to equate chip positions in μm with sky coordinates in radians. If we wanted to inter-compare these coordinates, we would need to do it indirectly, using other information based on the experimental set-up. For instance, we might need to know the size of the pixels expressed in mm and the orientation of the CCD chip in the focal plane.

Note that it is not simply the difference in physical units which prevents certain coordinates from being directly inter-compared (because the appropriate unit scaling factors could be included without any additional information). Neither is it the fact that different coordinate systems are in use (because we could legitimately inter-compare two different celestial coordinate systems without any extra information). Instead, it is the different nature of the coordinate spaces to which these coordinate systems have been applied.

We normally express this by saying that the coordinate systems apply to different *physical domains*. Although we may establish *ad hoc* relationships between coordinates in different physical domains, they are not intrinsically related to each other and we need to supply extra information before we can convert coordinates between them.

In AST, the role of the (character string) Domain attribute is to assign Frames to their respective physical domains. The way it operates is as follows:

- Coordinate systems which apply to the same physical domain (*i.e.* whose Frames have the same Domain value) can be directly inter-compared.

  If the domain has several coordinate systems associated with it (*e.g.* the celestial sphere), then a coordinate conversion may be involved. Otherwise, coordinate values may simply be equated.

- Coordinate systems which apply to different physical domains (*i.e.* whose Frames have different Domain values) cannot be directly inter-compared.

  If any relationship does exist between such coordinate systems—and it need not—then additional information must be supplied in order to establish the relationship between them in any particular case. We will see later (§13) how to establish such relationships between Frames in different domains.

With the basic Frames we are considering here, each physical domain only has a single (Cartesian) coordinate system associated with it, so that if two such Frames have the same Domain value, their coordinate systems will be identical and may simply be equated. With more specialised Frames, however, more than one coordinate system may apply to each domain. In such cases, a coordinate conversion may need to be performed.

When a basic Frame is created, its Domain attribute defaults to an empty string. This means that all such Frames belong to the same (null) domain by default and therefore describe the same unspecified physical coordinate space. In order to assign a Frame to a different domain, you simply need to set its Domain value. This is normally most conveniently done when it is created, as follows:

```
frame1 = astFrame( 2, "Domain=CCD_CHIP,"
                      "Unit(1)=micron,"
                      "Unit(2)=micron" );
frame2 = astFrame( 2, "Domain=FOCAL_PLANE,"
                      "Unit(1)=mm,"
                      "Unit(2)=mm" );
```

Here, we have created two Frames in different physical domains. Although their coordinate values all have units of length, they cannot be directly inter-compared (because their axes may be rotated with respect to each other, for instance).

All Domain values are automatically converted to upper case and white space is removed, but there are no other restrictions on the names you may use to label different physical domains. From a practical point of view, however, it is worth following a few conventions (§7.13).

7.13 Conventions for Domain Names

When choosing a value for the Domain attribute of a Frame, it obviously makes sense to avoid generic names which might clash with those used for similar (but subtly different!) purposes by other programmers. If you are developing software for an instrument, for example, and want to identify an instrumental coordinate system, then it is sensible to add a distinguishing prefix. For instance, you might use <INST>_FOCAL_PLANE, where <INST> (*e.g.* an acronym) identifies your instrument.

For some purposes, however, a standard choice of Domain name is desirable so that different items of software can communicate. For this purpose, the following Domain names are reserved by AST and the use recommended below should be carefully observed:

GRAPHICS

Identifies the coordinate space used by an underlying computer graphics system to specify plotting operations. Typically, when performing graphical operations, AST is used to define additional coordinate systems which are related to these "native" graphical coordinates. Plotting may be carried out in any of these coordinate systems, but the GRAPHICS domain identifies the native coordinates through which AST communicates with the underlying graphics system.

GRID

Identifies the instantaneous *data grid* used to store and handle data, together with an associated coordinate system. In this coordinate system, the first element stored in an array of data always has a coordinate value of unity at its centre and all elements have unit extent. This applies to all dimensions.

If data are copied or transformed to a new data grid (by whatever means), or a subset of the original grid is extracted, then the same rules apply to the copy or subset. Its first element therefore has GRID coordinate values of unity at its centre. Note that this means that GRID coordinates remain attached to the first element of the data grid and not to its data content (*e.g.* the features in an image).

PIXEL

Identifies an array of pixels and an associated *pixel-based* coordinate system which is related to the GRID coordinate system (above) simply by a shift of origin along each axis. This shift may be integral, fractional, positive, negative or zero. The data elements retain their unit extent along each axis.

Because the amount of shift is unspecified, the PIXEL domain is distinct from the GRID domain. The relationship between them contains a degree of uncertainty, such as typically arises from the different conventions used by different software systems. For instance, in some software the first pixel is regarded as being centred at (1,1), while in other software it is at (0.5,0.5). In addition, some software packages implement a "pixel origin" which allows pixel coordinates to start at an arbitrary value.

The GRID domain (which corresponds with the pixel-numbering convention used by FITS) is a special case of the PIXEL domain and avoids this uncertainty. In general, additional information is required in order to convert from one to the other.

SKY
Identifies the domain which contains all equivalent celestial coordinate systems. Because these are represented in AST by SkyFrames (§8), it should be no surprise that the default Domain value for a SkyFrame is SKY. Since there is only one sky, you probably won't need to change this very often.

SPECTRUM
Identifies the domain used to describe positions within an electro-magnetic spectrum. The AST SpecFrame (§9) class describes positions within this domain, allowing a wide range of different coordinate systems to be used (frequency, wavelength, *etc*). The default Domain value for a SpecFrame is SPECTRUM.

TIME
Identifies the domain used to describe moments in time. The AST TimeFrame class describes positions within this domain, allowing a wide range of different coordinate systems and timescales to be used. The default Domain value for a TimeFrame is TIME.

Although we have drawn a necessary distinction here between the GRID and PIXEL domains, we will continue to refer in general terms to image "pixels" and "pixel coordinates" whenever this distinction is not important. This should not be taken to imply that the GRID convention for numbering pixels is excluded—in fact, it is usually to be preferred (at the level of data handling being discussed in this document) and we recommend it.

7.14 The Unit Attribute

Each axis of a Frame has a Unit attribute which holds the physical units used to describe positions on the axis. The index of the axis to which the attribute refers should normally be placed in parentheses following the attribute name ("Unit(2)" for instance). However, if the Frame has only a single axis, then the axis index can be omitted.

In versions of AST prior to version 2.0, the Unit attribute was nothing more than a descriptive string intended purely for human readers—no part of the AST system used the Unit string for any purpose (other than inclusion in axis labels produced by the Plot class). In particular, no account was taken of the Unit attribute when finding the Mapping between two Frames. Thus if the conversion between a pair of 1-dimensional Frames representing velocity was found (using astConvert) the returned Mapping would always be a UnitMap, even if the Unit attributes of the two Frames were "km/h" and "m/s". This behaviour is referred to below as a *passive* Unit attribute.

As of AST version 2.0, a facility exists which allows the Unit attribute to be *active*; that is, differences in the Unit attribute may be taken into account when finding the Mapping between two Frames. In order to minimise the risk of breaking older software, the *default* behaviour of simple Frames and SkyFrames is unchanged from previous versions (*i.e.* they have passive Unit attributes). However, the new functions astSetActiveUnit and astGetActiveUnit allow this default behaviour to be changed. The SpecFrame and TimeFrame classes *always* have an active Unit attribute (attempts to change this are ignored).

For instance, consider the above example of two 1-dimensional Frames describing velocity. These Frames can be created as follows:

```
AstFrame *frame1, *frame2;
frame1 = astFrame( 1, "Domain=VELOCITY,Unit=km/h" );
frame2 = astFrame( 1, "Domain=VELOCITY,Unit=m/s" );
```

By default, these Frames have passive Unit attributes, and so an attempt to find a Mapping between them would ignore the difference in their Unit attributes and return a unit Mapping. To avoid this, we indicate that we want these Frames to have *active* Unit attributes, as follows:

```
astSetActiveUnit( frame1, 1 );
astSetActiveUnit( frame2, 1 );
```

If we then find the Mapping between them as follows:

```
AstFrameSet *cvt;
...
cvt = astConvert( frame1, frame2, "" );
```

the Mapping contained within the FrameSet returned by astConvert will be a one-dimensional ZoomMap which simply scales its input (a velocity in km/h) by a factor of 0.278 to create its output (a velocity in m/s).

In fact we need not have set the Unit attribute active in "frame1" since the behaviour of astConvert is determined by its "to" Frame (the second Frame parameter).

7.14.1 The Syntax for Unit Strings

Conversion between units systems relies on the use of a specific syntax for the Unit attribute. If the value of the Unit attribute does not conform to this syntax, then an error will be reported if an attempt is made to use it to determine an inter-unit Mapping (this will never happen if the Unit attribute is *passive*).

The adopted syntax is that described in FITS-WCS paper I "Representation of World Coordinate in FITS" by Greisen & Calabretta. We distinguish here between "basic" units and "derived" units: derived units are defined in terms of other units (either derived or basic), whereas basic units have no such definitions. Derived units may be represented by their own *symbol* (*e.g.* "Jy"—the Jansky) or by a *mathematical expression* which combines other symbols and constants to form a definition of the unit (*e.g.* "km/s"—kilometres per second). Unit symbols may be prefixed by a string representing a standard multiple or sub-multiple.

In addition to the unit symbols listed in FITS-WCS Paper I, any other arbitrary unit symbol may be used, with the proviso that it will not be possible to convert between Frames using such units. The exception to this is if both Frames refer to the same unknown unit string. For instance, an axis with unknown unit symbol "flop" *could* be converted to an axis with unit "Mflop" (Mega-flop).

Unit symbols (optionally prefixed with a multiple or sub-multiple) can be combined together using a limited range of mathematical operators and functions, to produce new units. Such expressions may also contain parentheses and numerical constants (these may optionally use "scientific" notation including an "E" character to represent the power of 10).

The following tables list the symbols for the basic and derived units which may be included in a units string, the standard prefixes for multiples and sub-multiples, and the strings which may be used to represent mathematical operators and functions.

Basic units

Quantity	Symbol	Full Name
length	m	metre
mass	g	gram
time	s	second
plane angle	rad	radian
solid angle	sr	steradian
temperature	K	Kelvin
electric current	A	Ampere
amount of substance	mol	mole
luminous intensity	cd	candela

7.14.2 Side-effects of Changing the Unit attribute

If an Axis has an active Unit attribute, changing its value (either by setting a new value or by clearing it so that the default value is re-instated) may cause the Label and Symbol attributes to be changed accordingly. For instance, if an Axis has Unit, Label and Symbol of "Hz", "Frequency" and "nu", then changing its Unit attribute to "log(Hz)" will cause AST to change its Label and Symbol to "log(Frequency)" and "Log(nu)". These changes are only made if the Unit attribute is active, and a Mapping can be found from the old units to the new units. On the other hand, changing the Unit from "Hz" to "MHz" would not cause any change to the Label or Symbol attributes.

Derived units			
Quantity	Symbol	Full Name	Definition
area	barn	barn	1.0E-28 m**2
area	pix	pixel	
area	pixel	pixel	
electric capacitance	F	Farad	C/V
electric charge	C	Coulomb	A s
electric conductance	S	Siemens	A/V
electric potential	V	Volt	J/C
electric resistance	Ohm	Ohm	V/A
energy	J	Joule	N m
energy	Ry	Rydberg	13.605692 eV
energy	eV	electron-Volt	1.60217733E-19 J
energy	erg	erg	1.0E-7 J
events	count	count	
events	ct	count	
events	ph	photon	
events	photon	photon	
flux density	Jy	Jansky	1.0E-26 W /m**2 /Hz
flux density	R	Rayleigh	1.0E10/(4*PI) photon.m**-2 /s/sr
flux density	mag	magnitude	
force	N	Newton	kg m/s**2
frequency	Hz	Hertz	1/s
illuminance	lx	lux	lm/m**2
inductance	H	Henry	Wb/A
length	AU	astronomical unit	1.49598E11 m
length	Angstrom	Angstrom	1.0E-10 m
length	lyr	light year	9.460730E15 m
length	pc	parsec	3.0867E16 m
length	solRad	solar radius	6.9599E8 m
luminosity	solLum	solar luminosity	3.8268E26 W
luminous flux	lm	lumen	cd sr
magnetic field	G	Gauss	1.0E-4 T
magnetic flux	Wb	Weber	V s
mass	solMass	solar mass	1.9891E30 kg
mass	u	unified atomic mass unit	1.6605387E-27 kg
magnetic flux density	T	Tesla	Wb/m**2
plane angle	arcmin	arc-minute	1/60 deg
plane angle	arcsec	arc-second	1/3600 deg
plane angle	mas	milli-arcsecond	1/3600000 deg
plane angle	deg	degree	pi/180 rad
power	W	Watt	J/s
pressure, stress	Pa	Pascal	N/m**2
time	a	year	31557600 s
time	d	day	86400 s
time	h	hour	3600 s
time	yr	year	31557600 s
time	min	minute	60 s
	D	Debye	1.0E-29/3 C.m

Prefixes for multiples & sub-multiples					
Sub-multiple	Name	Prefix	Sub-multiple	Name	Prefix
10^{-1}	deci	d	10	deca	da
10^{-2}	centi	c	10^{2}	hecto	h
10^{-3}	milli	m	10^{3}	kilo	k
10^{-6}	micro	u	10^{6}	mega	M
10^{-9}	nano	n	10^{9}	giga	G
10^{-12}	pico	p	10^{12}	tera	T
10^{-15}	femto	f	10^{15}	peta	P
10^{-18}	atto	a	10^{18}	exa	E
10^{-21}	zepto	z	10^{21}	zetta	Z
10^{-24}	yocto	y	10^{24}	yotta	Y

Mathematical operators & functions	
String	Meaning
sym1 sym2	multiplication (a space)
sym1*sym2	multiplication (an asterisk)
sym1.sym2	multiplication (a dot)
sym1/sym2	division
sym1**y	exponentiation (y must be a numerical constant)
sym1^y	exponentiation (y must be a numerical constant)
log(sym1)	common logarithm
ln(sym1)	natural logarithm
exp(sym1)	exponential
sqrt(sym1)	square root

8 Celestial Coordinate Systems (SkyFrames)

A Frame which is specialised for representing coordinate systems on the celestial sphere is obviously of great importance in astronomy. The SkyFrame is such a Frame. In this section we examine the additional properties and behaviour of a SkyFrame that distinguish it from a basic Frame (§7).

8.1 The SkyFrame Model

A SkyFrame is, of course, a Frame (§7) and also a Mapping (§5), so it inherits all the properties and behaviour of these two ancestral classes. When used as a Mapping, a SkyFrame implements a unit transformation, exactly like a basic Frame (§7.3) or a UnitMap, so this aspect of its behaviour is not of great importance.

When used as a Frame, however, a SkyFrame represents a 2-dimensional *spherical* coordinate system, in which the shortest distance between two points is a great circle. A SkyFrame therefore always has exactly two axes which represent the longitude and latitude of a coordinate system residing on the celestial sphere. Many such coordinate systems can be represented by a SkyFrame, as we will see shortly.

A SkyFrame can represent any of the commonly used celestial coordinate systems. Optionally, the origin of the longitude/latitude system can be moved to any specified point in the standard celestial system, allowing a SkyFrame to represent offsets from a specified sky position.

When it is first created, a SkyFrame's axes are always in the order (longitude, latitude) but this can be changed, if required, by using the astPermAxes function (§7.9). The order of the axes can be determined at any time using the LatAxis and LonAxis attributes. A SkyFrame's coordinate values are always stored as angles in (double precision) radians, regardless of the setting of the Unit attribute.

8.2 Creating a SkyFrame

The SkyFrame constructor function is particularly simple and a SkyFrame with default attributes is created as follows:

```
#include "ast.h"
AstSkyFrame *skyframe;

...

skyframe = astSkyFrame( "" );
```

Such a SkyFrame would represent the default celestial coordinate system which, at present, is the ICRS system (the default was ”FK5(J2000)” in versions of AST prior to 3.0).

8.3 Specifying a Particular Celestial Coordinate System

For many purposes, the ICRS coordinate system is perfectly adequate. In order to support conversion between a variety of celestial coordinate systems, however, you can create SkyFrames that represent any of these.

Selection of a particular coordinate system is performed simply by setting a value for the SkyFrame's (character string) System attribute. This setting is most conveniently done when the SkyFrame is created. For example, a SkyFrame representing the old FK4 (B1950.0) coordinate system would be created by:

```
skyframe = astSkyFrame( "System=FK4" );
```

Note that specifying "System=FK4" also changes the associated equinox (from J2000.0 to B1950.0). This is because the default value of the SkyFrame's Equinox attribute (§8.4) depends on the System attribute setting.

You may change the System value at any time, although this is not usually needed. The values supported are set out in the attribute's description in Appendix C and include a variety of equatorial coordinate systems, together with ecliptic and galactic coordinates.

General spherical coordinates are supported by specifying "System=unknown". You should note, though, that no Mapping can be created to convert between "unknown" coordinates and any of the other celestial coordinate systems (see §12).

8.4 Attributes which Qualify Celestial Coordinate Systems

Many celestial coordinate systems have some additional free parameters which serve to identify a particular coordinate system from amongst a broader class of related coordinate systems. For example, the FK5 (J2010.0) system is distinguished from the FK5 (J2000.0) system by a different equinox—and the coordinates of a fixed astronomical source would have different values when expressed in these two systems.

In AST, these free parameters are represented by additional SkyFrame attributes, each of which has a default appropriate to (*i.e.* defined by) the setting of the main System attribute. Each of these *qualifying attributes* may, however, be assigned an explicit value so as to select a particular coordinate system. Note, it is usually best to assign explicit values whenever possible rather than relying on defaults. Attribute should only be left at their default value if you "don't care" what value is used. In certain circumstances (particularly, when aligning two Frames), a default value for an attribute may be replaced by the value from another similar Frame. Such value replacement can be prevented by assigning an explicit value to the attribute, rather than simply relying on the default.

The main SkyFrame attributes which qualify the System attribute are:

Epoch
This attribute is inherited from the Frame class. It gives the moment in time when the coordinates are correct for the astronomical source under study (usually the date of observation).

Equinox
This value is used to qualify celestial coordinate systems that are notionally based on the Earth's equator and/or the ecliptic (the plane of the Earth's orbit around the Sun). The position of either of these planes is difficult to specify precisely, so in practice a model *mean* equator and/or ecliptic are used instead. These, together with the point on the sky that defines the coordinate origin

> (termed the *mean equinox*) move with time according to some model which smoothes out the more rapid fluctuations. The SkyFrame class supports both the old FK4 model and the newer FK5 one.
>
> Coordinates expressed in any of these systems vary with time due to movement (by definition) of the coordinate system itself, and must therefore be qualified by a moment in time (the *epoch of the mean equinox,* or "equinox" for short) which specifies the position of the model coordinate system on the sky. This is the role of the Equinox attribute.
>
> Note that it is quite valid and common to relate the position of a source to an equinox other than the date of observation. Usually a standard equinox such as J2000.0 is used, meaning that the coordinates are referred to axes defined by where the model mean equator and ecliptic would lie on the sky at the Julian epoch J2000.0.

For further details of these attributes you should consult their descriptions in Appendix C and for details of the System settings for which they are relevant, see the description of the System attribute (also in Appendix C). For the interested reader, an excellent overview of celestial coordinate systems can also be found in the documentation for the SLALIB library (SUN/67).

The value of these qualifying attributes is most conveniently set at the same time as the System value, *e.g.* when a SkyFrame is created. For instance:

```
skyframe = astSkyFrame( "System=Ecliptic, Equinox=J2005.5" );
```

would create a SkyFrame representing an ecliptic coordinate system referred to the mean equinox and ecliptic of Julian epoch J2005.5.

Note that it does no harm to assign values to qualifying attributes which are not relevant to the main System value. Any such values are stored, but are not used unless the System value is later set so that they become relevant.

8.5 Using Default SkyFrame Attributes

The default values supplied for many SkyFrame attributes will depend on the value of the SkyFrame's System attribute. In practice, this means that there is usually little need to specify many of these attributes explicitly unless you have some special requirement. This can be illustrated by using astShow to examine a SkyFrame, as follows:

```
astShow( astSkyFrame( "System=FK4-NO-E, Epoch=1958" ) );
```

The output from this might look like the following:

```
 Begin SkyFrame         # Description of celestial coordinate system
#   Title = "FK4 equatorial coordinates; no E-terms; mean equinox B1950.0;
epoch B1958.0"   # Title of coordinate system
    Naxes = 2   # Number of coordinate axes
#   Domain = "SKY"      # Coordinate system domain
```

```
       Epoch = 1958        # Besselian epoch of observation
   #   Lbl1 = "Right ascension"    # Label for axis 1
   #   Lbl2 = "Declination"        # Label for axis 2
       System = "FK4-NO-E"         # Coordinate system type
   #   Uni1 = "hh:mm:ss.s"         # Units for axis 1
   #   Uni2 = "ddd:mm:ss"  # Units for axis 2
   #   Dir1 = 0    # Plot axis 1 in reverse direction
   #   Bot2 = -1.5707963267949     # Lowest legal axis value
   #   Top2 = 1.5707963267949      # Highest legal axis value
       Ax1 =       # Axis number 1
          Begin SkyAxis    # Celestial coordinate axis
          End SkyAxis
       Ax2 =       # Axis number 2
          Begin SkyAxis    # Celestial coordinate axis
          End SkyAxis
    IsA Frame      # Coordinate system description
   #   Eqnox = 1950        # Besselian epoch of mean equinox
    End SkyFrame
```

Note that the defaults (indicated by the "#" comment character at the start of the line) for attributes such as the Title, axis Labels and Format specifiers are all set to values appropriate for the particular equatorial coordinate system that the SkyFrame represents.

This means, for example, that if we were to use this SkyFrame to format a right ascension value stored in radians using astFormat (§7.6), it would automatically result in a string in sexagesimal notation (such as "12:14:35.7") suitable for display. If we changed the value of the SkyFrame's Digits attribute (which is inherited from the Frame class), the number of digits appearing would also change accordingly.

These choices would be appropriate for a System value of "FK4-NO-E", but if a different System value were set, the defaults would be correspondingly different. For example, ecliptic longitude is traditionally expressed in degrees, so setting "System=ecliptic" would result in coordinate values being formatted as degrees by default.

Of course, if you do not like any of these defaults, you may always over-ride them by setting explicit attribute values yourself.

8.6 Formatting Celestial Coordinates

SkyFrames use astFormat for formatting coordinate values in the same way as other Frames (§7.6). However, they offer a different set of formatting options more appropriate to celestial coordinates.

The Digits attribute of a SkyFrame behaves in essentially the same way as for a basic Frame (§7.6), so the precision with which celestial coordinates are displayed can also be adjusted in this way. However, the range of format specifiers that can be given for the Format(axis) attribute, and the default format resulting from any particular Digits value, is different.

The syntax of SkyFrame format specifiers is detailed under the description of the Format(axis) attribute in Appendix C. Briefly, however, it allows celestial coordinates to be expressed either as angles or times and to include one or more of the fields:

- degrees or hours
- arc-minutes or minutes
- arc-seconds or seconds

with a specified number of decimal places for the final field. A range of field separators is also available, as the following examples show:

Format Specifier	**Example Formatted Value**
`d`	`219`
`d.3`	`219.123`
`dm`	`219:05`
`dm.2`	`219:05.44`
`dms`	`219:05:42`
`hms.1`	`15:44:13.8`
`bdms.2`	`219 05 42.81`
`lhms.3`	`15h44m13.88s`
`+zlhms`	`+06h10m44s`
`ms.1`	`13145:42.8`
`lmst.3`	`876m22.854s`
`s.2`	`788742.81`

Note the following key points:

- The required fields are specified using characters chosen from either "dms" or "hms" according to whether the value is to be formatted as an angle (in degrees) or a time (in hours).
- If no degrees or hours field is required, the distinction between angle and time may be made by including "t" to request time.
- The number of decimal places (for the final field) is indicated using "`.`" followed by an integer. An asterisk can be used in place of an integer, in which case the number of decimal places is chosen so that the total number of digits in the formatted value is equal to the value of the Digits attribute.
- "b" causes fields to be separated by blanks, while "l" causes them to be separated by the appropriate letters (the default being a colon).
- "z" causes padding with leading zeros.
- "+" cause a plus sign to be prefixed to positive values (negative values always have a minus sign).

The formatting performed by a SkyFrame is also influenced by the AsTime(axis) attribute, which has a boolean (integer) value for each SkyFrame axis. It determines whether the default format specifier for an axis will present values as angles (*e.g.* in degrees) if it is zero, or as times (*e.g.* in hours) if it is non-zero.

The default AsTime value depends on the celestial coordinate system which the SkyFrame represents which, in turn, depends on its System attribute value. For example, equatorial longitude values (right ascension) are normally expressed in hours, whereas ecliptic longitudes are normally expressed in degrees, so their default AsTime values will reflect this difference.

The value of the AsTime attribute may be set explicitly to over-ride these defaults if required, with the formatting precision being determined by the Digits/Digits(axis) value. Alternatively, the Format(axis) attribute may be set explicitly to specify both the format and precision required. Setting an explicit Format value always over-rides the effects of both the Digits and AsTime attributes (unless the Format value does not specify the required number of decimal places, in which case Digits is used to determine the default number of decimal places)

8.7 Reading Formatted Celestial Coordinates

The process of converting formatted celestial coordinates, such as might be produced by the astFormat function (§8.6), into numerical (double) coordinate values is performed by using astUnformat (§7.8) and passing it a pointer to a SkyFrame. The use of a SkyFrame means that the range of input formats accepted is appropriate to positions on the sky expressed as angles and/or times, while the returned value is in radians.

The following describes the forms of celestial coordinate which are supported:

- You may supply an optional sign, followed by between one and three fields representing either degrees, arc-minutes, arc-seconds or hours, minutes, seconds (*e.g.* “−12 42 03”).
- Each field should consist of a sequence of one or more digits, which may include leading zeros. At most one field may contain a decimal point, in which case it is taken to be the final field (*e.g.* decimal degrees might be given as “124.707”, while degrees and decimal arc-minutes might be given as “−13 33.8”).
- The first field given may take any value, allowing angles and times outside the conventional ranges to be represented. However, subsequent fields must have values of less than 60 (*e.g.* “720 45 31” is valid, whereas “11 45 61” is not).
- Fields may be separated by white space or by “:” (colon), but the choice of separator must be used consistently throughout the value. Additional white space may be present around fields and separators (*e.g.* “− 2: 04 : 7.1”).
- The following field identification characters may be used as separators to replace those above (or may be appended to the final field), in order to identify the field to which they are appended:

  d – degrees
  h – hours
  m – minutes (of arc or time)
  s – seconds (of arc or time)
  `'` – arc-minutes
  `"` – arc-seconds

  Either lower or upper case may be used. Fields must be given in order of decreasing significance (*e.g.* “−11D 3' 14.4"” or “22h14m11.2s”).

- The presence of certain field identification characters indicates whether the value is to be interpreted as an angle or a time (with 24 hours corresponding to 360 degrees), as follows:

  d – angle
  ' – angle
  " – angle
  h – time

  Incompatible angle/time identification characters may not be mixed (*e.g.* “10h14'3"” is not valid). The remaining field identification characters and separators do not specify a preference for an angle or a time and may be used with either.

- If no preference for an angle or a time is expressed anywhere within the value, then it is interpreted as an angle if the Format attribute string associated with the SkyFrame axis generates an angle and as a time otherwise. This ensures that values produced by astFormat (§8.6) are correctly interpreted by astUnformat.

- Fields may be omitted, in which case they default to zero. The remaining fields may be identified by using appropriate field identification characters (see above) and/or by adding extra colon separators (e.g. “−05m13s” is equivalent to “−:05:13”). If a field is not identified explicitly, it is assumed that adjacent fields have been given, after taking account of any extra separator characters. For example:

  10d – degrees
  10d12 – degrees and arc-minutes
  11:14" – arc-minutes and arc-seconds
  9h13s – hours and seconds of time
  :45:33 – minutes and seconds (of arc or time)
  :55: – minutes (of arc or time)
  ::13 – seconds (of arc or time)
  −6::2.5 – degrees/hours and seconds (of arc or time)
  07m14 – minutes and seconds (of arc or time)
  −8:14' – degrees and arc-minutes
  −h3:14 – minutes and seconds of time
  h:2.1 – seconds of time

- If fields are omitted in such a way that the remaining ones cannot be identified uniquely (e.g. “01:02”), then the first field (either given explicitly or implied by an extra leading colon separator) is taken to be the most significant field that astFormat would produce when formatting a value (using the Format attribute associated with the SkyFrame axis). By default, this means that the first field will normally be interpreted as degrees or hours. However, if this does not result in consistent field identification, then the last field (either given explicitly or implied by an extra trailing colon separator) is taken to to be the least significant field that astFormat would produce.

This final convention is intended to ensure that values formatted by astFormat which contain less than three fields will be correctly interpreted if read back using astUnformat, even if they do not contain field identification characters. However, it also affects other forms of input. For example, if the Format(axis) string were set to “mst.1” (producing two fields representing minutes and seconds of time), then formatted input would be interpreted by astUnformat as follows:

12 13 – minutes and seconds
12 – minutes
:13 – seconds
−18: – minutes
12.8 – minutes
1 2 3 – hours, minutes and seconds

4' – arc-minutes
60::" – degrees
−23:" – arc-minutes
−33h – hours

(in the last four cases, explicit field identification has been given which overrides the implicit identification).

Alternatively, if the Format(axis) string were set to "s.3" (producing only an arc-seconds field), then formatted input would be interpreted by astUnformat as follows:

12.8 – arc-seconds
12 13 – arc-minutes and arc-seconds
:12 – arc-seconds
13: – arc-minutes
1 2 3 – degrees, arc-minutes and arc-seconds

In general, if you are preparing formatted input data containing celestial coordinates and wish to omit certain fields, then you are advised to identify clearly those that you do provide by using the appropriate field identification characters and/or extra colon separators. This prevents you depending on the implicit field identification described above which, in turn, depends on an appropriate Format(axis) string having been set.

When writing software, it is also a good idea to set the Format(axis) string so that data input will be as simple as possible for the user. Unless some special effect is desired, this normally means that it should contain "d" or "h" to ensure that the first field entered by the user will be interpreted as degrees or hours, unless otherwise identified. This is the normal behaviour unless an explicit Format(axis) value has been set to override the default.

8.8 Representing Offsets from a Specified Sky Position

A SkyFrame can be modified so that its longitude and latitude axes are referred to an origin at any specified sky position. Such a coordinate system is referred to as an "offset" coordinate syetem. First, the System attribute should be set to represent the celestial coordinate system in which the origin is to be specified. Then the SkyRef attribute should be set to hold the coordinates of the origin within the selected celestial coordinate system.

By default, "north" in the new offset coordinate system is parallel to north in the original celestial coordinate system. However, the direction of north in the offset system can be controlled by assigning a value to the SkyRefP attribute. This attribute should be assigned the celestial coordinates of a point which is on the zero longitude meridian and which has non-zero latitude.

By default, the position given by the SkyRef attribute is used as the origin of the new longitude/latitude system, but an option exists to use it as the north pole of the system instead. This option is controlled by the SkyRefIs attribute. The choice of value for SkyRefIs depends on what sort of offset coordinate system you want. Setting SkyRefIs to "Origin" (the default) produces an offset coordinate system which is approximately Cartesian close to the specified position. Setting SkyRefIs to "Pole" produces an offset coordinate system which is approximately Polar close to the specified position.

9 Spectral Coordinate Systems (SpecFrames)

The SpecFrame is a Frame which is specialised for representing coordinate systems which describe a position within an electro-magnetic spectrum. In this section we examine the additional properties and behaviour of a SpecFrame that distinguish it from a basic Frame (§7).

9.1 The SpecFrame Model

As for a SkyFrame, a SpecFrame is a Frame (§7) and also a Mapping (§5), so it inherits all the properties and behaviour of these two ancestral classes. When used as a Mapping, a SpecFrame implements a unit transformation, exactly like a basic Frame (§7.3) or a UnitMap, so this aspect of its behaviour is not of great importance.

When used as a Frame, however, a SpecFrame represents a wide range of different 1-dimensional coordinate system which can be used to describe positions within a spectrum. The options available largely mirror those described in the FITS-WCS paper III *Representations of spectral coordinates in FITS* (Greisen, Valdes, Calabretta & Allen).

9.2 Creating a SpecFrame

The SpecFrame constructor function is particularly simple and a SpecFrame with default attributes is created as follows:

```
#include "ast.h"
AstSpecFrame *specframe;

...

specframe = astSpecFrame( "" );
```

Such a SpecFrame would represent the default coordinate system which is heliocentric wavelength in metres (i.e. wavelength corrected to take into account the Doppler shift caused by the velocity of the observer around the sun).

9.3 Specifying a Particular Spectral Coordinate System

Selection of a particular coordinate system is performed simply by setting a value for the SpecFrame's (character string) System attribute. This setting is most conveniently done when the SpecFrame is created. For example, a SpecFrame representing Energy would be created by:

```
specframe = astSpecFrame( "System=Energy" );
```

Note that specifying "System=Energy" also changes the associated Unit (from metres to Joules). This is because the default value of the SpecFrame's Unit attribute depends on the System attribute setting.

You may change the System value at any time, although this is not usually needed. The values supported are set out in the attribute's description in Appendix C and include a variety of velocity systems, together with frequency, wavelength, energy, wave-number, *etc.*

9.4 Attributes which Qualify Spectral Coordinate Systems

Many spectral coordinate systems have some additional free parameters which serve to identify a particular coordinate system from amongst a broader class of related coordinate systems. For example, the velocity systems are all parameterised by a rest frequency—the frequency which defines zero velocity, and all coordinate systems are qualified by a 'standard of rest" which indicates the rest frame to which the values refer.

In AST, these free parameters are represented by additional SpecFrame attributes, each of which has a default appropriate to (*i.e.* defined by) the setting of the main System attribute. Each of these *qualifying attributes* may, however, be assigned an explicit value so as to select a particular coordinate system. Note, it is usually best to assign explicit values whenever possible rather than relying on defaults. Attribute should only be left at their default value if you "don't care" what value is used. In certain circumstances (particularly, when aligning two Frames), a default value for an attribute may be replaced by the value from another similar Frame. Such value replacement can be prevented by assigning an explicit value to the attribute, rather than simply relying on the default.

The main SpecFrame attributes which qualify the System attribute are:

Epoch
This attribute is inherited from the Frame class. It gives the moment in time when the coordinates are correct for the astronomical source under study (usually the date of observation). It is needed in order to calculate the Doppler shift produced by the velocity of the observer relative to the centre of the earth, and of the earth relative to the sun.

StdOfRest
This specifies the rest frame in which the coordinates are correct. Transforming between different standards of rest involves taking account of the Doppler shift introduced by the relative motion of the two standards of rest.

RestFreq
Specifies the frequency which correspond to zero velocity. When setting a value for this attribute, the value may be supplied as a wavelength (including an indication of the units being used, "nm" "Angstrom", *etc.*), which will be automatically be converted to a frequency.

RefRA
Specifies the RA (FK5 J2000) of the source. This is used when converting between standards of rest. It specifies the direction along which the component of the relative velocity of the two standards of rest is taken.

RefDec
Specifies the Dec (FK5 J2000) of the source. Used in conjunction with REFRA.

SourceVel
This defines the "source" standard of rest. This is a rest frame which is moving towards the position given by RefRA and RefDec, at a velocity given by SourceVel. The velocity is stored internally as a heliocentric velocity, but can be given in any of the other supported standards of rest.

For further details of these attributes you should consult their descriptions in Appendix C and for details of the System settings for which they are relevant, see the description of the System attribute (also in Appendix C).

Note that it does no harm to assign values to qualifying attributes which are not relevant to the main System value. Any such values are stored, but are not used unless the System value is later set so that they become relevant.

9.5 Using Default SpecFrame Attributes

The default values supplied for many SpecFrame attributes will depend on the value of the SpecFrame's System attribute. In practice, this means that there is usually little need to specify many of these attributes explicitly unless you have some special requirement. This can be illustrated by using astShow to examine a SpecFrame, as follows:

```
astShow( astSpecFrame( "System=Vopt, RestFreq=250 GHz" ) );
```

The output from this might look like the following:

```
 Begin SpecFrame        # Description of spectral coordinate system
#   Title = "Optical velocity, rest frequency = 250 GHz"       # Title
of coordinate system
    Naxes = 1   # Number of coordinate axes
#   Domain = "SPECTRUM"         # Coordinate system domain
#   Epoch = 2000        # Julian epoch of observation
#   Lbl1 = "Optical velocity"  # Label for axis 1
    System = "VOPT"     # Coordinate system type
#   Uni1 = "km/s"       # Units for axis 1
    Ax1 =       # Axis number 1
       Begin Axis       # Coordinate axis
       End Axis
 IsA Frame      # Coordinate system description
#   SoR = "Heliocentric"        # Standard of rest
    RstFrq = 250000000000       # Rest frequency (Hz)
 End SpecFrame
```

Note that the defaults (indicated by the "#" comment character at the start of the line) for attributes such as the Title, axis Labels and Unit specifiers are all set to values appropriate for the particular velocity system that the SpecFrame represents.

These choices would be appropriate for a System value of "Vopt", but if a different System value were set, the defaults would be correspondingly different. For example, by default frequency is measured in units of GHz, not km/s, so setting "System=freq" would change the appropriate line above from:

```
#   Uni1 = "km/s"       # Units for axis 1
```

to

```
#   Uni1 = "GHz"        # Units for axis 1
```

Of course, if you do not like any of these defaults, you may always over-ride them by setting explicit attribute values yourself. For instance, you may choose to have your frequency axis expressed in "kHz" rather than "GHz". To do this simply set the attribute value as follows:

```
astSetC( specframe, "Unit", "kHz" );
```

No error will be reported if you accidentally set an inappropriate Unit value (say "J" - Joules)—after all, AST cannot tell what you are about to do, and you *may* be about to change the System value to "Energy". However, an error *will* be reported if you attempt to find a conversion between two SpecFrames (for instance using astConvert) if either SpecFrame has a Unit value which is inappropriate for its System value.

SpecFrame attributes, like all other attributes, all have default value. However, be aware that for some attributes these default values can never be more than "a legal numerical value" and have no astronomical significance. For instance, the RefRA and RefDec attributes (which give the source position) both have a default value of zero. So unless your source happens to be at that point (highly unlikely!) you will need to set new values. Likewise, the RestFreq (rest frequency) attribute has an arbitrary default value of 1.0E5 GHz. Some operations are not affected by inappropriate values for these attributes (for instance, converting from frequency to wavelength, changing axis units, *etc*), but some are. For instance, converting from frequency to velocity requires a correct rest frequency, moving between different standards of rest requires a correct source position. The moral is, always set explicit values for as many attributes as possible.

9.6 Creating Spectral Cubes

You can use a SpecFrame to describe the spectral axis in a data cube containing two spatial axes and a spectral axis. To do this you would create an appropriate SpecFrame, together with a 2-dimensional Frame (often a SkyFrame) to describe the spatial axes. You would then combine these two Frames together into a single CmpFrame.

```
AstSkyFrame *skyframe;
AstSpecFrame *specframe;
AstCmpFrame *cmpframe;
...
skyframe = astSkyFrame( "Epoch=J2002" );
specframe = astSpecFrame( "System=Freq,StdOfRest=LSRK" );
cmpframe = astCmpFrame( skyframe, specframe, "" );
```

In the resulting CmpFrame, axis 1 will be RA, axis 2 will be Dec and axis 3 will be Frequency. If this is not the order you want, you can permute the axes using astPermAxes.

There is one potential problem with this approach if you are interested in unusually high accuracy. Conversion between different standards of rest involves taking account of the Doppler shift caused by the relative motion of the two standards of rest. At some point this involves finding the component of the relative velocity in the direction of interest. For a SpecFrame, this direction is always given by the RefRA and RefDec attributes, even if the SpecFrame is embedded within a CmpFrame as above. It would be more appropriate if this "direction of interest" was specified by the values passed into the CmpFrame on the RA and DEC axes, allowing each pixel within a data cube to have a slightly different correction for Doppler shift.

Unfortunately, the SpecFrame class cannot do this (since it is purely a 1-dimensional Frame), and so some small degree of error will be introduced when converting between standards of rest, the size of the error varying from pixel to pixel. It is hoped that at some point in the future a sub-class of CmpFrame (a SpecCubeFrame) will be added to AST which allows for this spatial variation in Doppler shift.

The maximum velocity error introduced by this problem is of the order of $V * SIN(FOV)$, where FOV is the angular field of view, and V is the relative velocity of the two standards of rest. As an example, when correcting from the observers rest frame (i.e. the topocentric rest frame) to the kinematic local standard of rest the maximum value of V is about 20 km/s, so for 5 arc-minute field of view the maximum velocity error introduced by the correction will be about 0.03 km/s. As another example, the maximum error when correcting from the observers rest frame to the local group is about 5 km/s over a 1 degree field of view.

9.7 Handling Dual-Sideband Spectra

Dual sideband super-heterodyne receivers produce spectra in which each channel contains contributions from two different frequencies, referred to as the "upper sideband frequency" and the "lower sideband frequency". In the rest frame of the observer (topocentric), these are related to each other as follows:

$$f_{lsb} = 2.f_{LO} - f_{usb} \tag{1}$$

where f_{LO} is a fixed frequency known as the "local oscillator frequency". In other words, the local oscillator frequency is always mid-way between any pair of corresponding upper and lower sideband frequencies[16]. If you want to describe the spectral axis of such a spectrum using a SpecFrame you must choose whether you want the SpecFrame to describe f_{lsb} or f_{usb} - a basic SpecFrame cannot describe both sidebands simultaneously. However, there is a sub-class of SpecFrame, called DSBSpecFrame, which overcomes this difficulty.

A DSBSpecFrame has a SideBand attribute which indicates if the DSBSpecFrame is currently being used to describe the upper or lower sideband spectral axis. The value of this attribute can be changed at any time. If you use the astConvert function to find the Mapping between two DSBSpecFrames, the setting for the two SideBand attributes will be taken into account. Thus, if you take a copy of a DSBSpecFrame, toggle its SideBand attribute, and then use astConvert to find a Mapping from the original to the modified copy, the resulting Mapping will be of the form of equation 1 (if the DSBSpecFrame has its StdOfRest attribute set to "Topocentric").

In general, when finding a Mapping between two arbitrary DSBSpecFrames, the total Mapping is made of of three parts in series:

---

[16] Note, this simple relationship only applies if all frequencies are topocentric.

1. A Mapping which converts the first DSBSpecFrame into its upper sideband representation. If the DSBSpecFrame already represents its upper sideband, this Mapping will be a UnitMap.
2. A Mapping which converts from the first to the second DSBSpecFrame, treating them as if they were both basic SpecFrames. This takes account of any difference in units, standard of rest, system, *etc* between the two DSBSpecFrames.
3. A Mapping which converts the second DSBSpecFrame from its upper sideband representation to its current sideband. If the DSBSpecFrame currently represents its upper sideband, this Mapping will be a UnitMap.

If an attempt is made to find the Mapping between a DSBSpecFrame and a basic SpecFrame, then the DSBSpecFrame will be treated like a basic SpecFrame. In other words, the returned Mapping will not be affected by the setting of the SideBand attribute (or any of the other attributes specific to the DSBSpecFrame class).

In practice, the local oscillator frequency for a dual sideband instrument may not be easily available to an observer. Instead, it is common practice to specify the spectral position of some central feature in the observation (commonly the centre of the instrument passband), together with an “intermediate frequency”. Together, these two values allow the local oscillator frequency to be determined. The intermediate frequency is the difference between the topocentric frequency at the central spectral position and the topocentric frequency of the local oscillator. So:

$$f_{LO} = f_{central} + f_{if} \tag{2}$$

The DSBSpecFrame class uses the DSBCentre attribute to specify the central spectral position ($f_{central}$), and the IF attribute to specify the intermediate frequency (f_{if}). The DSBCentre value is given and returned in the spectral system described by the DSBSpecFrame (thus you do not need to calculate the corresponding topocentric frequency yourself - this will be done automatically by the DSBSpecFrame when you assign a new value to the DSBCentre attribute). The value assigned to the IF attribute should always be a topocentric frequency in units of Hz, however a negative value may be given to indicate that the DSBCentre value is in the upper sideband (that is, if $IF < 0$ then $f_{central} > f_{LO}$). A positive value for IF indicates that the DSBCentre value is in the lower sideband (that is, if $IF > 0$ then $f_{central} < f_{LO}$).

10 Time Systems (TimeFrames)

The TimeFrame is a Frame which is specialised for representing moments in time. In this section we examine the additional properties and behaviour of a TimeFrame that distinguish it from a basic Frame (§7).

10.1 The TimeFrame Model

As for a SkyFrame, a TimeFrame is a Frame (§7) and also a Mapping (§5), so it inherits all the properties and behaviour of these two ancestral classes. When used as a Mapping, a TimeFrame implements a unit transformation, exactly like a basic Frame (§7.3) or a UnitMap, so this aspect of its behaviour is not of great importance.

When used as a Frame, however, a TimeFrame represents a wide range of different 1-dimensional coordinate system which can be used to describe moments in time. Absolute times and relative (i.e. elapsed) times are supported (attribute TimeOrigin), as are a range of different time scales (attribute TimeScale). An absolute or relative value in any time scale can be represented in different forms such as Modified Julian Date, Julian Epoch, *etc* (attribute System). AST extends the definition of these systems to allow them to be used with any unit of time (attribute Unit). The TimeFrame class also allows times to formatted as either a simple floating point value or as a Gregorian date and time of day (attribute Format).

10.2 Creating a TimeFrame

The TimeFrame constructor function is particularly simple and a TimeFrame with default attributes is created as follows:

```
#include "ast.h"
AstTimeFrame *timeframe;

...

timeframe = astTimeFrame( "" );
```

Such a TimeFrame would represent the default coordinate system which is Modified Julian Date (with the usual units of days) in the International Atomic Time (TAI) time scale.

10.3 Specifying a Particular Time System

By setting the System attribute appropriately, the TimeFrame can represent Julian Date, Modified Julian Date, Julian Epoch or Besselian Epoch (the time scale is specified by a separate attribute called TimeScale).

Selection of a particular coordinate system is performed simply by setting a value for the TimeFrame's (character string) System attribute. This setting is most conveniently done when the TimeFrame is created. For example, a TimeFrame representing Julian Epoch would be created by:

```
timeframe = astTimeFrame( "System=JEPOCH" );
```

Note that specifying "System=JEPOCH" also changes the associated default Unit (from days to years). This is because the default value of the TimeFrame's Unit attribute depends on the System attribute setting.

You may change the System value at any time, although this is not usually needed. The values supported are set out in the attribute's description in Appendix C.

10.4 Attributes which Qualify Time Coordinate Systems

Time coordinate systems require some additional free parameters to identify a particular coordinate system from amongst a broader class of related coordinate systems. For example, all TimeFrames are qualified by the time scale (that is, the physical process used to define the flow of time), and some require the position of the observer's clock.

In AST, these free parameters are represented by additional TimeFrame attributes, each of which has a default appropriate to (*i.e.* defined by) the setting of the main System attribute. Each of these *qualifying attributes* may, however, be assigned an explicit value so as to select a particular coordinate system. Note, it is usually best to assign explicit values whenever possible rather than relying on defaults. Attribute should only be left at their default value if you "don't care" what value is used. In certain circumstances (particularly, when aligning two Frames), a default value for an attribute may be replaced by the value from another similar Frame. Such value replacement can be prevented by assigning an explicit value to the attribute, rather than simply relying on the default.

The main TimeFrame attributes which qualify the System attribute are:

TimeScale
This specifies the time scale.

LTOffset
This specifies the offset from Local Time to UTC in hours (time zones east of Greenwich have positive values). Note, AST uses the value as supplied without making any correction for daylight saving.

TimeOrigin
This specifies the zero point from which time values are measured, within the system specified by the System attribute. Thus, a value ofzero (the default) indicates that time values represent absolute times. Non-zero values may be used to indicate that the TimeFrame represents elapsed time since the specified origin.

For further details of these attributes you should consult their descriptions in Appendix C and for details of the System settings for which they are relevant, see the description of the System attribute (also in Appendix C).

Note that it does no harm to assign values to qualifying attributes which are not relevant to the main System or TimeScale value. Any such values are stored, but are not used unless the System and/or TimeScale value is later set so that they become relevant.

11 Compound Frames (CmpFrames)

We now turn to a rather special form of Mapping, the CmpFrame. The Frames we have considered so far have been atomic, in the sense that they represent pre-defined elementary physical domains. A CmpFrame, however, is a compound Frame. In essence, it is a structure for containing other Frames and its purpose is to allow those Frames to work together in various combinations while appearing as a single Object. A CmpFrame's behaviour is therefore not pre-defined, but is determined by the other Frames it contains (its "component" Frames).

As with compound Mappings, compound Frames can be nested within each other, forming arbitrarily complex Frames.

11.1 Creating a CmpFrame

A very common use for a CmpFrame within astronomy is to represent a "spectral cube". This is a 3-dimensional Frame in which one of the axes represents position within a spectrum, and the other two axes represent position on the sky (or some other spatial domain such as the focal plane of a telescope). As an example, we create such a CmpFrame in which axes 1 and 2 represent Right Ascension and Declination (ICRS), and axis 3 represents wavelength (these are the default coordinate Systems represented by a SkyFrame and a SpecFrame respectively):

```
AstSkyFrame *skyframe;
AstSpecFrame *specframe;
AstCmpFrame *cmpframe;
...
skyframe = astSkyFrame( "" );
specframe = astSpecFrame( "" );
cmpframe = astCmpFrame( skyframe, specframe, "" );
```

If it was desired to make RA and Dec correspond to axes 1 and 3, with axis 2 being the spectral axis, then the axes of the CmpFrame created above would need to be permuted as follows:

```
int perm[ 3 ];
...

perm[ 0 ] = 0;
perm[ 1 ] = 2;
perm[ 2 ] = 1;
astPermAxes( cmpframe, perm );
```

11.2 The Attributes of a CmpFrame

A CmpFrame *is a* Frame and so has all the attributes of a Frame. The default value for the Domain attribute for a CmpFrame is formed by concatenating the Domains of the two component Frames, separated by a minus sign ("-").[17] The (fixed) value for its System attribute

---

[17] If both component Frames have blank Domains, then the default Domain for the CmpFrame is the string "CMP".

is "Compound".[18] A CmpFrame has no further attributes over and above those common to all Frames. However, attributes of the two component Frames can be accessed as if they were attributes of the CmpFrame, as described below.

Frame attributes which are specific to individual axes (such as Label(2), Format(1), *etc*) simply mirror the corresponding axes of the relevant component Frame. That is, if the "Label(2)" attribute of a CmpFrame is accessed, the CmpFrame will forward the access request to the component Frame which contains axis 2. Thus, default values for axis attributes will be the same as those provided by the component Frames.

An axis index can optionally be appended to the name of Frames attributes which do not normally have such an index (System, Domain, Epoch, Title, *etc*). If this is done, the access request is forwarded to the component Frame containing the indicated axis. For instance, if a CmpFrame contains a SpecFrame and a SkyFrame in that order, and the axes have not been permuted, then getting the value of attribute "System" will return "Compound" as mentioned above (that is, the System value of the CmpFrame as a whole), whereas getting the value of attribute "System(1)" will return "Spectral"(that is, the System value of the component Frame containing axis 1 — the SpecFrame).

This technique is not limited to attributes common to all Frames. For instance, the SkyFrame class defines an attribute called Equinox which is not held by other classes of Frames. To set a value for the Equinox attribute of the SkyFrame contained within the above CmpFrame, assign the value to the "Equinox(2)" attribute of the CmpFrame. Since the SkyFrame defines both axes 2 and 3 of the CmpFrame, we could equivalently have set a value for "Equinox(3)" since this would also result in the attribute access being forwarded to the SkyFrame.

Finally, if an attribute is not qualified by a axis index, attempts will be made to access it using each of the CmpFrame axes in turn. Using the above example of the spectral cube, if an attempt was made to get the value of attribute "Equinox" (with no axis index), each axis in turn would be used. Since axis 1 is contained within a SpecFrame, the first attempt would fail since the SpecFrame class does not have an Equinox attribute. However, the second attempt would succeed because axis 2 is contained within a SkyFrame which *does* have an Equinox attribute. Thus the returned attribute value would be that obtained from the SkyFrame containing axis 2. When getting or testing an attribute value, the returned value is determined by the *first* axis which recognises the attribute. When setting an attribute value, *all* axes which recognises the attribute have the attribute value set to the given value. Likewise, when clearing an attribute value, all axes which recognises the attribute have the attribute value cleared.

---

[18] Any attempt to change the System value of a CmpFrame is ignored.

12 An Introduction to Coordinate System Conversions

In this section, we start to look at techniques for converting between different coordinate systems. At this stage, the tools we have available are Frames (§7), SkyFrames (§8), SpecFrames (§9), TimeFrames (§10) and various Mappings (§5). These are sufficient to allow us to begin examining the problem, but more sophisticated approaches will also emerge later (§14.2).

12.1 Converting between Celestial Coordinate Systems

We begin by examining how to convert between two celestial coordinate systems represented by SkyFrames, as this is both an illuminating and practical example. Consider the problem of converting celestial coordinates between:

1. The old FK4 system, with no E terms, a Besselian epoch of 1958.0 and a Besselian equinox of 1960.0.
2. An ecliptic coordinate system based on the mean equinox and ecliptic of Julian epoch 2010.5.

This example is arbitrary but not completely unrealistic. Unless you already have expertise with such conversions, you are unlikely to find it straightforward.

Using AST, we begin by creating two SkyFrames to represent these coordinate systems, as follows:

```
#include "ast.h"
AstSkyFrame *skyframe1, *skyframe2;

...

skyframe1 = astSkyFrame( "System=FK4-NO-E, Epoch=B1958, Equinox=B1960" );
skyframe2 = astSkyFrame( "System=Ecliptic, Equinox=J2010.5" );
```

Note how specifying the coordinate systems consists simply of initialising the attributes of each SkyFrame appropriately. The next step is to find a way of converting between these SkyFrames. This is done using astConvert, as follows:

```
AstFrameSet *cvt;

...

cvt = astConvert( skyframe1, skyframe2, "" );
if ( cvt == AST__NULL ) {
   <conversion is not possible>
} else {
   <conversion is possible>
}
```

The third argument of astConvert is not used here and should be an empty string.

astConvert will return a null result, AST__NULL (as defined in the "ast.h" header file), if conversion is not possible. In this example, conversion is possible, so it will return a pointer to a new Object that describes the conversion.

The Object returned is called a FrameSet. We have not discussed FrameSets yet (§13), but for the present purposes we can consider them simply as Objects that can behave both as Mappings and as Frames. It is the FrameSet's behaviour as a Mapping in which we are mainly interested here, because the Mapping it implements is the one we require—*i.e.* it converts between the two celestial coordinate systems (§14.1).

For example, if "alpha1" and "delta1" are two arrays containing the longitude and latitude, in radians, of N points on the sky in the original coordinate system (corresponding to "skyframe1"), then they could be converted into the new coordinate system (represented by "skyframe2") as follows:

```
#define N 10
double alpha1[ N ], delta1[ N ];
double alpha2[ N ], delta2[ N ];

...

astTran2( cvt, N, alpha1, delta1, 1, alpha2, delta2 );
```

The new coordinates are returned *via* the "alpha2" and "delta2" arrays. To transform coordinates in the opposite direction, we simply invert the 5th (boolean int) argument to astTran2, as follows:

```
astTran2( cvt, N, alpha2, delta2, 0, alpha1, delta1 );
```

The FrameSet returned by astConvert also contains information about the SkyFrames used in the conversion (§14.1). As we mentioned above, a FrameSet may be used as a Frame and in this case it behaves like the "destination" Frame used in the conversion (*i.e.* like "skyframe2"). We could therefore use the "cvt" FrameSet to calculate the distance between two points (with coordinates in radians) in the destination coordinate system, using astDistance:

```
double distance, point1[ 2 ], point2[ 2 ];

...

distance = astDistance( cvt, point1, point2 );
```

and the result would be the same as if the "skyframe2" SkyFrame had been used.

Another way to see how the FrameSet produced by astConvert retains information about the coordinate systems involved is to set its Report attribute (inherited from the Mapping class) so that it displays the coordinates before and after conversion (§4.8):

```
astSet( cvt, "Report=1" );
astTran2( cvt, N, alpha1, delta1, 1, alpha2, delta2 );
```

The output from this might look like the following:

```
(2:06:03.0, 34:22:39) --> (42.1087, 20.2717)
(2:08:20.6, 35:31:24) --> (43.0197, 21.1705)
(2:10:38.1, 36:40:09) --> (43.9295, 22.0716)
(2:12:55.6, 37:48:55) --> (44.8382, 22.9753)
(2:15:13.1, 38:57:40) --> (45.7459, 23.8814)
(2:17:30.6, 40:06:25) --> (46.6528, 24.7901)
(2:19:48.1, 41:15:11) --> (47.5589, 25.7013)
(2:22:05.6, 42:23:56) --> (48.4644, 26.6149)
(2:24:23.1, 43:32:41) --> (49.3695, 27.5311)
(2:26:40.6, 44:41:27) --> (50.2742, 28.4499)
```

Here, we see that the input FK4 equatorial coordinate values (given in radians) have been formatted automatically in sexagesimal notation using the conventional hours for right ascension and degrees for declination. Conversely, the output ecliptic coordinates are shown in decimal degrees, as is conventional for ecliptic coordinates. Both are displayed using the default precision of 7 digits.[19]

In fact, the "cvt" FrameSet has access to all the information in the original SkyFrames which were passed to astConvert. If you had set a new Digits attribute value for either of these, the formatting above would reflect the different precision you requested by displaying a greater or smaller number of digits.

12.2 Converting between Spectral Coordinate Systems

The principles described in the previous section for converting between celestial coordinate systems also apply to the task of converting between spectral coordinate systems. As an example, let's look at how we might convert between frequency measured in GHz as measured in the rest frame of the telescope, and radio velocity measured in km/s measured with respect the kinematic Local Standard of Rest.

First we create a default SpecFrame, and then set its attributes to describe the required radio velocity system (this is slightly more convenient, given the relatively large number of attributes, than specifying the attribute values in a single string such as would be passed to the SpecFrame constructor). We then take a copy of this SpecFrame, and change the attribute values so that the copy describes the original frequency system (modifying a copy, rather than creating a new SpecFrame from scratch, avoids the need to specify the epoch, reference position, *etc* a second time since they are all inherited by the copy):

```
#include "ast.h"
AstSpecFrame *specframe1, *specframe2;

...

specframe1 = astSpecFrame( "" );
astSet( specframe1, "System=vradio" );
```

---

[19] The leading digit is zero and is therefore not seen in this particular example.

```
astSet( specframe1, "Unit=km/s" );
astSet( specframe1, "Epoch=1996-Oct-2 12:13:56.985" );
astSet( specframe1, "ObsLon=W155:28:18" );
astSet( specframe1, "ObsLat=N19:49:34" );
astSet( specframe1, "RefRA=18:14:50.6" );
astSet( specframe1, "RefDec=-4:40:49" );
astSet( specframe1, "RestFreq=230.538 GHz" );
astSet( specframe1, "StdOfRest=LSRK" );

specframe2 = astCopy( specframe1 );
astSet( specframe1, "System=freq" );
astSet( specframe1, "Unit=GHz" );
astSet( specframe1, "StdOfRest=Topocentric" );
```

Note, the fact that a SpecFrame has only a single axis means that we were able to refer to the Unit attribute without an axis index. The other attributes are: the time of of observation (Epoch), the geographical position of the telescope (ObsLat & ObsLon), the position of the source on the sky (RefRA & RefDec), the rest frequency (RestFreq) and the standard of rest (StdOfRest).

The next step is to find a way of converting between these SpecFrames. We use exactly the same code that we did in the previous section where we were converting between celestial coordinate systems:

```
AstFrameSet *cvt;

...

cvt = astConvert( specframe1, specframe2, "" );
if ( cvt == AST__NULL ) {
   <conversion is not possible>
} else {
   <conversion is possible>
}
```

A before, this will give us a FrameSet (assuming conversion is possible, which should always be the case for our example), and we can use the FrameSet to convert between the two spectral coordinate systems. We use astTran1 in place of astTran2 since a SpecFrame has only one axis (unlike a SkyFrame which has two).

For example, if “frq” is an array containing the observed frequency, in GHz, of N spectral channels (describe by “specframe1”), then they could be converted into the new coordinate system (represented by “specframe2”) as follows:

```
#define N 10
double frq[ N ];
double vel[ N ];

...

astTran1( cvt, N, frq, 1, vel );
```

The radio velocity values are returned in the “vel” array.

12.3 Converting between Time Coordinate Systems

All the principles outlined in the previous section about aligning spectral cocordinate systems (SpecFrames) can be applied directly to the problem of aligning time coordinate systems (TimeFrames).

12.4 Handling SkyFrame Axis Permutations

We can illustrate an important point if we swap the axis order of either SkyFrame in the example above (§12.1) before identifying the conversion. Let's assume we use astPermAxes (§7.9) to do this to the second SkyFrame, before applying astConvert, as follows:

```
int perm[ 2 ] = { 2, 1 };

...

astPermAxes( skyframe2, perm );
cvt = astConvert( skyframe1, skyframe2, "" );
```

Now, the destination SkyFrame system no longer represents the coordinate system:

(ecliptic longitude, ecliptic latitude)

but instead represents the transposed system:

(ecliptic latitude, ecliptic longitude)

As a consequence, when we use the FrameSet returned by astConvert to apply a coordinate transformation, we obtain something like the following:

```
(2:06:03.0, 34:22:39) --> (20.2717, 42.1087)
(2:08:20.6, 35:31:24) --> (21.1705, 43.0197)
(2:10:38.1, 36:40:09) --> (22.0716, 43.9295)
(2:12:55.6, 37:48:55) --> (22.9753, 44.8382)
(2:15:13.1, 38:57:40) --> (23.8814, 45.7459)
(2:17:30.6, 40:06:25) --> (24.7901, 46.6528)
(2:19:48.1, 41:15:11) --> (25.7013, 47.5589)
(2:22:05.6, 42:23:56) --> (26.6149, 48.4644)
(2:24:23.1, 43:32:41) --> (27.5311, 49.3695)
(2:26:40.6, 44:41:27) --> (28.4499, 50.2742)
```

When compared to the original (§12.1), the output coordinate order has been swapped to compensate for the different destination SkyFrame axis order.

In all, there are four possible axis combinations, corresponding to two possible axis orders for each of the source and destination SkyFrames, and astConvert will convert correctly between any of these. The point to note is that a SkyFrame contains knowledge about how to convert to and from other SkyFrames. Since its two axes (longitude and latitude) are distinguishable, the conversion is able to take account of the axis order.

If you need to identify the axes of a SkyFrame explicitly, taking into account any axis permutations, the LatAxis and LonAxis attributes can be used. These are read-only attributes which give the indices of the latitude and longitude axes respectively.

12.5 Converting Between Frames

Having seen how clever SkyFrames are (§12.1 and §12.4), we will next examine how dumb a basic Frame can be in comparison. For example, if we create two 2-dimensional Frames and use astConvert to derive a conversion between them, as follows:

```
AstFrame *frame1, *frame2;

...

frame1 = astFrame( 2, "" );
frame2 = astFrame( 2, "" );
cvt = astConvert( frame1, frame2, "" );
```

then the coordinate transformation which the "cvt" FrameSet performs will be as follows:

```
(1, 2) --> (1, 2)
(2, 4) --> (2, 4)
(3, 6) --> (3, 6)
(4, 8) --> (4, 8)
(5, 10) --> (5, 10)
```

This is an identity transformation, exactly the same as a UnitMap (§5.9). Even if we permute the axis order of our Frames, as we did above (§12.4), we will fare no better. The conversion between our two basic Frames will always be an identity transformation.

The reason for this is that, unlike a SkyFrame, all basic Frames start life the same and have axes that are indistinguishable. Therefore, permuting their axes doesn't make them look any different—they still represent the same coordinate system.

12.6 The Choice of Alignment System

In practice, when AST is asked to find a conversion between two Frames describing two different coordinate systems on a given physical domain, it uses an intermediate "alignment" system. Thus, when finding a conversion from system A to system B, AST first finds the Mapping from system A to some alignment system, system C, and then finds the Mapping from this system C to the required system B. It finally concatenates these two Mappings to get the Mapping from system A to system B.

One advantage of this is that it cuts down the number of conversion algorithms required. If there are N different Systems which may be used to describe positions within the Domain, then this approach requires about $2 * N$ conversion algorithms to be written. The alternative approach of going directly from system A to system B would require about $N * N$ conversion algorithms.

In addition, the use of an intermediate alignment system highlights the nature of the conversion process. What do we mean by saying that a Mapping "converts a position in one coordinate system into the corresponding position in another"? In practice, it means that the input and output coordinates correspond to the same coordinates *in some third coordinate system.* The choice of this third coordinate system, the "alignment" system, can completely alter the nature of the Mapping. The Frame class has an attribute called AlignSystem which can be used to specify the alignment system.

As an example, consider the case of aligning two spectra calibrated in radio velocity, but each with a different rest frequency (each spectrum will be described by a SpecFrame). Since the rest frequencies differ, a given velocity will correspond to different frequencies in the two spectra. So when we come to "align" these two spectra (that is, find a Mapping which converts positions in one SpecFrame to the corresponding positions in the other), we have the choice of aligning the frequencies or aligning the velocities. Different Mappings will be required to describe these two forms of alignment. If we set AlignSystem to "Freq" then the returned Mapping will align the frequencies described by the two SpecFrames. On the other hand, if we set AlignSystem to "Vradio" then the returned Mapping will align the velocities.

Some choices of alignment system are redundant. For instance, in the above example, changing the alignment system from frequency to wavelength has no effect on the returned Mapping: if two spectra are aligned in frequency they will also be aligned in wavelength (assuming the speed of light doesn't change).

The default value for AlignSystem depends on the class of Frame. For a SpecFrame, the default is wavelength (or equivalently, frequency) since this is the system in which observations are usually made. The SpecFrame class also has an attribute called AlignStdOfRest which allows the standard of rest of the alignment system to be specified. Similarly, the TimeFrame class has an attribute called AlignTimeScale which allows the time scale of the alignment system to be specified. Currently, the SkyFrame uses ICRS as the default for AlignSystem, since this is a close approximation to an inertial frame of rest.

13 Coordinate System Networks (FrameSets)

We saw in §12 how astConvert could be used to find a Mapping that inter-relates a pair of coordinate systems represented by Frames. There is a limitation to this, however, in that it can only be applied to coordinate systems that are inter-related by suitable conventions. In the case of celestial coordinates, the relevant conventions are standards set out by the International Astronomical Union, and others, that define what these coordinate systems mean. In practice, however, the relationships between many other coordinate systems are also of practical importance.

Consider, for example, the focal plane of a telescope upon which an image of the sky is falling. We could measure positions in this focal plane in millimetres or, if there were a detector system such as a CCD present, we could count pixels. We could also use celestial coordinates of many different kinds. All of these systems are equivalent in their effectiveness at specifying positions in the focal plane, but some are more convenient than others for particular purposes.

Although we could, in principle, convert between all of these focal plane coordinate systems, there is no pre-defined convention for doing so. This is because the conversions required depend on where the telescope is pointing and how the CCD is mounted in the focal plane. Clearly, knowledge about this cannot be built into the AST library and must be supplied in some other way. Note that this is exactly the same problem as we met in §7.12 when discussing the Domain attribute—*i.e.* coordinate systems that apply to different physical domains require that extra information be supplied before we can convert between them.

What we need, therefore, is a general way to describe how coordinate systems are inter-related, so that when there is no convention already in place, we can define our own. We can then look forward to converting, say, from pixels into galactic coordinates and *vice versa.* In AST, the FrameSet class provides this capability.

13.1 The FrameSet Model

Consider a coordinate system (call it number 1) which is represented by a Frame of some kind. Now consider a Mapping which, when applied to the coordinates in system 1 yields coordinates in another system, number 2. The Mapping therefore inter-relates coordinate systems 1 and 2.

Now consider a second Mapping which inter-relates system 1 and a further coordinate system, number 3. If we wanted to convert coordinates between systems 2 and 3, we could do so by:

1. Applying our first Mapping in reverse, so as to convert between systems 2 and 1.
2. Applying the second Mapping, as given, to convert between systems 1 and 3.

We are not limited to three coordinate systems, of course. In fact, we could continue to introduce any number of further coordinate systems, so long as we have a suitable Mapping for each one which relates it to one of the Frames already present. Continuing in this way, we can build up a network in which Frames are inter-related by Mappings in such a way that there is always a way of converting between any pair of coordinate systems.

The FrameSet (Figure 7) encapsulates these ideas. It is a network composed of Frames and associated Mappings, in which there is always exactly one path, *via* Mappings, between any pair of Frames. Since we assemble FrameSets ourselves, they can be used to represent any coordinate systems we choose and to set up the particular relationships between them that we want.

13.2 Creating a FrameSet

Before we can create a FrameSet, we must have a Frame of some kind to put into it, so let's create a simple one:

```
#include "ast.h"
AstFrame *frame1;

...

frame1 = astFrame( 2, "Domain=A" );
```

We have set this Frame's Domain attribute (§7.12) to A so that it will be distinct from the others we will be using. We can now create a new FrameSet containing just this Frame, as follows:

```
AstFrameSet *frameset;

...

frameset = astFrameSet( frame1, "" );
```

So far, however, this Frame isn't related to any others.

13.3 Adding New Frames to a FrameSet

We can now add further Frames to the FrameSet created above (§13.2). To do so, we must supply a new Frame and an associated Mapping that relates it to any of the Frames that are already present (there is only one present so far). To keep the example simple, we will just use a ZoomMap that multiplies coordinates by 10. The required Objects are created as follows:

```
AstFrame *frame2;
AstMapping *mapping12;

...

frame2 = astFrame( 2, "Domain=B" );
mapping12 = astZoomMap( 2, 10.0, "" );
```

To add the new Frame into our FrameSet, we use the astAddFrame function:

```
astAddFrame( frameset, 1, mapping12, frame2 );
```

Whenever a Frame is added to a FrameSet, it is assigned an integer index. This index starts with 1 for the initial Frame used to create the FrameSet (§13.2) and increments by one every time a new Frame is added. This index is the primary way of identifying the Frames within a FrameSet.

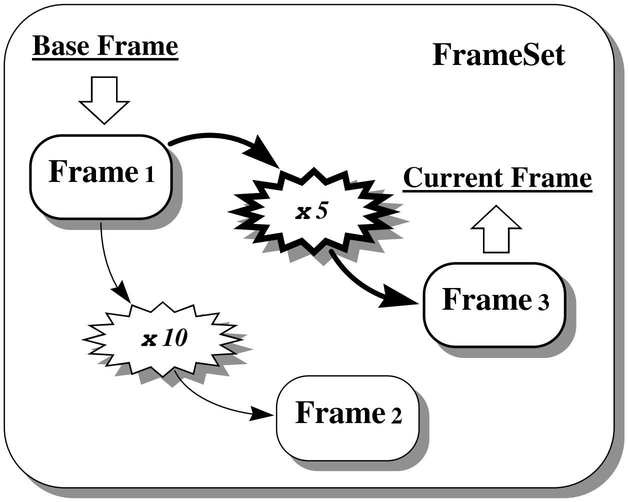


Figure 11: An example FrameSet, in which Frames 2 and 3 are related to Frame 1 by multiplying its coordinates by factors of 10 and 5 respectively. The FrameSet's Base attribute has the value 1 and its Current attribute has the value 3. The transformation performed when the FrameSet is used as a Mapping (*i.e.* from its base to its current Frame) is shown in bold.

When a Frame is added, we also have to specify which of the existing ones the new Frame is related to. Here, we chose number 1, the only one present so far, and the new one we added became number 2.

Note that a FrameSet does not make copies of the Frames and Mappings that you insert into it. Instead, it holds pointers to them. This means that if you retain the original pointers to these Objects and alter them, you will indirectly be altering the FrameSet's contents. You can, of course, always use astCopy (§4.13) to make a separate copy of any Object if you need to ensure its independence.

We could also add a third Frame into our FrameSet, this time defining a coordinate system which is reached by multiplying the original coordinates (of "frame1") by 5:

```
astAddFrame( frameset, 1, astZoomMap( 2, 5.0, "" ), astFrame( 2, "Domain=C" ) );
```

Here, we have avoided storing unnecessary pointer values by using function invocations directly as arguments for astAddFrame. This assumes that we are using astBegin and astEnd (§4.10) to ensure that Objects are correctly deleted when no longer required.

Our example FrameSet now contains three Frames and two Mappings with the arrangement shown in Figure 11. The total number of Frames is given by its read-only Nframe attribute.

13.4 The Base and Current Frames

At all times, one of the Frames in a FrameSet is designated to be its *base* Frame and one to be its *current* Frame (Figure 11). These Frames are identified by two integer FrameSet attributes, Base and Current, which hold the indices of the nominated Frames within the FrameSet.

The existence of the base and current Frames reflects an important application of FrameSets, which is to attach coordinate systems to entities such as data arrays, data files, plotting surfaces (for graphics), *etc.* In this context, the base Frame represents the "native" coordinate system of the attached entity—for example, the pixel coordinates of an image or the intrinsic coordinates of a plotting surface. The other Frames within the FrameSet represent alternative coordinate systems which may also be used to refer to positions within that entity. The current Frame represents the particular coordinate system which is currently selected for use. For instance, if an image were being displayed, you would aim to label it with coordinates corresponding to the current Frame. In order to see a different coordinate system, a software user would arrange for a different Frame to be made current.

The choice of base and current Frames may be changed at any time, simply by assigning new values to the FrameSet's Base and Current attributes. For example, to make the Frame with index 3 become the current Frame, you could use:

```
astSetI( frameset, "Current", 3 );
```

You can nominate the same Frame to be both the base and current Frame if you wish.

By default (*i.e.* if the Base or Current attribute is un-set), the first Frame added to a FrameSet becomes its base Frame and the last one added becomes its current Frame.[20] Whenever a new Frame is added to a FrameSet, the Current attribute is modified so that the new Frame becomes the current one. This behaviour is reflected in the state of the example FrameSet in Figure 11.

13.5 Referring to the Base and Current Frames

It is often necessary to refer to the base and current Frames (§13.4) within a FrameSet, but it can be cumbersome having to obtain their indices from the Base and Current attributes on each occasion. To make this easier, two macros, AST__BASE and AST__CURRENT, are defined in the "ast.h" header file and may be used to represent the indices of the base and current Frames respectively. They may be used whenever a Frame index is required.

For example, when adding a new Frame to a FrameSet (§13.3), you could use the following to indicate that the new Frame is related to the existing current Frame, whatever its index happens to be:

```
AstFrame *frame;
AstMapping *mapping;

...

astAddFrame( frameset, AST__CURRENT, mapping, frame );
```

Of course, the Frame you added would then become the new current Frame.

---

[20] Although this is reversed if the FrameSet's Invert attribute is non-zero.

13.6 Using a FrameSet as a Mapping

The FrameSet class inherits properties and behaviour from the Frame class (§7) and, in turn, from the Mapping class (§5). Its behaviour when used as a Mapping is particularly important.

Consider, for instance, passing a FrameSet pointer to a coordinate transformation function such as astTran2:

```
#define N 10
double xin[ N ], yin[ N ], xout[ N ], yout[ N ];

...

astTran2( frameset, N, xin, yin, 1, xout, yout );
```

The coordinate transformation applied by this FrameSet would be the one which converts between its base and current Frames. Using the FrameSet in Figure 11, for example, the coordinates would be multiplied by a factor of 5. If we instead requested the FrameSet's inverse transformation, we would be transforming from its current Frame to its base Frame, so our example FrameSet would then multiply by a factor of 0.2.

Whenever the choice of base and current Frames changes, the transformations which a FrameSet performs when used as a Mapping also change to reflect this. The Nin and Nout attributes may also change in consequence, because they are determined by the numbers of axes in the FrameSet's base and current Frames respectively. These numbers need not necessarily be equal, of course.

Like any Mapping, a FrameSet may also be inverted by changing the boolean sense of its Invert attribute, *e.g.* using astInvert (§5.5). If this is happens, the values of the FrameSet's Base and Current attributes are interchanged, along with its Nin and Nout attributes, so that its base and current Frames swap places. When used as a Mapping, the FrameSet will therefore perform the inverse transformation to that which it performed previously.

To summarise, a FrameSet may be used exactly like any other Mapping which inter-relates the coordinate systems described by its base and current Frames.

13.7 Extracting a Mapping from a FrameSet

Although it is very convenient to use a FrameSet when a Mapping is required (§13.6), a FrameSet necessarily contains additional information and sometimes this might cause inefficiency or confusion. For example, if you wanted to use a Mapping contained in one FrameSet and insert it into another, it would probably not be efficient to insert the whole of the first FrameSet into the second one, although it would work.

In such a situation, the astGetMapping function allows you to extract a Mapping from a FrameSet. You do this by specifying the two Frames which the Mapping should inter-relate using their indices within the FrameSet. For example:

```
map = astGetMapping( frameset, 2, 3 );
```

would return a pointer to a Mapping that converted between Frames 2 and 3 in the FrameSet. Its inverse transformation would then convert in the opposite direction, *i.e.* between Frames 3 and 2. Note that this Mapping might not be independent of the Mappings contained within the FrameSet—*i.e.* they may share sub-Objects—so astCopy should be used to make a copy if you need to guarantee independence (§4.13).

Very often, the Mapping returned by astGetMapping will be a compound Mapping, or CmpMap (§6). This reflects the fact that conversion between the two Frames may need to be done *via* an intermediate coordinate system so that several stages may be involved. You can, however, easily simplify this Mapping (where this is possible) by using the astSimplify function (§6.7) and this is recommended if you plan to use it for transforming a large amount of data.

13.8 Using a FrameSet as a Frame

A FrameSet can also be used as a Frame, in which capacity it almost always behaves as if its current Frame had been used instead. For example, if you request the Title attribute of a FrameSet using:

```
const char *title;

...

title = astGetC( frameset, "Title" );
```

the result will be the Title of the current Frame, or a suitable default if the current Frame's Title attribute is un-set. The same also applies to other attribute operations—*i.e.* setting, clearing and testing attributes. Most attributes shared by both Frames and FrameSets behave in this way, such as Naxes, Label(axis), Format(axis), *etc.* There are, however, a few exceptions:

Class
Has the value "FrameSet".

ID
Identifies the particular FrameSet (not its current Frame).

Nin
Equals the number of axes in the FrameSet's base Frame.

Invert
Is independent of any of the Objects within the FrameSet.

Nobject
Counts the number of active FrameSets.

RefCount
Counts the number of active pointers to the FrameSet (not to its current Frame).

Note that the set of attributes possessed by a FrameSet can vary, depending on the nature of its current Frame. For example, if the current Frame is a SkyFrame (§8), then the FrameSet will acquire an Equinox attribute from it which can be set, enquired, *etc.* However, if the current Frame is changed to be a basic Frame, which does not have an Equinox attribute, then this attribute will be absent from the FrameSet as well. Any attempt to reference it will then result in an error.

13.9 Extracting a Frame from a FrameSet

Although a FrameSet may be used in place of its current Frame in most situations, it is sometimes convenient to have direct access to a specified Frame within it. This may be obtained using the astGetFrame function, as follows:

```
frame = astGetFrame( frameset, AST__BASE );
```

This would return a pointer (not a copy) to the base Frame within the FrameSet. Note the use of AST__BASE (§13.5) as shorthand for the value of the FrameSet's Base attribute, which gives the base Frame's index.

13.10 Removing a Frame from a FrameSet

Removing a Frame from a FrameSet is straightforward and is performed using the astRemoveFrame function. You identify the Frame you wish to remove in the usual way, by giving its index within the FrameSet. For example, the following would remove the Frame with index 1:

```
astRemoveFrame( frameset, 1 );
```

The only restriction is that you cannot remove the last remaining Frame because a FrameSet must always contain at least one Frame. When a Frame is removed, the Frames which follow it are re-numbered (*i.e.* their indices are reduced by one) so as to preserve the sequence of consecutive Frame indices. The FrameSet's Nframe attribute is also decremented.

If appropriate, astRemoveFrame will modify the FrameSet's Base and/or Current attributes so that they continue to identify the same Frames as previously. If either the base or current Frame is removed, however, the corresponding attribute will become un-set, so that it reverts to its default value (§13.4) and therefore identifies an alternative Frame.

Note that it is quite permissible to remove any Frame from a FrameSet, even although other Frames may appear to depend on it. For example, in Figure 11, if Frame 1 were removed, the correct relationship between Frames 2 and 3 would still be preserved, although they would be re-numbered as Frames 1 and 2.

14 Higher Level Operations on FrameSets

14.1 Creating FrameSets with astConvert

Before considering the important subject of using FrameSets to convert between coordinate systems (§14.2), let us return briefly to reconsider the output generated by astConvert. We used this function earlier (§12), when converting between the coordinate systems represented by various kinds of Frame, and indicated that it returns a FrameSet to represent the coordinate conversion it identifies. We are now in a position to examine the structure of this FrameSet.

Take our earlier example (§12.1) of converting between the celestial coordinate systems represented by two SkyFrames:

```
#include "ast.h"
AstFrameSet *cvt;
AstSkyFrame *skyframe1, *skyframe2;

...

skyframe1 = astSkyFrame( "System=FK4-NO-E, Epoch=B1958, Equinox=B1960" );
skyframe2 = astSkyFrame( "System=Ecliptic, Equinox=J2010.5" );

cvt = astConvert( skyframe1, skyframe2, "" );
```

This will produce a pointer, “cvt”, to the FrameSet shown in Figure 12. As can be seen, this FrameSet contains just two Frames. The source Frame supplied to astConvert becomes its base Frame, while the destination Frame becomes its current Frame. (The FrameSet, of course, simply holds pointers to these Frames, rather than making copies.) The Mapping which relates the base Frame to the current Frame is the one which implements the required conversion.

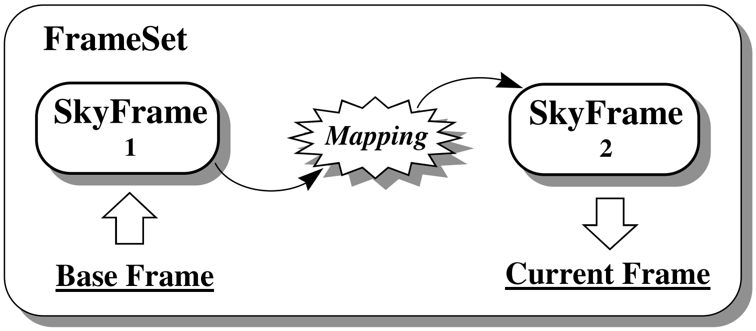


Figure 12: The FrameSet produced when astConvert is used to convert between the coordinate systems represented by two SkyFrames. The source SkyFrame becomes the base Frame, while the destination SkyFrame becomes the current Frame. The Mapping between them implements the required conversion.

As we noted earlier (§12.1), the FrameSet returned by astConvert may be used both as a Mapping and as a Frame to perform most of the functions you are likely to need. However, the Mapping may be extracted for use on its own if necessary, using astGetMapping (§13.7), for example:

```
AstMapping *mapping;

...

mapping = astGetMapping( cvt, AST__BASE, AST__CURRENT );
```

14.2 Converting between FrameSet Coordinate Systems

We now consider the process of converting between the coordinate systems represented by two FrameSets. This is a most important operation, as a subsequent example (§14.3) will show, and is illustrated in Figure 13. Recalling (§13.8) that a FrameSet will behave like its current Frame when necessary, conversion between two FrameSets is performed using astConvert (§12.1), but supplying pointers to FrameSets instead of Frames. The effect of this is to convert between the coordinate systems represented by the current Frames of each FrameSet:

```
AstFrameSet *frameseta, *framesetb;

...

cvt = astConvert( frameseta, framesetb, "SKY" );
```

When using FrameSets, we are presented with considerably more conversion options than when using Frames alone. This is because each current Frame is related to all the other Frames in its respective FrameSet. Therefore, if we can establish a link between any pair of Frames, one from each FrameSet, we can form a complete conversion path between the two current Frames (Figure 13).

This expanded range of options is, of course, precisely the intention. By connecting Frames together within a FrameSet, we have extended the range of coordinate systems that can be reached from any one of them. We are therefore no longer restricted to converting between Frames with the same Domain value (§7.12), but can go *via* a range of intermediate coordinate systems in order to make the connection we require. Transformation between different domains has therefore become possible because, in assembling the FrameSets, we provided the additional information needed to inter-relate them.

It is important to appreciate, however, that the choice of "missing link" is crucial in determining the conversion that results. Although each FrameSet may be perfectly self-consistent internally, this does not mean that all conversion paths through the combined network of Mappings are equivalent. Quite the contrary in fact: everything depends on where the inter-connecting link between the two FrameSets is made. In practice, there may be a large number of possible pairings of Frames and hence of possible links. Other factors must therefore be used to restrict the choice. These are:

1. Not every possible pairing of Frames is legitimate. For example, you cannot convert directly between a basic Frame and a SkyFrame which belong to different classes, so such pairings will be ignored.

2. In a similar way, you cannot convert directly between Frames with different Domain values (§7.12). If the Domain attribute is used consistently (typically only one Frame in each FrameSet will have a particular Domain value), then this further restricts the choice.

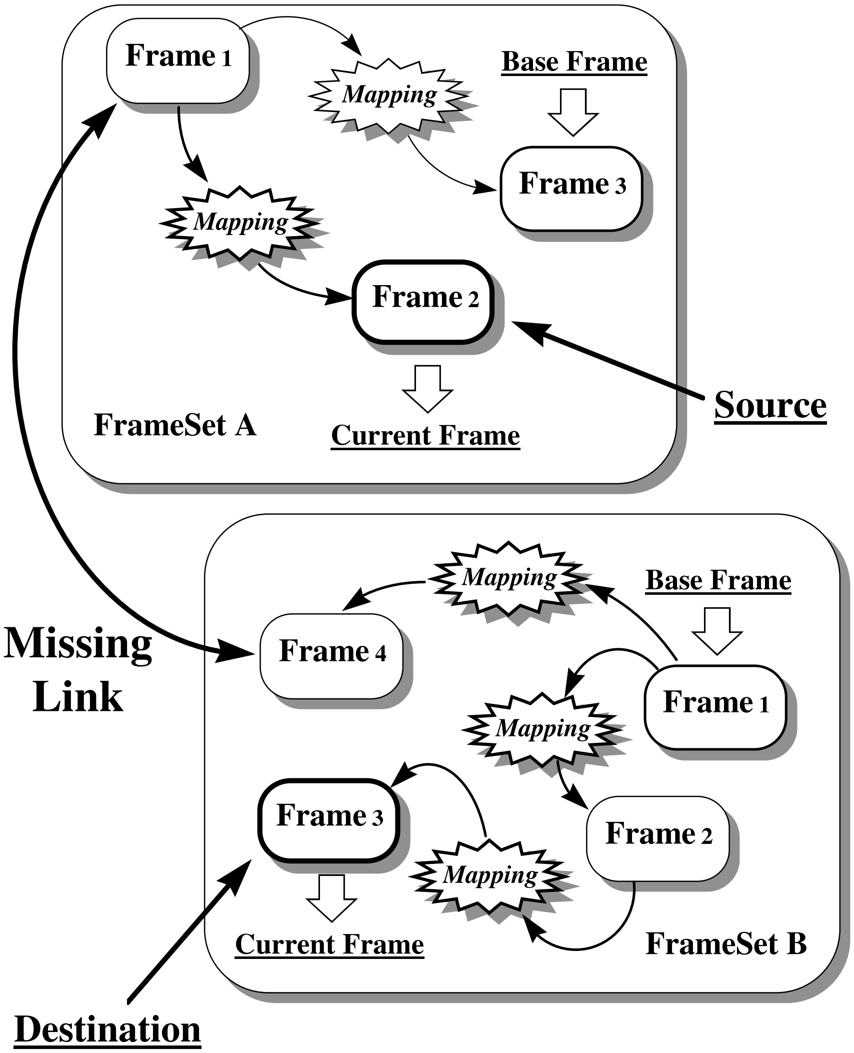


Figure 13: Conversion between two FrameSets is performed by establishing a link between a pair of Frames, one from each FrameSet. If conversion between these two Frames is possible, then a route for converting between the current Frames of both FrameSets can also be found. In practice, there may be many ways of pairing Frames to find the "missing link", so the Frames' Domain attribute may be used to narrow the choice.

3. The third argument of astConvert may then be used to specify explicitly which Domain value the paired Frames should have. You may also supply a comma-separated list of preferences here (see below).

4. If the above steps fail to uniquely identify the link, then the first suitable pairing of Frames is used, so that any ambiguity is resolved by the order in which Frames are considered for pairing (see the description of the astConvert function in Appendix B for details of the search order).[21]

In the example above we supplied the string "SKY" as the third argument of astConvert. This constitutes a request that a pair of Frames with the Domain value SKY (*i.e.* representing celestial coordinate systems) should be used to inter-relate the two FrameSets. Note that this does not specify which celestial coordinate system to use, but is a general request that the two FrameSets be inter-related using coordinates on the celestial sphere.

Of course, it may be that this request cannot be met because there may not be a celestial coordinate system in both FrameSets. If this is likely to happen, we can supply a list of preferences, or a *domain search path,* as the third argument to astConvert, such as the following:

```
cvt = astConvert( frameseta, framesetb, "SKY,PIXEL,GRID," );
```

Now, if the two FrameSets cannot be inter-related using the SKY domain, astConvert will attempt to use the PIXEL domain instead. If this also fails, it will try the GRID domain. A blank field in the domain search path (here indicated by the final comma) allows any Domain value to be used. This can be employed as a last resort when all else has failed.

If astConvert succeeds in identifying a conversion, it will return a pointer to a FrameSet (§14.1) in which the source and destination Frames are inter-connected by the required Mapping. In this case, of course, these Frames will be the current Frames of the two FrameSets, but in all other respects the returned FrameSet is the same as when converting between Frames.

Very importantly, however, astConvert may modify the FrameSets you are converting between. It does this, in order to indicate which pairing of Frames was used to inter-relate them, by changing the Base attribute for each FrameSet so that the Frame used in the pairing becomes its base Frame (§13.4).

Finally, note that astConvert may also be used to convert between a FrameSet and a Frame, or *vice versa.* If a pointer to a Frame is supplied for either the first or second argument, it will behave like a FrameSet containing only a single Frame.

14.3 Example—Registering Two Images

Consider two images which have been calibrated by attaching FrameSets to them, such that the base Frame of each FrameSet corresponds to the raw data grid coordinates of each image (the GRID domain of §7.13). Suppose, also, that these FrameSets contain an unknown number of other Frames, representing alternative world coordinate systems. What we wish to do is register

---

[21] If you find that how this ambiguity is resolved actually makes a difference to the conversion that results, then you have probably constructed a FrameSet which lacks internal self-consistency. For example, you might have two Frames representing indistinguishable coordinate systems but inter-related by a non-null Mapping.

these two images, such that we can transform from a position in the data grid of one into the corresponding position in the data grid of the other. This is a very practical example because images will typically be calibrated using FrameSets in precisely this way.

The first step will probably involve making a copy of both FrameSets (using astCopy—§4.13), since we will be modifying them. Let "frameseta" and "framesetb" be pointers to these copies. Since we want to convert between the base Frames of these FrameSets (*i.e.* their data grid coordinates), the next step is to make these Frames current. This is simply done by inverting both FrameSets, which interchanges their base and current Frames. astInvert will perform this task:

```
astInvert( frameseta );
astInvert( framesetb );
```

To identify the required conversion, we now use astConvert, supplying a suitable domain search path with which we would like our two images to be registered:

```
cvt = astConvert( frameseta, framesetb, "SKY,PIXEL,GRID" );
if ( cvt == AST__NULL ) {
   <no conversion was possible>
} else {
   <conversion was possible>
}
```

The effects of this are:

1. astConvert first attempts to register the two images on the celestial sphere (*i.e.* using the SKY domain). To do this, it searches for a celestial coordinate system, although not necessarily the same one, attached to each image. If it finds a suitable pair of coordinate systems, it then registers the images by matching corresponding positions on the sky.
2. If this fails, astConvert next tries to match positions in the PIXEL domain (§7.12). If it succeeds, the two images will then be registered so that their corresponding pixel positions correspond. If the PIXEL domain is offset from the data grid (as typically happens in data reduction systems which implement a "pixel origin"), then this will be correctly accounted for.
3. If this also fails, the GRID domain is finally used. This will result in image registration by matching corresponding points in the data grids used by both images. This means they will be aligned so that the first element their data arrays correspond.
4. If all of the above fail, astConvert will return the value AST__NULL. Otherwise a pointer to a FrameSet will be returned.

The resulting "cvt" FrameSet may then be used directly (§12.1) to convert between positions in the data grid of the first image and corresponding positions in the data grid of the second image.

To determine which domain was used to achieve registration, we can use the fact that the Base attribute of each FrameSet is set by astConvert to indicate which intermediate Frames were used. We can therefore simply invert either FrameSet (to make its base Frame become the current one) and then enquire the Domain value:

```
const char *domain;

...

astInvert( frameseta );
domain = astGetC( frameseta, "Domain" );
```

If conversion was successful, the result will be one of the strings “SKY”, “PIXEL” or “GRID”.

14.4 Re-Defining a FrameSet Coordinate System

As discussed earlier (§13.4), an important application of a FrameSet is to allow coordinate system information to be attached to entities such as images in order to calibrate them. In addition, one of the main objectives of AST is to simplify the propagation of such information through successive stages of data processing, so that it remains consistent with the associated image data.

In such a situation, the FrameSet's base Frame would correspond with the image's data grid coordinates and its other Frames (if any) with the various alternative world coordinate systems associated with the image. If the data processing being performed does not change the relationship between the image's data grid coordinates and any of the associated world coordinate systems, then propagation of the WCS information is straightforward and simply involves copying the FrameSet associated with the image.

If any of these relationships change, however, then corresponding changes must be made to the way Frames within the FrameSet are inter-related. By far the most common case occurs when the image undergoes some geometrical transformation resulting in “re-gridding” on to another data grid, but the same principles can be applied to any re-definition of a coordinate system.

To pursue the re-gridding example, we would need to modify our FrameSet to account for the fact that the image's data grid coordinate system (corresponding to the FrameSet's base Frame) has changed. Looking at the steps needed in detail, we might proceed as follows:

1. Create a Mapping which represents the relationship between the original data grid coordinate system and the new one.
2. Obtain a Frame to represent the new data grid coordinate system (we could re-use the original base Frame here, using astGetFrame to obtain a pointer to it).
3. Add the new Frame to the FrameSet, related to the original base Frame by the new Mapping. This Frame now represents the new data grid coordinate system and is correctly related to all the other Frames present.[22]
4. Remove the original base Frame (representing the old data grid coordinate system).
5. Make the new Frame the base Frame and restore the original current Frame.

The effect of these steps is to change the relationship between the base Frame and all the other Frames present. It is as if a new Mapping has been interposed between the Frame we want to alter and all the other Frames within the FrameSet (Figure 14).

---

[22] This is because any transformation to or from this new Frame must go *via* the base Frame representing the original data grid coordinate system, which we assume was correctly related to all the other Frames present.

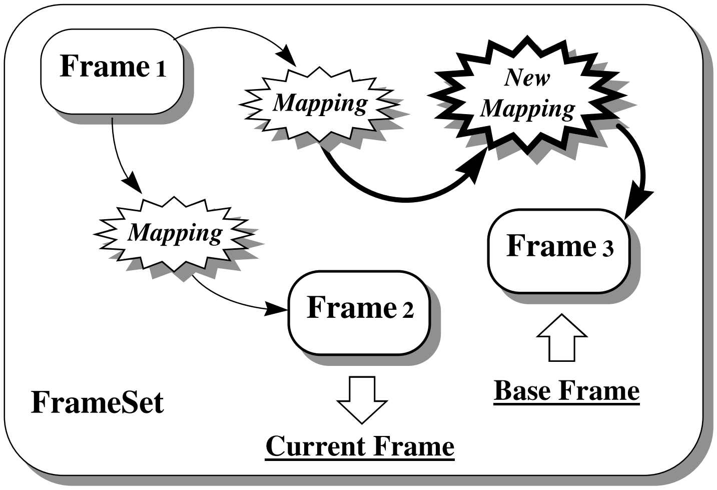


Figure 14: The effect of astRemapFrame is to interpose a Mapping between a nominated Frame within a FrameSet and the remaining contents of the FrameSet. This effectively "re-defines" the coordinate system represented by the affected Frame. It may be used to compensate (say) for geometrical changes made to an associated image. The inter-relationships between all the other Frames within the FrameSet remain unchanged.

Performing the steps above is rather lengthy, however, so the astRemapFrame function is provided to perform all of these operations in one go. A practical example of its use is given below (§14.5).

14.5 Example—Binning an Image

As an example of using astRemapFrame, consider a case where the pixels of a 2-dimensional image have been binned 2×2, so as to reduce the image size by a factor of two in each dimension. We must now modify the associated FrameSet to reflect this change to the image. Much the same process would be needed for any other geometrical change the image might undergo.

We first set up a Mapping (a WinMap in this case) which relates the data grid coordinates in the original image to those in the new one:

```
AstWinMap *winmap;
double ina[ 2 ] = { 0.5, 0.5 };
double inb[ 2 ] = { 2.5, 2.5 };
double outa[ 2 ] = { 0.5, 0.5 };
double outb[ 2 ] = { 1.5, 1.5 };

...

winmap = astWinMap( 2, ina, inb, outa, outb, "" );
```

Here, we have simply set up arrays containing the data grid coordinates of the bottom left and top right corners of the first element in the output image ("outa" and "outb") and the corresponding coordinates in the input image ("ina" and "inb"). astWinMap then creates a

WinMap which performs the required transformation. We do not need to know the size of the image.

We can then pass this WinMap to astRemapFrame. This modifies the relationship between our FrameSet's base Frame and the other Frames in the FrameSet, so that the base Frame represents the data grid coordinate system of the new image rather than the old one:

```
AstFrameSet *frameset;

...

astRemapFrame( frameset, AST__BASE, winmap );
```

Any other coordinate systems described by the FrameSet, no matter how many of these there might be, are now correctly associated with the new image.

14.6 Maintaining the Integrity of FrameSets

When constructing a FrameSet, you are provided with a framework into which you can place any combination of Frames and Mappings that you wish. There are relatively few constraints on this process and no checks are performed to see whether the FrameSet you construct makes physical sense. It is quite possible, for example, to construct a FrameSet containing two identical SkyFrames which are inter-related by a non-unit Mapping. AST will not object if you do this, but it makes no sense, because applying a non-unit Mapping to any set of celestial coordinates cannot yield positions that are still in the original coordinate system. If you use such a FrameSet to perform coordinate conversions, you are likely to get unpredictable results because the information in the FrameSet is corrupt.

It is, of course, your responsibility as a programmer to ensure the validity of any information which you insert into a FrameSet. Normally, this is straightforward and simply consists of formulating your problem correctly (a diagram can often help to clarify how coordinate systems are inter-related) and writing the appropriate bug-free code to construct the FrameSet. However, once you start to modify an existing FrameSet, there are new opportunities for corrupting it!

Consider, for example, a FrameSet whose current Frame is a SkyFrame. We can set a new value for this SkyFrame's Equinox attribute simply by using astSet on the FrameSet, as follows:

```
astSet( frameset, "Equinox=J2010" );
```

The effect of this will be to change the celestial coordinate system which the current Frame represents. You can see, however, that this has the potential to make the FrameSet corrupt unless corresponding changes are also made to the Mapping which relates this SkyFrame to the other Frames within the FrameSet. In fact, it is a general rule that any change to a FrameSet which affects its current Frame can potentially require corresponding changes to the FrameSet's Mappings in order to maintain its overall integrity.

Fortunately, once you have stored valid information in a FrameSet, AST will look after these details for you automatically, so that the FrameSet's integrity is maintained. In the example above, it would do this by appropriately re-mapping the current Frame (as if astRemapFrame had been used—§14.4) in response to the use of astSet. One way of illustrating this process is as follows:

```
AstSkyFrame *skyframe;

...

skyframe = astSkyFrame( "" );
frameSet = astFrameSet( skyframe );
astAddFrame( frameset, 1, astUnitMap( 2, "" ), skyframe );
```

This constructs a trivial FrameSet whose base and current Frames are both the same SkyFrame connected by a UnitMap. You can think of this as a "pipe" connecting two coordinate systems. At present, these two systems represent identical ICRS coordinates, so the FrameSet implements a unit Mapping. We can change the coordinate system on the current end of this pipe as follows:

```
astSet( frameset, "System=Ecliptic, Equinox=J2010" );
```

and the Mapping which the FrameSet implements would change accordingly. To change the coordinate system on the base end of the pipe, we might use:

```
astInvert( frameset );
astSet( frameset, "System=Galactic" );
astInvert( frameset );
```

The FrameSet would then convert between galactic and ecliptic coordinates.

Note that astSet is not the only function which has this effect: astClear behaves similarly, as also does astPermAxes (§7.9). If you need to circumvent this mechanism for any reason, this can be done by going behind the scenes and obtaining a pointer directly to the Frame you wish to modify. Consider the following, for example:

```
skyframe = astGetFrame( frameset, AST__CURRENT );
astSet( skyframe, "Equinox=J2010" );
skyframe = astAnnul( skyframe );
```

Here, astSet is applied to the SkyFrame pointer rather than the FrameSet pointer, so the usual checks on FrameSet integrity do not occur. The SkyFrame's Equinox attribute will therefore be modified without any corresponding change to the FrameSet's Mappings. In this case you must take responsibility yourself for maintaining the FrameSet's integrity, perhaps through appropriate use of astRemapFrame.

14.7 Merging FrameSets

As well as adding individual Frames to a FrameSet (§13.3), it is also possible to add complete sets of inter-related Frames which are contained within another FrameSet. This, of course, corresponds to the process of merging two FrameSets (Figure 15).

This process is performed by adding one FrameSet to another using astAddFrame, in much the same manner as when adding a new Frame to an existing FrameSet (§13.3). It is simply a matter of providing a FrameSet pointer, instead of a Frame pointer, for the 4th argument. In performing the merger you must, as usual, supply a Mapping, but in this case the Mapping should relate the current Frame of the FrameSet being added to one of the Frames already present. For example, you might perform the merger shown in Figure 15 as follows:

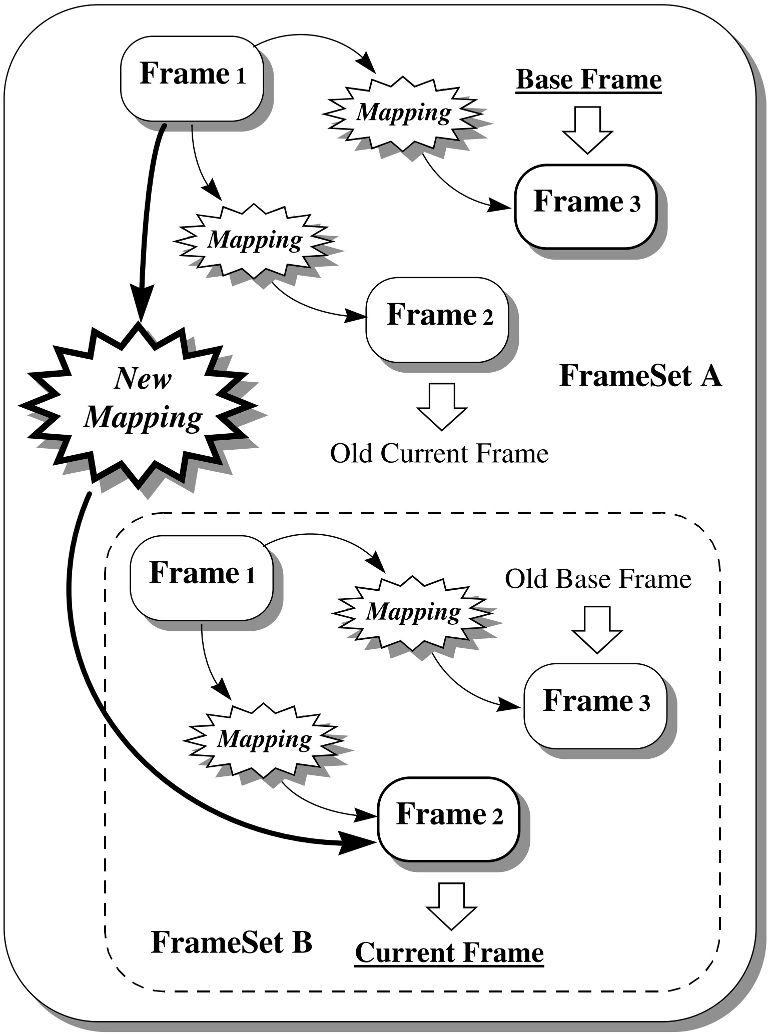


Figure 15: Two FrameSets in the process of being merged using astAddFrame. FrameSet B is being added to FrameSet A by supplying a new Mapping which inter-relates a nominated Frame in A (here number 1) and the current Frame of B. In the merged FrameSet, the Frames contributed by B will be re-numbered to become Frames 4, 5 and 6. The base Frame will remain unchanged, but the current Frame of B becomes the new current Frame. Note that FrameSet B itself is not altered by this process.

```
AstMapping *mapping;

...

astAddFrame( frameseta, 1, mapping, framesetb );
```

The Frames acquired by "frameseta" from the FrameSet being added ("framesetb") are renumbered so that they retain their original order and follow on consecutively after the Frames that were already present, whose indices remain unchanged. The base Frame of "frameseta" remains unchanged, but the current Frame of "framesetb" becomes its new current Frame. All the inter-relationships between Frames in both FrameSets remain in place and are preserved in the merged FrameSet.

Note that while this process modifies the first FrameSet ("frameseta"), it leaves the original contents of the one being added ("framesetb") unchanged.

15 Saving and Restoring Objects (Channels)

Facilities are provided by the AST library for performing input and output (I/O) with any kind of Object. This means it is possible to write any Object into various external representations for storage, and then to read these representations back in, so as to restore the original Object. Typically, an Object would be written by one program and read back in by another.

We refer to "external representations" in the plural because AST is designed to function independently of any particular data storage system. This means that Objects may need converting into a number of different external representations in order to be compatible with (say) the astronomical data storage system in which they will reside.

In this section, we discuss the basic I/O facilities which support external representations based on a textual format referred to as the AST "native format". These are implemented using a new kind of Object—a Channel. We will examine later how to use other representations, based on an XML format or on the use of FITS headers, for storing Objects. These are implemented using more specialised forms of Channel called XmlChan (§18) and FitsChan (§16).

15.1 The Channel Model

The best way to start thinking about a Channel is like a C file stream, and to think of the process of creating a Channel as that of opening a file and obtaining a FILE pointer. Subsequently, you can read and write Objects *via* the Channel.

This analogy is not quite perfect, however, because a Channel has, in principle, two "files" attached to it. One is used when reading, and the other when writing. These are termed the Channel's *source* and *sink* respectively. In practice, the source and sink may both be the same, in which case the analogy with the C file stream is correct, but this need not always be so. It is not necessarily so with the basic Channel, as we will now see (§15.2).

15.2 Creating a Channel

The process of creating a Channel is straightforward. As you might expect, it uses the constructor function astChannel:

```
#include "ast.h"
AstChannel *channel;

...

channel = astChannel( NULL, NULL, "" );
```

The first two arguments to astChannel specify the external source and sink that the Channel is to use. There arguments are pointers to C functions and we will examine their use in more detail later (§15.13 and §15.14).

In this very simple example we have supplied NULL pointers for both the source and sink functions. This requests the default behaviour, which means that textual input will be read from the program's standard input stream (typically, this means your keyboard) while textual output will go to the standard output stream (typically appearing on your screen). On UNIX systems, of course, either of these streams can easily be redirected to files. This default behaviour can be changed by assigning values to the Channel's SinkFile and/or SourceFile attributes. These attributes specify the paths to text files that are to be used in place of the standard input and output streams.

15.3 Writing Objects to a Channel

The process of saving Objects is very straightforward. You can simply write any Object to a Channel using the astWrite function, as follows:

```
int nobj;
AstObject *object;

...

nobj = astWrite( channel, object );
```

The effect of this will be to produce a textual description of the Object which will appear, by default, on your program's standard output stream. Any class of Object may be converted into text in this way.

astWrite returns a count of the number of Objects written. Usually, this will be one, unless the Object supplied cannot be represented. With a basic Channel all Objects can be represented, so a value of one will always be returned unless there has been an error. We will see later, however, that more specialised forms of Channel may impose restrictions on the kind of Object you can write (§17.2). In such cases, astWrite may return zero to indicate that the Object was not acceptable.

15.4 Reading Objects from a Channel

Before discussing the format of the output produced above (§15.3), let us consider how to read it back, so as to reconstruct the original Object. Naturally, we would first need to save the output in a file. We can do that either by using the SinkFile attribute, or (on UNIX systems), by redirecting standard output to a file using a shell command like:

```
program1 >file
```

Within a subsequent program, we can read this Object back in by using the astRead function, having first created a suitable Channel:

```
object = astRead( channel );
```

By default, this function will read from the standard input stream (the default source for a basic Channel), so we would need to ensure that our second program reads its input from the file in which the Object description is stored. On UNIX systems, we could again use a shell redirection command such as:

```
program2 <file
```

Alternatively, we could have assigned a value to the SinkFile attribute before invoking astRead.

15.5 Saving and Restoring Multiple Objects

I/O operations performed on a basic Channel are sequential. This means that if you write more than one Object to a Channel, each new Object's textual description is simply appended to the previous one. You can store any number of Objects in this way, subject only to the storage space you have available.

After you read an Object back from a basic Channel, the Channel is "positioned" at the end of that Object's textual description. If you then perform another read, you will read the next Object's textual description and therefore retrieve the next Object. This process may be repeated to read each Object in turn. When there are no more Objects to be read, astRead will return the value AST__NULL to indicate an *end-of-file.*

15.6 Validating Input

The pointer returned by astRead (§15.4) could identify any class of Object—this is determined entirely by the external data being read. If it is necessary to test for a particular class (say a Frame), this may be done as follows using the appropriate member of the astIsA<Class> family of functions:

```
int ok;

...

ok = astIsAFrame( object );
```

Note, however, that this will accept any Frame, so would be equally happy with a basic Frame or a SkyFrame. An alternative validation strategy would be to obtain the value of the Object's Class attribute and then test this character string, as follows:

```
#include <string.h>

...

ok = !strcmp( astGetC( object, "Class" ), "Frame" );
```

This would only accept a basic Frame and would reject a SkyFrame.

15.7 Storing an ID String with an Object

Occasionally, you may want to store a number of Objects and later retrieve them and use each for a different purpose. If the Objects are of the same class, you cannot use the Class attribute to distinguish them when you read them back (*c.f.* §15.6). Although relying on the order in which they are stored is a possible solution, this becomes complicated if some of the Objects are optional and may not always be present. It also makes extending your data format in future more difficult.

To help with this, every AST Object has an ID attribute and an Ident attribute, both of which allows you, in effect, to attach a textual identification label to it. You simply set the ID or Ident attribute before writing the Object:

```
astSet( object, "ID=Calibration" );
nobj = astWrite( channel, object );
```

You can then test its value after you read the Object back:

```
object = astRead( channel );
if ( !strcmp( astGetC( object, "ID" ), "Calibration" ) ) {
   <the Calibration Object has been read>
} else {
   <some other Object has been read>
}
```

The only difference between the ID and Ident attributes is that the ID attribute is unique to a particular Object and is lost if, for example, you make a copy of the Object. The Ident attrubute, on the other hand, is transferred to the new Object when a copy is made. Consequently, it is safest to set the value of the ID attribute immediately before you perform the write.

15.8 The Textual Output Format

Let us now examine the format of the textual output produced by writing an Object to a basic Channel (§15.3). To give a concrete example, suppose the Object in question is a SkyFrame, written out as follows:

```
AstSkyFrame *skyframe;

...

nobj = astWrite( channel, skyframe );
```

The output should then look like the following:

```
 Begin SkyFrame  # Description of celestial coordinate system
#   Title = "FK4 Equatorial Coordinates, no E-terms, Mean Equinox B1950.0, Epoch B1958.0"  # Title
    Naxes = 2  # Number of coordinate axes
#   Domain = "SKY"  # Coordinate system domain
#   Lbl1 = "Right Ascension"  # Label for axis 1
#   Lbl2 = "Declination"  # Label for axis 2
#   Uni1 = "hh:mm:ss.s"  # Units for axis 1
#   Uni2 = "ddd:mm:ss"  # Units for axis 2
#   Dir1 = 0  # Plot axis 1 in reverse direction (hint)
    Ax1 =  # Axis number 1
       Begin SkyAxis  # Celestial coordinate axis
       End SkyAxis
    Ax2 =  # Axis number 2
       Begin SkyAxis  # Celestial coordinate axis
       End SkyAxis
 IsA Frame  # Coordinate system description
    System = "FK4-NO-E"  # Celestial coordinate system type
    Epoch = 1958  # Besselian epoch of observation
#   Eqnox = 1950  # Besselian epoch of mean equinox
 End SkyFrame
```

You will notice that this output is designed both for a human reader, in that it is formatted, and also to be read back by a computer in order to reconstruct the SkyFrame. In fact, this is precisely the way that astShow works (§4.4), this function being roughly equivalent to the following use of a Channel:

```
channel = astChannel( NULL, NULL, "" );
(void) astWrite( channel, object );
channel = astAnnul( channel );
```

Some lines of the output start with a "#" comment character, which turns the rest of the line into a comment. These lines will be ignored when read back in by astRead. They typically contain default values, or values that can be derived in some way from the other data present, so that they do not actually need to be stored in order to reconstruct the original Object. They are provided purely for human information. The same comment character is also used to append explanatory comments to most output lines.

It is not sensible to attempt a complete description of this output format because every class of Object is potentially different and each can define how its own data should be represented. However, there are some basic rules, which mean that the following common features will usually be present:

1. Each Object is delimited by matching "Begin" and "End" lines, which also identify the class of Object involved.
2. Within each Object description, data values are represented by a simple "keyword = value" syntax, with one value to a line.
3. Lines beginning "IsA" are used to mark the divisions between data belonging to different levels in the class hierarchy (Appendix A). Thus, "IsA Frame" marks the end of data associated with the Frame class and the start of data associated with some derived class (a SkyFrame in the above example). "IsA" lines may be omitted if associated data values are absent and no confusion arises.
4. Objects may contain other Objects as data. This is indicated by an absent value, with the description of the data Object following on subsequent lines.
5. Indentation is used to clarify the overall structure.

Beyond these general principles, the best guide to what a particular line of output represents will generally be the comment which accompanies it together with a general knowledge of the class of Object being described.

15.9 Controlling the Amount of Output

It is not always necessary for the output from astWrite (§15.3) to be human-readable, so a Channel has attributes that allow the amount of detail in the output to be controlled.

The first of these is the integer attribute Full, which controls the extent to which optional, commented out, output lines are produced. By default, Full is zero, and this results in the standard style of output (§15.8) where default values that may be helpful to humans are included. To suppress these optional lines, Full should be set to −1. This is most conveniently done when the Channel is created, so that:

```
channel = astChannel( NULL, NULL, "Full=-1" );
(void) astWrite( channel, skyframe );
channel = astAnnul( channel );
```

would result in output containing only the essential information, such as:

```
 Begin SkyFrame  # Description of celestial coordinate system
    Naxes = 2  # Number of coordinate axes
    Ax1 =  # Axis number 1
       Begin SkyAxis  # Celestial coordinate axis
       End SkyAxis
    Ax2 =  # Axis number 2
       Begin SkyAxis  # Celestial coordinate axis
       End SkyAxis
 IsA Frame  # Coordinate system description
    System = "FK4-NO-E"  # Celestial coordinate system type
    Epoch = 1958  # Besselian epoch of observation
 End SkyFrame
```

In contrast, setting Full to +1 will result in additional output lines which will reveal every last detail of the Object's construction. Often this will be rather more than you want, especially for more complex Objects, but it can sometimes help when debugging programs. This is how a SkyFrame appears at this level of detail:

```
 Begin SkyFrame  # Description of celestial coordinate system
#   RefCnt = 1  # Count of active Object pointers
#   Nobj = 1  # Count of active Objects in same class
 IsA Object  # Astrometry Object
#   Nin = 2  # Number of input coordinates
#   Nout = 2  # Number of output coordinates
#   Invert = 0  # Mapping not inverted
#   Fwd = 1  # Forward transformation defined
#   Inv = 1  # Inverse transformation defined
#   Report = 0  # Don't report coordinate transformations
 IsA Mapping  # Mapping between coordinate systems
#   Title = "FK4 Equatorial Coordinates, no E-terms, Mean Equinox B1950.0, Epoch B1958.0"  # Title
    Naxes = 2  # Number of coordinate axes
#   Domain = "SKY"  # Coordinate system domain
#   Lbl1 = "Right Ascension"  # Label for axis 1
#   Lbl2 = "Declination"  # Label for axis 2
#   Sym1 = "RA"  # Symbol for axis 1
#   Sym2 = "Dec"  # Symbol for axis 2
#   Uni1 = "hh:mm:ss.s"  # Units for axis 1
#   Uni2 = "ddd:mm:ss"  # Units for axis 2
#   Dig1 = 7  # Individual precision for axis 1
#   Dig2 = 7  # Individual precision for axis 2
#   Digits = 7  # Default formatting precision
#   Fmt1 = "hms.1"  # Format specifier for axis 1
#   Fmt2 = "dms"  # Format specifier for axis 2
#   Dir1 = 0  # Plot axis 1 in reverse direction (hint)
#   Dir2 = 1  # Plot axis 2 in conventional direction (hint)
#   Presrv = 0  # Don't preserve target axes
#   Permut = 1  # Axes may be permuted to match
#   MinAx = 2  # Minimum number of axes to match
#   MaxAx = 2  # Maximum number of axes to match
#   MchEnd = 0  # Match initial target axes
#   Prm1 = 1  # Axis 1 not permuted
#   Prm2 = 2  # Axis 2 not permuted
    Ax1 =  # Axis number 1
       Begin SkyAxis  # Celestial coordinate axis
#         RefCnt = 1  # Count of active Object pointers
#         Nobj = 2  # Count of active Objects in same class
       IsA Object  # Astrometry Object
#         Label = "Angle on Sky"  # Axis Label
#         Symbol = "delta"  # Axis symbol
#         Unit = "ddd:mm:ss"  # Axis units
#         Digits = 7  # Default formatting precision
#         Format = "dms"  # Format specifier
#         Dirn = 1  # Plot in conventional direction
       IsA Axis  # Coordinate axis
#         Format = "dms"  # Format specifier
#         IsLat = 0  # Longitude axis (not latitude)
#         AsTime = 0  # Display values as angles (not times)
       End SkyAxis
    Ax2 =  # Axis number 2
       Begin SkyAxis  # Celestial coordinate axis
#         RefCnt = 1  # Count of active Object pointers
#         Nobj = 2  # Count of active Objects in same class
       IsA Object  # Astrometry Object
#         Label = "Angle on Sky"  # Axis Label
#         Symbol = "delta"  # Axis symbol
#         Unit = "ddd:mm:ss"  # Axis units
#         Digits = 7  # Default formatting precision
#         Format = "dms"  # Format specifier
#         Dirn = 1  # Plot in conventional direction
       IsA Axis  # Coordinate axis
#         Format = "dms"  # Format specifier
#         IsLat = 0  # Longitude axis (not latitude)
#         AsTime = 0  # Display values as angles (not times)
       End SkyAxis
 IsA Frame  # Coordinate system description
    System = "FK4-NO-E"  # Celestial coordinate system type
    Epoch = 1958  # Besselian epoch of observation
#   Eqnox = 1950  # Besselian epoch of mean equinox
 End SkyFrame
```

15.10 Controlling Commenting

Another way of controlling output from a Channel is *via* the boolean (integer) Comment attribute, which controls whether comments are appended to describe the purpose of each value. Comment has the value 1 by default but, if set to zero, will suppress these comments. This is normally appropriate only if you wish to minimise the amount of output, for example:

```
astSet( channel, "Full=-1, Comment=0" );
nobj = astWrite( channel, skyframe );
```

might result in the following more compact output:

```
Begin SkyFrame
   Naxes = 2
   Ax1 =
      Begin SkyAxis
      End SkyAxis
   Ax2 =
      Begin SkyAxis
      End SkyAxis
IsA Frame
   System = "FK4-NO-E"
   Epoch = 1958
End SkyFrame
```

15.11 Editing Textual Output

The safest advice about editing the textual output from astWrite (or astShow) is "don't!"—unless you know what you are doing.

Having given that warning, however, it is sometimes possible to make changes to the text, or even to write entire Object descriptions from scratch, and to read the results back in to construct new Objects. Normally, simple changes to numerical values are safest, but be aware that this is a back door method of creating Objects, so you are on your own! There are a number of potential pitfalls. In particular:

- astRead is intended for retrieving data written by astWrite and not for reading data input by humans. As such, the data validation provided is very limited and is certainly not foolproof. This makes it quite easy to construct Objects that are internally inconsistent by this means. In contrast, the normal programming interface incorporates numerous checks designed to make it impossible to construct invalid Objects. You should not necessarily think you have found a bug if your changes to an Object's textual description fail to produce the results you expected!

- In many instances the names associated with values in textual output will correspond with Object attributes. Sometimes, however, these names may differ from the attribute name. This is mainly because of length restrictions imposed by other common external formats, such as FITS headers. Some of the names used do not correspond with attributes at all.

- It is safest to change single numerical or string values. Beware of changing the size or shape of Objects (*e.g.* the number of axes in a Frame). Often, these values must match others stored elsewhere within the Object and changing them in a haphazard fashion will not produce useful results.

- Be wary about un-commenting default values. Sometimes this will work, but often these values are derived from other Objects stored more deeply in the structure and the proper place to insert a new value is not where the default itself appears.

15.12 Mixing Objects with other Text

By default, when you use astRead to read from a basic Channel (§15.4), it is assumed that you are reading a stream of text containing only AST Objects, which follow each other end-to-end. If any extraneous input data are encountered which do not appear to form part of the textual description of an Object, then an error will result. In particular, the first input line must identify the start of an Object description, so you cannot start reading half way through an Object.

Sometimes, however, you may want to store AST Object descriptions intermixed with other textual data. You can do this by setting the Channel's boolean (integer) Skip attribute to 1. This will cause every read to skip over extraneous data until the start of a new AST Object description, if any, is found. So long as your other data do not mimic the appearance of an AST Object description, the two sets of data can co-exist.

For example, by setting Skip to 1, the following complete C program will read all the AST Objects whose descriptions appear in the source of this document, ignoring the other text. astShow is used to display those found:

```
#include "ast.h"
main() {
   AstChannel *channel;
   AstObject *object;

   channel = astChannel( NULL, NULL, "Skip=1" );
   while ( ( object = astRead( channel ) ) != AST__NULL ) {
      astShow( object );
      object = astAnnul( object );
   }
   channel = astAnnul( channel );
}
```

15.13 Reading Objects from Files

Thus far, we have only considered the default behaviour of a Channel in reading and writing Objects through a program's standard input and output streams. We will now consider how to access Objects stored in files more directly.

The simple approach is to use the SinkFile and SourceFile attributes of the Channel. For instance, the following will read a pair of Objects from a text file called "fred.txt":

```
astSet( channel, "SourceFile=fred.txt" );
obj1 = astRead( channel );
obj2 = astRead( channel );
astClear( channel, "SourceFile" );
```

Note, the act of clearing the attribute tells AST that no more Objects are to be read from the file and so the file is then closed. If the attribute is not cleared, the file will remain open and further Objects can be read from it. The file will always be closed when the Channel is deleted.

This simple approach will normally be sufficient. However, because the AST library is designed to be used from more than one language, it has to be a little careful about reading and writing to files. This is due to incompatibilities that may exist between the file I/O facilities provided by different languages. If such incompatibilities prevent the above simple system being used, we need to adopt a system that off-loads all file I/O to external code.

What this means in practice is that if the above simple approach cannot be used, you must instead provide some simple C functions that perform the actual transfer of data to and from files and similar external data stores. The functions you provide are supplied as the source and/or sink function arguments to astChannel when you create a Channel (§15.2). An example is the best way to illustrate this.

Consider the following simple function called Source. It reads a single line of text from a C input stream and returns a pointer to it, or NULL if there is no more input:

```
#include <stdio.h>
#define LEN 200
static FILE *input_stream;

const char *Source( void ) {
   static char buffer[ LEN + 2 ];
   return fgets( buffer, LEN + 2, input_stream );
}
```

Note that the input stream is a static variable which we will also access from our main program. This might look something like this (omitting error checking for brevity):

```
/* Open the input file. */
input_stream = fopen( "infile.ast", "r" );

/* Create a Channel and read an Object from it. */
channel = astChannel( Source, NULL, "" );
object = astRead( channel );

...

/* Annul the Channel and close the file when done. */
channel = astAnnul( channel );
(void) fclose( input_stream );
```

Here, we first open the required input file, saving the resulting FILE pointer. We then pass a pointer to our Source function as the first argument to astChannel when creating a new Channel. When we read an Object from this Channel with astRead, the Source function will be called to obtain the textual data from the file, the end-of-file being detected when this function returns NULL.

Note, if a value is set for the SourceFile attribute, the astRead function will ignore any source function specified when the Channel was created.

15.14 Writing Objects to Files

As for reading, writing Objects to files can be done in two different ways. Again, the simple approach is to use the SinkFile attribute of the Channel. For instance, the following will write a pair of Objects to a text file called “fred.txt”:

```
   astSet( channel, "SinkFile=fred.txt" );
   nobj = astWrite( channel, object1 );
   nobj = astWrite( channel, object2 );
   astClear( channel, "SinkFile" );
```

Note, the act of clearing the attribute tells AST that no more output will be written to the file and so the file is then closed. If the attribute is not cleared, the file will remain open and further Objects can be written to it. The file will always be closed when the Channel is deleted.

If the details of the language's I/O system on the computer you are using means that the above approach cannot be used, then we can write a Sink function, that writes a line of output text to a file, and use it in basically the same way as the Source function in the previous section (§15.13):

```
static FILE *output_stream;

void Sink( const char *line ) {
   (void) fprintf( output_stream, "%s\n", line );
}
```

Note that we must supply the final newline character ourselves.

In this case, our main program would supply a pointer to this Sink function as the second argument to astChannel, as follows:

```
/* Open the output file. */
output_stream = fopen( "outfile.ast", "w" );

/* Create a Channel and write an Object to it. */
channel = astChannel( Source, Sink, "" );
nobj = astWrite( channel, object );

   ...

/* Annul the Channel and close the file when done. */
channel = astAnnul( channel );
(void) fclose( output_stream );
```

Note that we can specify a source and/or a sink function for the Channel, and that these may use either the same file, or different files according to whether we are reading or writing. AST has no knowledge of the underlying file system, nor of file positioning. It just reads and writes sequentially. If you wish, for example, to reposition a file at the beginning in between reads and writes, then this can be done directly (and completely independently of AST) using standard C functions.

If an error occurs in your source or sink function, you can communicate this to the AST library by setting its error status to any error value using astSetStatus (§4.15). This will immediately terminate the read or write operation.

Note, if a value is set for the SinkFile attribute, the astWrite function will ignore any sink function specified when the Channel was created.

15.15 Reading and Writing Objects to other Places

It should be obvious from the above (§15.13 and §15.14) that a Channel's source and sink functions provide a flexible means of intercepting textual data that describes AST Objects as it flows in and out of your program. In fact, you might like to regard a Channel simply as a filter for converting AST Objects to and from a stream of text which is then handled by your source and sink functions, where the real I/O occurs.

This gives you the ability to store AST Objects in virtually any data system, so long as you can convert a stream of text into something that can be stored (it need no longer be text) and retrieve it again. There is generally no need to retain comments. Other possibilities, such as inter-process and network communication, could also be implemented *via* source and sink functions in basically the same way.

16 Storing AST Objects in FITS Headers (FitsChans)

A FITS header is a sequence of 80-character strings, formatted according to particular rules defined by the Flexible Image Transport System (FITS). FITS[23] is a widely-used standard for data interchange in astronomy and has also been adopted as a data processing format in some astronomical data reduction systems. The individual 80-character strings in a FITS header are usually called *cards* or *header cards* (for entirely anachronistic reasons).

A sequence of FITS cards appears as a header at the start of every FITS data file, and sometimes also at other points within it, and is used to provide ancillary information which qualifies or describes the main array of data stored in the file. As such, FITS headers are prime territory for storing information about the coordinate systems associated with data held in FITS files.

In this section, we will examine how to store information in FITS headers directly in the form of AST Objects—a process which is supported by a specialised class of Channel called a FitsChan. Our discussion here will turn out to be a transitional step that emphasises the similarities between a FitsChan and a Channel (§15). At the same time, it will prepare us for the next section (§17), where we will examine how to use a FitsChan to tackle some of the more difficult problems that FITS headers can present.

16.1 The Native FITS Encoding

As it turns out, we are not the first to have thought of storing WCS information in FITS headers. In fact, the original FITS standard (1981 vintage) defined a set of header keywords for this purpose which have been widely used, although they have proved too limited for many practical purposes.

At the time of writing, a number of different ways of using FITS headers for storing WCS information are in use, most (although not all) based on the original standard. We will refer to these alternative ways of storing the information as FITS *encodings* but will defer a discussion of their advantages and limitations until the next section (§17).

Here, we will examine how to store AST Objects directly in FITS headers. In effect, this defines a new encoding, which we will term the *native encoding.* This is a special kind of encoding, because not only does it allow us to associate conventional WCS calibration information with FITS data, but it also allows any other information that can be expressed in terms of AST Objects to be stored as well. In fact, the native encoding provides us with facilities roughly analogous to those of the Channel (§15)—*i.e.* a lossless way of transferring AST Objects from program to program—but based on FITS headers instead of free-format text.

16.2 The FitsChan Model

I/O between AST Objects and FITS headers is supported by a specialised form of Channel called a FitsChan. A FitsChan contains a buffer which may hold any number, including zero, of FITS header cards. This buffer forms a workspace in which you can assemble FITS cards and manipulate them before writing them out to a file.

By default, when a FitsChan is first created, it contains no cards and there are five ways of inserting cards into it:

---

[23] http://fits.gsfc.nasa.gov/

1. You may add cards yourself, one at a time, using astPutFits (§16.8).
2. You may add cards yourself, supplying all cards concatenated into a single string, using astPutCards (§16.9).
3. You may write an AST Object to the FitsChan (using astWrite), which will have the effect of creating new cards within the FitsChan which describe the Object (§16.5).
4. You may assign a value to the SourceFile attribute of the FitsChan. The value should be the path to a text file holding a set of FITS header cards, one per line. When the SourceFile value is set (using astSetC or astSet), the file is opened and the headers copied from it into the FitsChan. The file is then imemdiately closed.
5. You may specify a source function which reads data from some external store of FITS cards, just like the source associated with a basic Channel (§15.13). If you supply a source function, it will be called when the FitsChan is created in order to fill it with an initial set of cards (§16.14).

There are also four ways of removing cards from a FitsChan:

1. You may delete cards yourself, one at a time, using astDelFits (§16.13).
2. You may read an AST Object from the FitsChan (using astRead), which will have the effect of removing those cards from the FitsChan which describe the Object (§16.10).
3. You may assign a value to the FitsChan's SinkFile attribute. When the FitsCHan is deleted, any remaining headers are written out to a text file with path equal to the value of the SinkFile attribute.
4. Alternatively, you may specify a sink function which writes data to some external store of FITS cards, just like the sink associated with a basic Channel (§15.14). If you supply a sink function, it will be called when the FitsChan is deleted in order to write out any FITS cards that remain in it (§16.14). Note, the sink function is not called if the SinkFile attribute has been set.

Note, in particular, that reading an AST Object from a FitsChan is *destructive.* That is, it deletes the FITS cards that describe the Object. The reason for this is explained in §17.5.

In addition to the above, you may also read individual cards from a FitsChan using the function astFindFits (which is not destructive). This is the main means of writing out FITS cards if you have not supplied a sink function. astFindFits also provides a means of searching for particular FITS cards (by keyword, for example) and there are other facilities for overwriting cards when required (§16.13).

16.3 Creating a FitsChan

The FitsChan constructor function, astFitsChan, is straightforward to use:

```
#include "ast.h"
AstFitsChan *fitschan;

...

fitschan = astFitsChan( NULL, NULL, "Encoding=NATIVE" );
```

Here, we have omitted any source or sink functions by supplying NULL pointers for the first two arguments. We have also initialised the FitsChan's Encoding attribute to NATIVE. This indicates that we will be using the native encoding (§16.1) to store and retrieve Objects. If this was left unspecified, the default would depend on the FitsChan's contents. An attempt is made to use whatever encoding appears to have been used previously. For an empty FitsChan, the default is NATIVE, but it does no harm to be sure.

16.4 Addressing Cards in a FitsChan

Because a FitsChan contains an ordered sequence of header cards, a mechanism is needed for addressing them. This allows you to specify where new cards are to be added, for example, or which card is to be deleted.

This role is filled by the FitsChan's integer Card attribute, which gives the index of the *current card* in the FitsChan. You can nominate any card you like to be current, simply by setting a new value for the Card attribute, for example:

```
int icard;

...

astSetI( fitschan, "Card", icard )
```

where "icard" contains the index of the card on which you wish to operate next. Some functions will update the Card attribute as a means of advancing through the sequence of cards, when reading them for example, or to indicate which card matches a search criterion.

The default value for Card is one, which is the index of the first card. This means that you can "rewind" a FitsChan to access its first card by clearing the Card attribute:

```
astClear( fitschan, "Card" );
```

The total number of cards in a FitsChan is given by the integer Ncard attribute. This is a read-only attribute whose value is automatically updated as you add or remove cards. It means you can address all the cards in sequence using a loop such as the following:

```
int ncard;

...

ncard = astGetI( fitschan, "Ncard" );
for ( icard = 1; icard <= ncard; icard++ ) {
   astSetI( fitschan, "Card", icard );
   <access the current card>
}
```

However, it is usually possible to write slightly tidier loops based on the astFindFits function described later (§16.6 and §16.13).

If you set the Card attribute to a value larger than Ncard, the FitsChan is regarded as being positioned at its *end-of-file.* In this case there is no current card and an attempt to obtain a value for the Card attribute will always return the value Ncard + 1. When a FitsChan is empty, it is always at the end-of-file.

16.5 Writing Native Objects to a FitsChan

Having created an empty FitsChan (§16.3), you can write any AST Object to it in the native encoding using the astWrite function. Let us assume we are writing a SkyFrame,[24] as follows:

```
AstSkyFrame *skyframe;
int nobj;

...

nobj = astWrite( fitschan, skyframe );
```

Since we have selected the native encoding (§16.1), there are no restrictions on the class of Object we may write, so astWrite should always return a value of one, unless an error occurs. Unlike a basic Channel (§15.3), this write operation will not produce any output from our program. The FITS headers produced are simply stored inside the FitsChan.

After this write operation, the Ncard attribute will be updated to reflect the number of new cards added to the FitsChan and the Card attribute will point at the card immediately after the last one written. Since our FitsChan was initially empty, the Card attribute will, in this example, point at the end-of-file (§16.4).

The FITS standard imposes a limit of 68 characters on the length of strings which may be stored in a single header card. Sometimes, a description of an AST Object involves the use of strings which exceed this limit (*e.g.* a Frame title can be of arbitrary length). If this occurs, the long string will be split over two or more header cards. Each "continuation" card will have the keyword `CONTINUE` in columns 1 to 8, and will contain a space in column 9 (instead of the usual equals sign). An ampersand ("`&`") is appended to the end of each of the strings (except the last one) to indicate that the string is continued on the next card.

Note, this splitting of long strings over several cards only occurs when writing AST Objects to a FitsChan using the astWrite function and the *native* encoding. If a long string is stored in a FitsChan using (for instance) the astPutFits or astPutCards function, it will simply be truncated.

16.6 Extracting Individual Cards from a FitsChan

To examine the contents of the FitsChan after writing the SkyFrame above (§16.5), we must write a simple loop to extract each card in turn and print it out. We must also remember to rewind the FitsChan first, *e.g.* using astClear. The following loop would do:

---

[24] More probably, you would want to write a FrameSet, but for purposes of illustration a SkyFrame contains a more manageable amount of data.

```
#include <stdio.h>
char card[ 81 ];

...

astClear( fitschan, "Card" );
while ( astFindFits( fitschan, "%f", card, 1 ) ) (void) printf( "%s\n", card );
```

Here, we have used the astFindFits function to find a FITS card by keyword. It is given a keyword template of "%f", which matches any FITS keyword, so it always finds the current card, which it returns. Its fourth argument is set to 1, to indicate that the Card attribute should be incremented afterwards so that the following card will be found the next time around the loop. astFindFits returns zero when it reaches the end-of-file and this terminates the loop.

If we were storing the FITS headers in an output FITS file instead of printing them out, we might use a loop like this but replace "printf" with a suitable data storage operation. This would only be necessary if we had not provided a sink function for the FitsChan (§16.14).

16.7 The Native FitsChan Output Format

If we print out the FITS header cards describing the SkyFrame we wrote earlier (§16.5), we should obtain something like the following:

```
COMMENT AST ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++ AST
COMMENT AST            Beginning of AST data for SkyFrame object             AST
COMMENT AST ................................................................ AST
BEGAST_A= 'SkyFrame'           / Description of celestial coordinate system
NAXES_A =                    2 / Number of coordinate axes
AX1_A   = '        '           / Axis number 1
BEGAST_B= 'SkyAxis '           / Celestial coordinate axis
ENDAST_A= 'SkyAxis '           / End of object definition
AX2_A   = '        '           / Axis number 2
BEGAST_C= 'SkyAxis '           / Celestial coordinate axis
ENDAST_B= 'SkyAxis '           / End of object definition
ISA_A   = 'Frame   '           / Coordinate system description
SYSTEM_A= 'FK4-NO-E'           / Celestial coordinate system type
EPOCH_A =               1958.0 / Besselian epoch of observation
ENDAST_C= 'SkyFrame'           / End of object definition
COMMENT AST ................................................................ AST
COMMENT AST               End of AST data for SkyFrame object                AST
COMMENT AST ---------------------------------------------------------------- AST
```

As you can see, this resembles the information that would be written to a basic Channel to describe the same SkyFrame (§15.8), except that it has been formatted into 80-character header cards according to FITS conventions.

There are also a number of other differences worth noting:

1. There is no unnecessary information about default values provided for the benefit of the human reader. This is because the Full attribute for a FitsChan defaults to −1, thus

suppressing this information (*c.f.* §15.9). You can restore the information if you wish by setting Full to 0 or +1, in which case additional COMMENT cards will be generated to hold it.

2. The information is not indented, because FITS does not allow this. However, if you change the Full attribute to 0 or +1, comments will be included that are intended to help break up the sequence of headers and highlight its structure. This will probably only be of use if you are attempting to track down a problem by examining the FITS cards produced in detail.
3. The FITS keywords which appear to the left of the "=" signs have additional characters ("_A", "_B", *etc.*) appended to them. This is done in order to make each keyword unique.

This last point is worth further comment and is necessary because the FITS standard only allows for certain keywords (such as COMMENT and HISTORY) to appear more than once. astWrite therefore appends an arbitrary sequence of two characters to each new keyword it generates in order to ensure that it does not duplicate any already present in the FitsChan.

The main risk from not following this convention is that some software might ignore (say) all but the last occurrence of a keyword before passing the FITS headers on. Such an event is unlikely, but would obviously destroy the information present, so astWrite enforces the uniqueness of the keywords it uses. The extra characters added are ignored when the information is read back.

As with a basic Channel, you can also suppress the comments produced in a FitsChan by setting the boolean (integer) Comment attribute to zero (§15.10). However, FITS headers are traditionally generously commented, so this is not recommended.

16.8 Adding Individual Cards to a FitsChan

To insert individual cards into a FitsChan, prior to reading them back as Objects for example, you should use the astPutFits function. You can insert a card in front of the current one as follows:

```
astPutFits( fitschan, card, 0 );
```

where the third argument of zero indicates that the current card should not be overwritten. Note that facilities are not provided by AST for formatting the card contents.

After inserting a card, the FitsChan's Card attribute points at the original Card, or at the end-of-file if the FitsChan was originally empty. Entering a sequence of cards is therefore straightforward. If "cards" is an array of pointers to strings containing FITS header cards and "ncards" is the number of cards, then a loop such as the following will insert the cards in sequence into a FitsChan:

```
#define MAXCARD 100
char *cards[ MAXCARD ];
int ncard;

...

for ( icard = 0; icard < ncard; icard++ ) astPutFits( fitschan, cards[ icard ], 0 );
```

The string containing a card need not be null terminated if it is at least 80 characters long (we have not allocated space for the strings themselves in this brief example).

Note that astPutFits enforces the validity of a FitsChan by rejecting any cards which do not adhere to the FITS standard. If any such cards are detected, an error will result.

16.9 Adding Concatenated Cards to a FitsChan

If you have all your cards concatenated together into a single long string, each occupying 80 characters (with no delimiters), you can insert them into a FitsChan in a single call using astPutCards. This call first empties the supplied FitsChan of any existing cards, then inserts the new cards, and finally rewinds the FitsChan so that a subsequent call to astRead will start reading from the first supplied card. The astPutCards function uses astPutFits internally to interpret and store each individual card, and so the caveats in §16.8 should be read.

For instance, if you are using the CFITSIO library for access to FITS files, you can use the CFITSIO fits_hdr2str function to obtain a string suitable for passing to astPutCards:

```
if( !fits_hdr2str( fptr, 0, NULL, 0, &header, &nkeys, &status ) )
   fitschan = astFitsChan( NULL, NULL, "" );
   astPutCards( fitschan, header );
   header = free( header );
   wcsinfo = astRead( fitschan );

   ...
}
```

16.10 Reading Native Objects From a FitsChan

Once you have stored a FITS header description of an Object in a FitsChan using the native encoding (§16.5), you can read it back using astRead in much the same way as with a basic Channel (§15.4). Similar comments about validating the Object you read also apply (§15.6). If you have just written to the FitsChan, you must remember to rewind it first:

```
AstObject *object;

...

astClear( fitschan, "Card" );
object = astRead( fitschan );
```

An important feature of a FitsChan is that read operations are destructive. This means that if an Object description is found, it will be consumed by astRead which will remove all the cards involved, including associated COMMENT cards, from the FitsChan. Thus, if you write an Object to a FitsChan, rewind, and read the same Object back, you should end up with the original FitsChan contents. If you need to circumvent this behaviour for any reason, it is a simple matter to make a copy of a FitsChan using astCopy (§4.13). If you then read from the copy, the original FitsChan will remain untouched.

After a read completes, the FitsChan's Card attribute identifies the card immediately following the last card read, or the end-of-file of there are no more cards.

Since the *native* encoding is being used, any long strings involved in the object description will have been split into two or more adjacent contuation cards when the Object was stored in the header using function astWrite. The astRead function reverses this process by concatenating any such adjacent continuation cards to re-create the original long string.

16.11 Saving and Restoring Multiple Objects in a FitsChan

When using the native FITS encoding, multiple Objects may be stored and all I/O operations are sequential. This means that you can simply write a sequence of Objects to a FitsChan. After each write operation, the Card attribute will be updated so that the next write appends the next Object description to the previous one.

If you then rewind the FitsChan, you can read the Objects back in the original order. Reading them back will, of course, remove their descriptions from the FitsChan (§16.10) but the behaviour of the Card attribute is such that successive reads will simply return each Object in sequence.

The only thing that may require care, given that a FitsChan can always be addressed randomly by setting its Card attribute, is to avoid writing one Object on top of another. For obvious reasons, the Object descriptions in a FitsChan must remain separate if they are to make sense when read back.

16.12 Mixing Native Objects with Other FITS Cards

Of course, any real FITS header will contain other information besides AST Objects, if only the mandatory FITS cards that must accompany all FITS data. When FITS headers are read in from a real dataset, therefore, any native AST Object descriptions will be inter-mixed with many other cards.

Because this is the normal state of affairs, the boolean (integer) Skip attribute for a FitsChan defaults to one. This means that when you read an Object From a FitsChan, any irrelevant cards will simply be skipped over until the start of the next Object description, if any, is found. If you start reading part way through an Object description, no error will result. The remainder of the description will simply be skipped.

Setting Skip to zero will change this behaviour to resemble that of a basic Channel (§15.12), where extraneous data are not permitted by default, but this will probably rarely be useful.

16.13 Finding and Changing Cards in a FitsChan

You can search for, and retrieve, particular cards in a FitsChan by keyword, using the function astFindFits. This performs a search, starting at the current card, until it finds a card whose keyword matches the template you supply, or the end-of-file is reached.

If a suitable card is found, astFindFits optionally returns the card's contents and then sets the FitsChan's Card attribute either to identify the card found, or the one following it. The way you want the Card attribute to be set is indicated by the final boolean (int) argument to astFindFits. A value of one is returned to indicate success. If a suitable card cannot be found, astFindFits returns a value of zero to indicate failure and sets the FitsChan's Card attribute to the end-of-file.

Requesting that the Card attribute be set to indicate the card that astFindFits finds is useful if you want to replace that card with a new one, as in this example:

```
char newcard[ 81 ];

...

(void) astFindFits( fitschan, "AIRMASS", NULL, 0 );
astPutFits( fitschan, newcard, 1 );
```

Here, astFindFits is used to search for a card with the keyword AIRMASS, with a NULL pointer being given to indicate that we do not want the card's contents returned. If the card is found, astPutFits then overwrites it with a new card. Otherwise, the Card attribute ends up pointing at the end-of-file and the new card is simply appended to the end of the FitsChan.

A similar approach can be used to delete selected cards from a FitsChan using astDelFits, which deletes the current card:

```
if ( astFindFits( fitschan, "BSCALE", NULL, 0 ) ) astDelFits( fitschan );
```

This deletes the first card, if any, with the BSCALE keyword.

Requesting that astFindFits increments the Card attribute to identify the card following the one found is more useful when writing loops. For example, the following loop extracts each card whose keyword matches the template "CD%6d" (that is, "CD" followed by six decimal digits):

```
while ( astFindFits( fitschan, "CD%6d", card, 1 ) {
   <process the card's contents>
}
```

For further details of keyword templates, see the description of astFindFits in Appendix B.

16.14 Source and Sink Functions for FitsChans

The use of source and sink functions with a FitsChan is optional. This is because you can always arrange to explicitly fill a FitsChan with FITS cards (§16.8 and §16.9) and you can also extract any cards that remain and write them out yourself (§16.6) before you delete the FitsChan.

If you choose to use these functions, however, they behave in a very similar manner to those used by a Channel (§15.13 and §15.14). You supply pointers to these functions, as arguments to the constructor function astFitsChan when you create the FitsChan (§16.3). The source function is invoked implicitly at this point to fill the FitsChan with FITS cards and the FitsChan is then rewound, so that the first card becomes current. The sink function is automatically invoked later, when the FitsChan is deleted, in order to write out any cards that remain in it.

The only real difference between the source and sink functions for a FitsChan and a basic Channel is that FITS cards are limited in length to 80 characters, so the choice of buffer size is simplified. The "Source" and "Sink" functions in §15.13 and §15.14 could therefore be used to access FITS headers stored in text files simply by changing LEN to be 80. If you were not accessing a text file, however, appropriate changes to the I/O statements would be needed since the separating newline characters would be absent. The details obviously depend on the format of the file you are handling, which need not necessarily be a true FITS file.

17 Using Foreign FITS Encodings

We saw in the previous section (§16) how to store and retrieve any kind of AST Object in a FITS header by using a FitsChan. To achieve this, we set the FitsChan's Encoding attribute to NATIVE. However, the Objects we wrote could then only be read back by other programs that use AST.

In practice, we will also encounter FITS headers containing WCS information written by other software systems. We will probably also need to write FITS headers in a format that can be understood by these systems. Indeed, this interchange of data is one of the main reasons for the existence of FITS, so in this section we will examine how to accommodate these requirements.

17.1 The Foreign FITS Encodings

As mentioned previously (§16.1), there are a number of conventions currently in use for storing WCS information in FITS headers, which we call *encodings.* Here, we are concerned with those encodings defined by software systems other than AST, which we term *foreign encodings.*

Currently, AST supports six foreign encodings, which may be selected by setting the Encoding attribute of a FitsChan to one of the following (character string) values:

DSS

This encoding stores WCS information using the convention developed at the Space Telescope Science Institute for the Digitised Sky Survey (DSS) astrometric plate calibrations. DSS images which use this convention are widely available and it is understood by a number of important and well-established astronomy applications.

However, the calibration model used (based on a polynomial fit) is not easily applicable to other types of data and creating the polynomial coefficients needed to calibrate your own images can prove difficult. For this reason, the DSS encoding is probably best viewed as a "read-only" format. It is possible, however, to read in WCS information using this encoding and then to write it back out again, so long as only minor changes have been made.

FITS-WCS

This encoding is very important because it is based on a new FITS standard which should, for the first time, address the problem of celestial coordinate systems in a proper manner, by considerably extending the original FITS standard.

The conventions used are described in a series of papers by E.W. Greisen, M. Calabretta, *et. al.*, often referred to as the "FITS-WCS papers". They are described at http://fits.gsfc.nasa.gov/fits_wcs.html. Now that the first two papers in this series have been agreed, this encoding should be understood by any FITS-WCS compliant software and it is likely to be adopted widely for FITS data in future. For details of the coverage of these conventions provided by the FitsChan class, see Appendix G.

FITS-IRAF

This encoding is based on the conventions described in the document "World Coordinate Systems Representations Within the FITS Format" by R.J. Hanisch and D.G. Wells, 1988.[25] It is employed by the IRAF data analysis facility, so its use will facilitate data exchange with IRAF. This encoding is in effect a sub-set of the current FITS-WCS encoding.

FITS-PC

This encoding is based on a previous version of the proposed new FITS WCS standard which used `PCjjjjiii` and `CDELTj` keywords to describe axis rotation and scaling. Versions of AST prior to V1.5 used this scheme for the FITS-WCS encoding. As of V1.5, FITS-WCS uses `CDi_j` keywords instead.[26] The FITS-PC encoding is included in AST V1.5 only to allow FITS-WCS data created with previous versions to be read. It should not, in general, be used to create new data sets.

FITS-AIPS

This encoding is based on the conventions described in the document "Non-linear Coordinate Systems in AIPS" by Eric W. Greisen (revised 9th September, 1994).[27] It is currently employed by the AIPS data analysis facility, so its use will facilitate data exchange with AIPS. This encoding uses `CROTAi` and `CDELTi` keywords to describe axis rotation and scaling.

FITS-AIPS++

Encodes coordinate system information in FITS header cards using the conventions used by the AIPS++ project. This is an extension of FITS-AIPS which includes some of the features of FITS-PC and FITS-IRAF.

For more detail about the above encodings, see the description of the Encoding attribute in Appendix C.

17.2 Limitations of Foreign Encodings

The foreign encodings available for storing WCS information in FITS headers have a number of limitations when compared with the native encoding of AST Objects (§16). The main ones are:

1. Only one class of AST Object, the FrameSet, may be represented using a foreign FITS encoding. This should not come as a surprise, because the purpose of storing WCS information in FITS headers is to attach coordinate systems to an associated array of data. Since the FrameSet is the AST Object designed for the same purpose (§13.4), there is a natural correspondence.

   The way in which a FrameSet is translated to and from the foreign encoding also follows from this correspondence. The FrameSet's base Frame identifies the data grid coordinates of the associated FITS data. These are the same as FITS pixel coordinates, in which the first pixel (in 2 dimensions) has coordinates (1,1) at its centre. Similarly, the current Frame of the FrameSet identifies the FITS world coordinate system associated with the data.

---

[25] Available by ftp from fits.cv.nrao.edu /fits/documents/wcs/wcs88.ps.Z

[26] There are many other differences between the previous and the current FITS-WCS encodings. The keywords to describe axis rotation and scaling is used purely as a label to identify the scheme.

[27] Available by ftp from fits.cv.nrao.edu /fits/documents/wcs/aips27.ps.Z

2. You may store a representation of only a single FrameSet in any individual set of FITS header cards (*i.e.* in a single FitsChan) at one time. If you attempt to store more than one, you may over-write the previous one or generate an invalid representation of your WCS information.

   This is mainly a consequence of the use of fixed FITS keywords by foreign encodings and the fact that you cannot, in general, have multiple FITS cards with the same keyword.

3. In general, it will not be possible to store every possible FrameSet that you might construct. Depending on the encoding, only certain FrameSets that conform to particular restrictions can be represented and, even then, some of their information may be lost. See the description of the Encoding attribute in Appendix C for more details of these limitations.

It should be understood that using foreign encodings to read and write information held in AST Objects is essentially a process of converting the data format. As such, it potentially suffers from the same problems faced by all such processes, *i.e.* differences between the AST data model and that of the foreign encoding may cause some information to be lost. Because the AST model is extremely flexible, however, any data loss can largely be eliminated when reading. Instead, this effect manifests itself in the form of the above encoding-dependent restrictions on the kind of AST Objects which may be written.

One of the aims of the AST library, of course, is to insulate you from the details of these foreign encodings and the restrictions they impose. We will see shortly, therefore, how AST provides a mechanism for determining whether your WCS information satisfies the necessary conditions and allows you to make an automatic choice of which encoding to use.

17.3 Identifying Foreign Encodings on Input

Let us now examine the practicalities of extracting WCS information from a set of FITS header cards which have been written by some other software system. We will pretend that our program does not know which encoding has been used for the WCS information and must discover this for itself. In order to have a concrete example, however, we will use the following set of cards. These use the FITS-AIPS encoding and contain a typical mix of other FITS cards which are irrelevant to the WCS information in which we are interested:

```
SIMPLE  =                    T / Written by IDL:  30-Jul-1997 05:35:42.00
BITPIX  =                  -32 / Bits per pixel.
NAXIS   =                    2 / Number of dimensions
NAXIS1  =                  300 / Length of x axis.
NAXIS2  =                  300 / Length of y axis.
CTYPE1  = 'GLON-ZEA'           / X-axis type
CTYPE2  = 'GLAT-ZEA'           / Y-axis type
CRVAL1  =           -149.56866 / Reference pixel value
CRVAL2  =           -19.758201 / Reference pixel value
CRPIX1  =              150.500 / Reference pixel
CRPIX2  =              150.500 / Reference pixel
CDELT1  =             -1.20000 / Degrees/pixel
CDELT2  =              1.20000 / Degrees/pixel
CROTA1  =              0.00000 / Rotation in degrees.
```

```
SURVEY  = 'COBE DIRBE'
BUNITS  = 'MJy/sr  '          /
ORIGIN  = 'CDAC    '          / Cosmology Data Analysis Center
TELESCOP= 'COBE    '          / COsmic Background Explorer satellite
INSTRUME= 'DIRBE   '          / COBE instrument [DIRBE, DMR, FIRAS]
PIXRESOL=                   9 / Quad tree pixel resolution [6, 9]
DATE    = '27/09/94'          / FITS file creation date (dd/mm/yy)
DATE-MAP= '16/09/94'          / Date of original file creation (dd/mm/yy)
COMMENT     COBE specific keywords
DATE-BEG= '08/12/89'          / date of initial data represented (dd/mm/yy)
DATE-END= '25/09/90'          / date of final data represented   (dd/mm/yy)
```

The first step is to create a FitsChan and insert these cards into it. If "cards" is an array of pointers to character strings holding the header cards and "ncards" is the number of cards, this could be done as follows:

```
#include "ast.h"
#define MAXCARD 100
AstFitsChan *fitschan;
char *cards[ MAXCARD ];
int icard, ncard;

...

fitschan = astFitsChan( NULL, NULL, "" );
for ( icard = 0; icard < ncard; icard++ ) astPutFits( fitschan, cards[ icard ], 0 );
```

Note that we have not initialised the Encoding attribute of the FitsChan as we did in §16.3 when we wanted to use the native encoding. This is because we are pretending not to know which encoding to use and want AST to determine this for us. By leaving the Encoding attribute unset, its default value will adjust to whichever encoding AST considers to be most appropriate, according to the FITS header cards present. For details of how this choice is made, see the description of the Encoding attribute in Appendix C.

This approach has the obvious advantages of making our program simpler and more flexible and of freeing us from having to know about the different encodings available. As a bonus, it also means that the program will be able to read any new encodings that AST may support in future, without needing to be changed.

At this point, we could enquire the default value of the Encoding attribute, which indicates which encoding AST intends to use, as follows:

```
const char *encode;

...


encode = astGetC( fitschan, "Encoding" );
```

The result of this enquiry would be the string "FITS-AIPS". Note that we could also have set the FitsChan's Encoding attribute explicitly, such as when creating it:

```
fitschan = astFitsChan( NULL, NULL, "Encoding=FITS-AIPS" );
```

If we tried to read information using this encoding (§17.4), but failed, we could then change the encoding and try again. This would allow our program to take control of how the optimum choice of encoding is arrived at. However, it would also involve using explicit knowledge of the encodings available and this is best avoided if possible.

17.4 Reading Foreign WCS Information from a FITS Header

Having stored a set of FITS header cards in a FitsChan and determined how the WCS information is encoded (§17.3), the next step is to read an AST Object from the FitsChan using astRead. We must also remember to rewind the FitsChan first, if necessary, such as by clearing its Card attribute, which defaults to 1:

```
AstObject *wcsinfo;

...

astClear( fitschan, "Card" );
wcsinfo = astRead( fitschan );
```

If the pointer returned by astRead is not equal to AST__NULL, then an Object has been read successfully. Otherwise, there was either no information to read or the choice of FITS encoding (§17.3) was inappropriate.

At this point you might like to indulge in a little data validation along the lines described in §15.6, for example:

```
if ( !strcmp( astGetC( wcsinfo, "Class" ), "FrameSet" ) ) {
   <the Object is a FrameSet, so use it>
} else {
   <something unexpected was read>
}
```

If a foreign encoding has definitely been used, then the Object will automatically be a FrameSet (§17.2), so this stage can be omitted. However, if the native encoding (§16.1) might have been employed, which is a possibility if you accept the FitsChan's default Encoding value, then any class of Object might have been read and a quick check would be worthwhile.

If you used astShow (§4.4) to examine the FrameSet which results from reading our example FITS header (§17.3), you would find that its base Frame describes the image's pixel coordinate system and that its current Frame is a SkyFrame representing galactic coordinates. These two Frames are inter-related by a Mapping (actually a CmpMap) which incorporates the effects of various rotations, scalings and a "zenithal equal area" sky projection, so that each pixel of the FITS image is mapped on to a corresponding sky position in galactic coordinates.

Because this FrameSet may be used both as a Mapping (§13.6) and as a Frame (§13.8), it may be employed directly to perform many useful operations without any need to decompose it into its component parts. These include:

- Transforming data grid (FITS pixel) coordinates into galactic coordinates and *vice versa* (§13.6).
- Formatting coordinate values (either pixel or galactic coordinates) ready for display to a user (§7.6 and §7.7).
- Enquiring about axis labels (or other axis information—§7.5) which might be used, for example, to label columns of coordinates in a table (§7.4).
- Aligning the image with another image from which a similar FrameSet has been obtained (§14.3).
- Creating a Plot (§21), which can be used to overlay a variety of graphical information (including a coordinate grid—Figure 8) on the displayed image.
- Generating a new FrameSet which reflects any geometrical processing you perform on the associated image data (§14.5). This new FrameSet could then be written out as FITS headers to describe the modified image (§17.7).

If the FrameSet contains other Frames (apart from the base and current Frames), then you would also have access to information about other coordinate systems associated with the image.

17.5 Removing WCS Information from FITS Headers—the Destructive Read

It is instructive at this point to examine the contents of a FitsChan after we have read a FrameSet from it (§17.4). The following would rewind our FitsChan and display its contents:

```
#include <stdio.h>
char card[ 81 ];

...

astClear( fitschan, "Card" );
while ( astFindFits( fitschan, "%f", card, 1 ) ) (void) printf( "%s\n", card );
```

The output, if we started with the example FITS header in §17.3, might look like this:

```
SIMPLE  =                    T /  Written by IDL:  30-Jul-1997 05:35:42.00
BITPIX  =                  -32 /  Bits per pixel.
NAXIS   =                    2 /  Number of dimensions
NAXIS1  =                  300 /  Length of x axis.
NAXIS2  =                  300 /  Length of y axis.
SURVEY  = 'COBE DIRBE'
BUNITS  = 'MJy/sr  '
ORIGIN  = 'CDAC    '           /  Cosmology Data Analysis Center
TELESCOP= 'COBE    '           /  COsmic Background Explorer satellite
INSTRUME= 'DIRBE   '           /  COBE instrument [DIRBE, DMR, FIRAS]
PIXRESOL=                    9 /  Quad tree pixel resolution [6, 9]
DATE    = '27/09/94'           /  FITS file creation date (dd/mm/yy)
DATE-MAP= '16/09/94'           /  Date of original file creation (dd/mm/yy)
COMMENT     COBE specific keywords
DATE-BEG= '08/12/89'           /  date of initial data represented (dd/mm/yy)
DATE-END= '25/09/90'           /  date of final data represented   (dd/mm/yy)
```

Comparing this with the original, you can see that all the FITS cards that represent WCS information have been removed. They have effectively been "sucked out" of the FitsChan by the destructive read that astRead performs and converted into an equivalent FrameSet. AST remembers where they were stored, however, so that if we later write WCS information back into the FitsChan (§17.7) they will, as far as possible, go back into their original locations. This helps to preserve the overall layout of the FITS header.

You can now see why astRead performs destructive reads. It is a mechanism for removing WCS information from a FITS header while insulating you, as a programmer, from the details of the encoding being used. It means you can ensure that all relevant header cards have been removed, giving you a clean slate, without having to know which FITS keywords any particular encoding uses.

Clearing this WCS information out of a FITS header is particularly important when considering how to write new WCS information back after processing (§17.7). If any relevant FITS cards are left over from the input dataset and find their way into the new processed header, they could interfere with the new information being written.[28] The destructive read mechanism ensures that this doesn't happen.

17.6 Propagating WCS Information through Data Processing Steps

One of the purposes of AST is to make it feasible to propagate WCS information through successive stages of data processing, so that it remains consistent with the associated image data. As far as possible, this should happen regardless of the FITS encoding used to store the original WCS information.

If the data processing being performed does not change the relationship between image pixel and world coordinates (whatever these may be), then propagation of the WCS information is straightforward. You can simply copy the FITS header from input to output.

If this relationship changes, however, then the WCS information must be processed alongside the image data and a new FITS header generated to represent it. In this case, the sequence of operations within your program would probably be as follows:

1. Read the image data and associated FITS header from the input dataset, putting the header cards into a FitsChan (§17.3).
2. Read an AST Object, a FrameSet, from the FitsChan (typically using a foreign FITS encoding—§17.4).
3. Process the image data and modify the FrameSet accordingly (*e.g.* §14.5).
4. Write the FrameSet back into the FitsChan (§17.7).
5. Perform any other modification of FITS header cards your program may require.
6. Write the FitsChan contents (*i.e.* processed header cards) and image data to the output dataset.

---

[28] This can happen if a particular keyword is present in the input header but is not used in the output header (whether particular keywords are used can depend on the WCS information being stored). In such a case, the original value would not be over-written by a new output value, so would remain erroneously present.

In stage (2), the original WCS information will be removed from the FitsChan by a destructive read. Later, in stage (4), new WCS information is written to replace it. This is the process which we consider next (§17.7).

17.7 Writing Foreign WCS Information to a FITS Header

Before we can write processed WCS information held in a FrameSet back into a FitsChan in preparation for output, we must select the FITS encoding to use. Unfortunately, we cannot simply depend on the default value of the Encoding attribute, as we did when reading the input information (§17.3), because the destructive action of reading the WCS data (§17.5) will have altered the FitsChan's contents. This, in turn, will have changed the choice of default encoding, probably causing it to revert to NATIVE.

We will return to the question of the optimum choice of encoding below. For now, let's assume that we want to use the same encoding for output as we used for input. Since we enquired what that was before we read the input WCS data from the FitsChan (§17.3), we can now set that value explicitly. We can also set the FitsChan's Card attribute back to 1 at the same time (because the write will fail if the FitsChan is not rewound). astWrite can then be used to write the output WCS information into the FitsChan:

```
int nobj;

...

astSet( fitschan, "Card=1, Encoding=%s", encode );
nobj = astWrite( fitschan, wcsinfo );
```

The value returned by astWrite (assigned to "nobj") indicates how many Objects were written. This will either be 1 or zero. A value of zero is used to indicate that the information could not be encoded in the form you requested. If this happens, nothing will have been written.

If your choice of encoding proves inadequate, the probable reason is that the changes you have made to the FrameSet have caused it to depart from the data model which the encoding assumes. AST knows about the data model used by each encoding and will attempt to simplify the FrameSet you provide so as to fit into that model, thus relieving you of the need to understand the details and limitations of each encoding yourself.[29] When this attempt fails, however, you must consider what alternative encoding to use.

Ideally, you would probably want to try a sequence of alternative encodings, using an approach such as the following:

```
/* 1. */
astSet( fitschan, "Card=1, Encoding=FITS-IRAF" );
if ( !astWrite( fitschan, wcsinfo ) ) {

/* 2. */
   astSetC( fitschan, "Encoding", encode );
   if ( !astWrite( fitschan, wcsinfo ) ) {
```

---

[29] Storing values in the FitsChan for FITS headers NAXIS1, NAXIS2, *etc.* (the grid dimensions in pixels), before invoking astWrite can sometimes help to produce a successful write.

```
/* 3. */
      astSet( fitschan, "Encoding=NATIVE" );
      (void) astWrite( fitschan, wcsinfo );
   }
}
```

That is:

1. Start by trying the FITS-WCS encoding, on the grounds that FITS should provide a universal interchange standard in which all WCS information should be expressed if possible.
2. If that fails, then try the original encoding used for the input WCS information, on the grounds that you are at least not making the information any harder for others to read than it originally was.
3. If that also fails, then you are probably trying to store fairly complex information for which you need the native encoding. Only other AST programs will then be able to read this information, but these are probably the only programs that will be able to do anything sensible with it anyway.

An alternative approach might be to encode the WCS information in several ways, since this gives the maximum chance that other software will be able to read it. This approach is only possible if there is no significant conflict between the FITS keywords used by the different encodings[30]. Adopting this approach would simply require multiple calls to astWrite, rewinding the FitsChan and changing its Encoding value before each one.

Unfortunately, however, there is a drawback to duplicating WCS information in the FITS header in this way, because any program which modifies one version of this information and simply copies the remainder of the header will risk producing two inconsistent sets of information. This could obviously be confusing to subsequent software. Whether you consider this a worthwhile risk probably depends on the use to which you expect your data to be put.

---

[30] In practice, this means you should avoid mixing FITS-IRAF, FITS-WCS, FITS-AIPS, FITS-AIPS++ and FITS-PC encodings since they share many keywords.

18 Storing AST Objects as XML (XmlChan)

XML[31] is fast becoming the standard format for passing structured data around the internet, and much general purpose software has been written for tasks such as the parsing, editing, display and transformation of XML data. The XmlChan class (a specialised form of Channel) provides facilities for storing AST objects externally in the form of XML documents, thus allowing such software to be used.

The primary XML format used by the XmlChan class is a fairly close transliteration of the AST native format produced by the basic Channel class. Currently, there is no DTD or schema defining the structure of data produced in this format by an XmlChan. The following is a native AST representation of a simple 1-D Frame (including comments and with the Full attribute set to zero so that some default attribute values are included as extra comments):

```
 Begin Frame    # Coordinate system description
#   Title = "1-d coordinate system"     # Title of coordinate system
    Naxes = 1   # Number of coordinate axes
    Domain = "SCREEN"   # Coordinate system domain
#   Lbl1 = "Axis 1"     # Label for axis 1
#   Uni1 = "cm"         # Units for axis 1
    Ax1 =       # Axis number 1
       Begin Axis       # Coordinate axis
          Unit = "cm"   # Axis units
       End Axis
 End Frame
```

The corresponding XmlChan output would look like:

```
<Frame xmlns="http://www.starlink.ac.uk/ast/xml/"
       desc="Coordinate system description">
   <_attribute name="Title" quoted="true" value="1-d coordinate system"
               desc="Title of coordinate system" default="true"/>
   <_attribute name="Naxes" value="1" desc="Number of coordinate axes"/>
   <_attribute name="Domain" quoted="true" value="SCREEN"
               desc="Coordinate system domain"/>
   <_attribute name="Lbl1" quoted="true" value="Axis 1"
               desc="Label for axis 1" default="true"/>
   <_attribute name="Uni1" quoted="true" value="cm"
               desc="Units for axis 1" default="true"/>
   <Axis label="Ax1" desc="Coordinate axis">
      <!--Axis number 1-->
      <_attribute name="Unit" quoted="true" value="cm" desc="Axis units"/>
   </Axis>
</Frame>
```

Notes:

1. The AST class name is used as the name for an XML element which contain a description of an AST object.

---

[31] http://www.w3.org/XML/

2. AST attributes are described by XML elements with the name "_attribute". Unfortunately, the word "attribute" is also used by XML to refer to a "name=value" pair within an element start tag. So for instance, the "Title" attribute of the AST Frame object is described within an XML element with name "_attribute" in which the XML attribute "name" has the value "Title", and the XML attribute "value" has the value "1-d coordinate system". The moral is always to be clear clear about the context (AST or XML) in which the word *attribute* is being used!

3. The XML includes comments both as XML attributes with the name "desc", and as separate comment tags.

4. Elements which describe default values are identified by the fact that they have an XML attribute called "default" set to the value "true". These elements are ignored when being read back into an XmlChan.

5. The outer-most XML element of an AST object will set the default namespace to `http://www.starlink.ac.u` which will be inherited by all nested elements.

The XmlChan class changes the default value for the Comment and Full attributes (inherited from the base Channel class) to zero and -1, resulting in terse output by default. With the default values for these attributes, the above XML is reduced to the following:

```
<Frame xmlns="http://www.starlink.ac.uk/ast/xml/">
   <_attribute name="Naxes" value="1"/>
   <_attribute name="Domain" quoted="true" value="SCREEN"/>
   <Axis label="Ax1">
      <_attribute name="Unit" quoted="true" value="cm"/>
   </Axis>
</Frame>
```

The XmlChan class uses the Skip attributes very similarly to the Channel class. If Skip is zero (the default) then an error will be reported if the text supplied by the source function does not begin with an AST Object. If Skip is non-zero, then initial text is skipped over without error until the start of an AST object is found. this allows an AST object to be located within a larger XML document.

18.1 Reading IVOA Space-Time-Coordinates XML (STC-X) Descriptions

The XmlChan class also provides support for reading (but not writing) XML documents which use a restricted subset of an early draft (V1.20) of the IVOA Space-Time-Coordinates XML (STC-X) system. The version of STC-X finally adopted by the IVOA differs in several significant respects from V1.20, and so the STC-X support currently provided by AST is mainly of historical interest. Note, AST also supports the alternative "STC-S" linear string description of the STC model (see §19).

STC-X V1.20 is documented at http://www.ivoa.net/Documents/WD/STC/STC-20050225.html, and the current version is documented at http://www.ivoa.net/Documents/latest/STC-X.html.

When an STC-X document is read using an XmlChan, the read operation produces an AST Object of the Stc class, which is itself a subclass of Region. Specifically, each such Object will be an instance of StcSearchLocation, StcResourceProfile, StcCatalogEntryLocation or StcObsDataLocation. See the description of the XmlChan class and the XmlFormat attribute for further details.

19 Reading and writing STC-S descriptions (StcsChans)

The StcsChan class provides facilities for reading and writing IVOA "STC-S" descriptions. STC-S (see http://www.ivoa.net/Documents/latest/STC-S.html) is a linear string syntax that allows simple specification of the STC metadata describing a region in an astronomical coordinate system. AST supports a subset of the STC-S specification, allowing an STC-S description of a region within an AST-supported astronomical coordinate system to be converted into an equivalent AST Region object, and vice-versa. For further details, see the full description of the StcsChan class in Appendix D.

20 Creating Your Own Private Mappings (IntraMaps)

20.1 The Need for Extensibility

However many Mapping classes are provided by AST, sooner or later you will want to transform coordinates in some way that has not been foreseen. You might want to plot a graph in some novel curvilinear coordinate system (perhaps you already have a WCS system in your software and just want to use AST for its graphical capabilities). Alternatively, you might need to calibrate a complex dataset (like an objective prism plate) where each position must be converted to world coordinates with reference to calibration data under the control of an elaborate algorithm.

In such cases, it is clear that the basic pre-formed components provided by AST for building Mappings are just not enough. What you need is access to a programming language. However, if you write your own software to transform coordinate values, then it must be made available in the form of an AST class (from which you can create Objects) before it can be used in conjunction with other AST facilities.

At this point you might consider writing your own AST class, but this is not recommended. Not only would the internal conventions used by AST take some time to master, but you might also find yourself having to change your software whenever a new version of AST was released. Fortunately, there is a much easier route provided by the IntraMap class.

20.2 The IntraMap Model

To allow you to write your own Mappings, AST provides a special kind of Mapping called an IntraMap. An IntraMap is a sort of "wrapper" for a coordinate transformation function written in C. You write this function yourself and then register it with AST. This, in effect, creates a new class from which you can create Mappings (*i.e.* IntraMaps) which will transform coordinates in whatever way your transformation function specifies.

Because IntraMaps are Mappings, they may be used in the same way as any other Mapping. For instance, they may be combined in series or parallel with other Mappings using a CmpMap (§6), they may be inverted (§5.5), you may enquire about their attributes (§4.5), they may be inserted into FrameSets (§13), *etc.* They do, however, have some important limitations of which you should be aware before we go on to consider how to create them.

20.3 Limitations of IntraMaps

By now, you might be wondering why any other kind of Mapping is required at all. After all, why not simply write your own coordinate transformation functions in C, wrap them up in IntraMaps and do away with all the other Mapping classes in AST?

The reason is not too hard to find. Any transformation function you write is created solely by you, so it is a private extension which does not form a permanent part of AST. If you use it to calibrate some data and then pass that data to someone else, who has only the standard version of AST, then they will not be able to interpret it.

Thus, while an IntraMap is fine for use by you and your collaborators (who we assume have access to the same transformation functions), it does not address the need for universal data exchange like other AST Mappings do. This is where the "Intra" in the class name "IntraMap" comes from, implying private or internal usage.

For this reason, it is unwise to store IntraMaps in datasets, unless they will be used solely for communication between collaborating items of software which share conventions about their use. A private database describing coordinate systems on a graphics device might be an example where IntraMaps would be suitable, because the data would probably never be accessed by anyone else's software. Restricting IntraMap usage to within a single program (*i.e.* never writing it out) is, of course, completely safe.

If, by accident, an IntraMap should happen to escape as part of a dataset, then the unsuspecting recipient is likely to receive an error message when they attempt to read the data. However, AST will associate details of the IntraMap's transformation function and its author (if provided) with the data, so that the recipient can make an intelligent enquiry to obtain the necessary software if this proves essential.

20.4 Writing a Transformation Function

The first stage in creating an IntraMap is to write the coordinate transformation function. This should have a calling interface like the astTranP function provided by AST (*q.v.*). Here is a simple example of a suitable transformation function which transforms coordinates by squaring them:

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

void SqrTran( AstMapping *this, int npoint, int ncoord_in,
              const double *ptr_in[], int forward, int ncoord_out,
              double *ptr_out[] ) {
   int point, coord;
   double x;

/* Forward transformation. */
   if ( forward ) {
      for ( point = 0; point < npoint; point++ ) {
         for ( coord = 0; coord < ncoord_in; coord++ ) {
            x = ptr_in[ coord ][ point ];
            ptr_out[ coord ][ point ] = ( x == AST__BAD ) ? AST__BAD : x * x;
         }
      }

/* Inverse transformation. */
   } else {
      for ( point = 0; point < npoint; point++ ) {
         for ( coord = 0; coord < ncoord_in; coord++ ) {
            x = ptr_in[ coord ][ point ];
            ptr_out[ coord ][ point ] =
               ( x < 0.0 || x == AST__BAD ) ? AST__BAD : sqrt( x );
         }
      }
   }
}
```

As you can see, the function comes in two halves which implement the forward and inverse coordinate transformations. The number of points to be transformed ("npoint") and the numbers of input and output coordinates per point ("ncoord_in" and "ncoord_out"—in this case both are assumed equal) are passed to the function. A pair of loops then accesses all the coordinate values. Note that it is legitimate to omit one or other of the forward/inverse transformations and simply not to implement it, if it will not be required. It is also permissible to require that the numbers of input and output coordinates be fixed (*e.g.* at 2), or to write the function so that it can handle arbitrary dimensionality, as here.

Before using an incoming coordinate, the function must first check that it is not set to the value AST__BAD, which indicates missing data (§5.8). If it is, the same value is also assigned to any affected output coordinates. The value AST__BAD is also generated if any coordinates cannot be transformed. In this example, this can happen with the inverse transformation if negative values are encountered, so that the square root cannot be taken.

There are very few restrictions on what a coordinate transformation function may do. For example, it may freely perform I/O to access any external data needed, it may invoke other AST facilities (but beware of unwanted recursion), *etc.* Typically, you may also want to pass information to it *via* global variables. Remember, however, that whatever facilities the transformation function requires must be available in every program which uses it.

Generally, it is not a good idea to retain context information within a transformation function. That is, it should transform each set of coordinates as a single point and retain no memory of the points it has transformed before. This is in order to conform with the AST model of a Mapping.

If an error occurs within a transformation function, it should use the astSetStatus function (§4.15) to set the AST status to an error value before returning. This will alert AST to the error, causing it to abort the current operation. The error value AST__ITFER is available for this purpose, but other values may also be used (*e.g.* if you wish to distinguish different types of error).

20.5 Registering a Transformation Function

Having written your coordinate transformation function, the next step is to register it with AST. Registration is performed using astIntraReg, as follows:

```
void SqrTran( AstMapping *, int, int, const double *[], int, int, double *[] );

const char *author, *contact, *purpose;

...

purpose = "Square each coordinate value";
author =  "R.F. Warren-Smith & D.S. Berry";
contact = "http://www.starlink.ac.uk/cgi-bin/htxserver/sun211.htx/?xref_SqrTran";

astIntraReg( "SqrTran", 2, 2, SqrTran, 0, purpose, author, contact );
```

Note that you should also provide a function prototype to describe the transformation function (the implementation of the function itself would suffice, of course).

The first argument to astIntraReg is a name by which the transformation function will be known. This will be used when we come to create an IntraMap and is case sensitive. We recommend that you use the actual function name here and make this sufficiently unusual that it is unlikely to clash with any other functions in most people's software.

The next two arguments specify the number of input and output coordinates which the transformation function will handle. These correspond with the Nin and Nout attributes of the IntraMap we will create. Here, we have set them both to 2, which means that we will only be able to create IntraMaps with 2 input and 2 output coordinates (despite the fact that the transformation function can actually handle other dimensionalities). We will see later (§20.8) how to remove this restriction.

The fourth argument should contain a set of flags which describe the transformation function in a little more detail. We will return to this shortly (§20.7 & §20.10). For now, we supply a value of zero.

The remaining arguments are character strings which document the transformation function, mainly for the benefit of anyone who is unfortunate enough to encounter a reference to it in their data which they cannot interpret. As explained above (§20.3), you should try and avoid this, but accidents will happen, so you should always provide strings containing the following:

1. A short description of what the transformation function is for.
2. The name of the author.
3. Contact details, such as an e-mail or WWW address.

The idea is that anyone finding an IntraMap in their data, but lacking the necessary transformation function, should be able to contact the author and make a sensible enquiry in order to obtain it. If you expect many enquiries, you may like to set up a World Wide Web page and use that instead (in the example above, we use the WWW address of the relevant part of this document).

20.6 Creating an IntraMap

Once a transformation function has been registered, creating an IntraMap from it is simple:

```
AstIntraMap *intramap;

...

intramap = astIntraMap( "SqrTran", 2, 2, "" );
```

We simply use the astIntraMap constructor function and pass it the name of the transformation function to use. This name is the same (case sensitive) one that we associated with the function when we registered it using astIntraReg (§20.5).

You can, of course, register any number of transformation functions and select which one to use whenever you create an IntraMap. You can also create any number of independent IntraMaps using each transformation function. In this sense, each transformation function you register effectively creates a new "sub-class" of IntraMap, from which you can create Objects just like any other class. However, an error will occur if you attempt to use a transformation function that has not yet been registered.

The second and third arguments to astIntraMap are the numbers of input and output coordinates. These define the Nin and Nout attributes for the IntraMap that is created and they must match the corresponding numbers given when the transformation function was registered.

The final argument is the usual attribute initialisation string. You may set attribute values for an IntraMap in exactly the same way as for any other Mapping (§4.6, and also see §20.9).

20.7 Restricted Implementations of Transformation Functions

You may not always want to use both the forward and inverse transformations when you create an IntraMap, so it is possible to omit either from the underlying coordinate transformation function. Consider the following, for example:

```
void Poly3Tran( AstMapping *this, int npoint, int ncoord_in,
                const double *ptr_in[], int forward, int ncoord_out,
                double *ptr_out[] ) {
   double x;
   int point;

/* Forward transformation. */
   for ( point = 0; point < npoint; point++ ) {
      x = ptr_in[ 0 ][ point ];
      ptr_out[ 0 ][ point ] = ( x == AST__BAD ) ? AST__BAD :
         6.18 + x * ( 0.12 + x * ( -0.003 + x * 0.0000101 ) );
   }
}
```

This implements a 1-dimensional cubic polynomial transformation. Since this is somewhat awkward to invert, however, we have only implemented the forward transformation. When registering the function, this is indicated via the "flags" argument to astIntraReg, as follows:

```
void Poly3Tran( AstMapping *, int, int, const double *[], int, int, double *[] );

...

astIntraReg( "Poly3Tran", 1, 1, Poly3Tran, AST__NOINV,
             purpose, author, contact );
```

Here, the fifth argument has been set to the flag value AST__NOINV to indicate the lack of an inverse. If the forward transformation were absent, we would use AST__NOFOR instead. Flag values for this argument may be combined using a bitwise OR if necessary.

20.8 Variable Numbers of Coordinates

In our earlier examples, we have used a fixed number of input and output coordinates when registering a coordinate transformation function. It is not necessary to impose this restriction, however, if the transformation function can cope with a variable number of coordinates (as with the example in §20.4). We indicate the acceptability of a variable number when registering the transformation function by supplying the value AST__ANY for the number of input and/or output coordinates, as follows:

```
astIntraReg( "SqrTran", AST__ANY, AST__ANY, SqrTran, 0,
             purpose, author, contact );
```

The result is that an IntraMap may now be created with any number of input and output coordinates. For example:

```
AstIntraMap *intramap1, *intramap2;

...

intramap1 = astIntraMap( "SqrTran", 1, 1, "" );
intramap2 = astIntraMap( "SqrTran", 3, 3, "Invert=1" );
```

It is possible to fix either the number of input or output coordinates (by supplying an explicit number to astIntraReg), but more subtle restrictions on the number of coordinates, such as requiring that Nin and Nout be equal, are not supported. This means that:

```
intramap = astIntraMap( "SqrTran", 1, 2, "" );
```

will be accepted without error, although the transformation function cannot actually handle such a combination sensibly. If this is important, it would be worth adding a check within the transformation function itself, so that the error would be detected when it came to be used.

20.9 Adapting a Transformation Function to Individual IntraMaps

In the examples given so far, our coordinate transformation functions have not made use of the “this” pointer passed to them (which identifies the IntraMap whose transformation we are implementing). In practice, this will often be the case. However, the presence of the “this” pointer allows the transformation function to invoke any other AST function on the IntraMap, and this permits enquiries about its attributes. The transformation function’s behaviour can therefore be modified according to any attribute values which are set. This turns out to be a useful thing to do, so each IntraMap has a special IntraFlag attribute reserved for exactly this purpose.

Consider, for instance, the case where the transformation function has access to several alternative sets of internally-stored data which it may apply to perform its transformation. Rather than implement many different versions of the transformation function, you may switch between them by setting a value for the IntraFlag attribute when you create an instance of an IntraMap, for example:

```
intramap1 = astIntraMap( "MyTran", 2, 2, "IntraFlag=A" );
intramap2 = astIntraMap( "MyTran", 2, 2, "IntraFlag=B" );
```

The transformation function may then enquire the value of the IntraFlag attribute (*e.g.* using astGetC and passing it the "this" pointer) and use whichever dataset is required for that particular IntraMap.

This approach is particularly useful when the number of possible transformations is unbounded or not known in advance, in which case the IntraFlag attribute may be used to hold numerical values encoded as part of a character string (effectively using them as data for the IntraMap). It is also superior to the use of a global switch for communication (*e.g.* setting an index to select the "current" data before using the IntraMap), because it continues to work when several IntraMaps are embedded within a more complex compound Mapping, when you may have no control over the order in which they are used.

20.10 Simplifying IntraMaps

A notable disadvantage of IntraMaps is that they are "black boxes" as far as AST is concerned. This means that they have limited ability to participate in the simplification of compound Mappings performed, *e.g.*, by astSimplify (§6.7), because AST cannot know how they interact with other Mappings. In reality, of course, they will often implement such specialised coordinate transformations that the simplification possibilities will be rather limited anyway.

One important simplification, however, is the ability of a Mapping to cancel with its own inverse to yield a unit Mapping (a UnitMap). This is important because Mappings are frequently used to relate a dataset to some external standard (a celestial coordinate system, for example). When inter-relating two similar datasets calibrated using the same standard, part of the Mapping often cancels, because it is applied first in one direction and then the other, effectively eliminating the reference to the standard. This is often a useful simplification and can lead to greater efficiency.

Many transformations have this property of cancelling with their own inverse, but not necessarily all. Consider the following transformation function, for example:

```
void MaxTran( AstMapping *this, int npoint, int ncoord_in,
              const double *ptr_in[], int forward, int ncoord_out,
              double *ptr_out[] ) {
   double hi, x;
   int coord, point;

/* Forward transformation. */
   if ( forward ) {
      for ( point = 0; point < npoint; point++ ) {
         hi = AST__BAD;
         for ( coord = 0; coord < ncoord_in; coord++ ) {
            x = ptr_in[ coord ][ point ];
            if ( x != AST__BAD ) {
               if ( x > hi || hi == AST__BAD ) hi = x;
            }
         }
         ptr_out[ 0 ][ point ] = hi;
      }

/* Inverse transformation. */
   } else {
      for ( coord = 0; coord < ncoord_out; coord++ ) {
         for ( point = 0; point < npoint; point++ ) {
            ptr_out[ coord ][ point ] = ptr_in[ 0 ][ point ];
         }
      }
   }
}
```

This function takes any number of input coordinates and returns a single output coordinate which is the maximum value of the input coordinates. Its inverse (actually a "pseudo-inverse") sets all the input coordinates to the value of the output coordinate.[32]

If this function is applied in the forward direction and then in the inverse direction, it does **not** in general restore the original coordinate values. However, if applied in the inverse direction and then the forward direction, it does. Hence, replacing the sequence of operations with an equivalent UnitMap is possible in the latter case, but not in the former.

To distinguish these possibilities, two flag values are provided for use with astIntraReg to indicate what simplification (if any) is possible. For example, to register the above transformation function, we might use:

```
void MaxTran( AstMapping *, int, int, const double *[], int, int, double *[] );

...

astIntraReg( "MaxTran", AST__ANY, 1, MaxTran, AST__SIMPIF,
             purpose, author, contact );
```

Here, the flag value AST__SIMPIF supplied for the fifth argument indicates that simplification is possible if the transformation is applied in the inverse direction followed by the forward direction. To indicate the complementary case, the flag AST__SIMPFI would be used instead. If both simplifications are possible (as with the SqrTran function in §20.4), then we would use the bitwise OR of both values.

In practice, some judgement is usually necessary when deciding whether to allow simplification. For example, seen in one light our SqrTran function (§20.4) does not cancel with its own inverse, because squaring a coordinate value and then taking its square root can change the original value, if this was negative. Therefore, replacing this combination with a UnitMap will change the behaviour of a compound Mapping and should not be allowed. Seen in another light, however, where the coordinates being processed are intrinsically all positive, it is a permissible and probably useful simplification.

If such distinctions are ever important in practice, it is simple to register the same transformation function twice with different flag values (use a separate name for each) and then use whichever is appropriate when creating an IntraMap.

20.11 Writing and Reading IntraMaps

It is most important to realise that when you write an IntraMap to a Channel (§15.3), the transformation function which it uses is not stored with it. To do so is impossible, because the

---

[32] Remember that "ptr_in" identifies the original "output" coordinates when applying the inverse transformation and "ptr_out" identifies the original "input" coordinates.

function has been compiled and loaded into memory ready for execution before AST gets to see it. However, AST does store the name associated with the transformation function and various details about the IntraMap itself.

This means that any program attempting to read the IntraMap (§15.4) cannot make use of it unless it also has independent access to the original transformation function. If it does not have access to this function, an error will occur at the point where the IntraMap is read and the associated error message will direct the user to the author of the transformation function for more information.

However, if the necessary transformation function is available, and has been registered before the read operation takes place, then AST is able to re-create the original IntraMap and will do so. Registration of the transformation function must, of course, use the same name (and, in fact, be identical in most particulars) as was used in the original program which wrote the data.

This means that a set of co-operating programs which all have access to the same set of transformation functions and register them in identical fashion (see §20.12 for how this can best be achieved) can freely exchange data that contain IntraMaps. The need to avoid exporting such data to unsuspecting third parties (§20.3) must, however, be re-iterated.

20.12 Managing Transformation Functions in Libraries

If you are developing a large suite of data reduction software, you may have a need to use IntraMaps at various points within it. Very probably this will occur in unrelated modules which are compiled separately and then stored in a library. Since the transformation functions required must be registered before they can be used, this makes it difficult to decide where to perform this registration, especially since any particular data reduction program may use an arbitrary subset of the modules in your library.

To assist with this problem, AST allows you to perform the same registration of a transformation function any number of times, so long as it is performed using an identical invocation of astIntraReg on each occasion (*i.e.* all of its arguments must be identical). This means you do not have to keep track of whether a particular function has already been registered but could, in fact, register it on each occasion immediately before it is required (wherever that may be). In order that all registrations are identical, however, it is recommended that you group them all together into a single function, perhaps as follows:

```
void MyTrans( void ) {

   ...

   astIntraReg( "MaxTran", AST__ANY, 1, MaxTran, AST__SIMPIF,
                purpose, author, contact );

   ...

   astIntraReg( "Poly3Tran", 1, 1, Poly3Tran, AST__NOINV,
                purpose, author, contact );

   ...
```

```
       astIntraReg( "SqrTran", 2, 2, SqrTran, 0,
                    purpose, author, contact );
    }
```

You can then simply invoke this function wherever necessary. It is, in fact, particularly important to register all relevant transformation functions in this way before you attempt to read an Object that might be (or contain) an IntraMap (§20.11). This is because you may not know in advance which of these transformation functions the IntraMap will use, so they must all be available in order to avoid an error.

21 Producing Graphical Output (Plots)

Graphical output from AST is performed though an Object called a Plot, which is a specialised form of FrameSet. A Plot does not represent the graphical content itself, but is a route through which plotting operations, such as drawing lines and curves, are conveyed on to a plotting surface to appear as visible graphics.

21.1 The Plot Model

When a Plot is created, it is initialised by providing a FrameSet whose base Frame (as specified by its Base attribute) is mapped linearly or logarithmically (as specified by the LogPlot attribues) on to a *plotting area.* This is a rectangular region in the graphical coordinate space of the underlying graphics system and becomes the new base Frame of the Plot. In effect, the Plot becomes attached to the plotting surface, in rather the same way that a basic FrameSet might be attached to (say) an image.

The current Frame of the Plot (derived from the current Frame of the FrameSet supplied) is used to represent a *physical coordinate system.* This is the system in which plotting operations are performed by your program. Every plotting operation is then transformed through the Mapping which inter-relates the Plot's current and base Frames in order to appear on the plotting surface.

An example may help here. Suppose we start with a FrameSet whose base Frame describes the pixel coordinates of an image and whose current Frame describes a celestial (equatorial) coordinate system. Let us assume that these two Frames are inter-related by a Mapping within the FrameSet which represents a particular sky projection.

When a Plot is created from this FrameSet, we specify how the pixel coordinates (the base Frame) maps on to the plotting surface. This simply corresponds to telling the Plot where we have previously plotted the image data. If we now use the Plot to plot a line with latitude zero in our physical coordinate system, as given by the current Frame, this line would appear as a curve (the equator) on the plotting surface, correctly registered with the image.

There are a number of plotting functions provided, which all work in a similar way. Plotting operations are transformed through the Mapping which the Plot represents before they appear on the plotting surface.[33] It is possible to draw symbols, lines, axes, entire grids and more in this way.

21.2 Plotting Symbols

The simplest form of plotting is to draw symbols (termed *markers*) at a set of points. This is performed by astMark, which is supplied with a set of physical coordinates at which to place the markers:

```
#include "ast.h"
#define NCOORD 2
#define NMARK 10
double in[ NCOORD ][ NMARK ];
```

---

[33] Like any FrameSet, a Plot can be used as a Mapping. In this case it is the inverse transformation which is used when plotting (*i.e.* that which transforms between the current and base Frames).

```
int type;

...

astMark( plot, NMARK, NCOORD, NMARK, in, type );
```

Here, NMARK specifies how many markers to plot and NCOORD specifies how many coordinates are being supplied for each point.[34] The array "in" supplies the coordinates and the integer "type" specifies which type of marker to plot.

21.3 Plotting Geodesic Curves

There is no Plot routine to draw a straight line, because any straight line in physical coordinates can potentially turn into a curve in graphical coordinates. We therefore start by considering how to draw geodesic curves. These are curves which trace the path of shortest distance between two points in physical coordinates and are the basic drawing element in a Plot.

In many instances, the geodesic will, in fact, be a straight line, but this depends on the Plot's current Frame. If this represents a celestial coordinate system, for instance, it will be a great circle (corresponding with the behaviour of the astDistance function which defines the metric of the physical coordinate space). The geodesic will, of course, be transformed into graphics coordinates before being plotted. A geodesic curve is plotted using astCurve as follows:

```
double start[ NCOORD ], finish[ NCOORD ];

...

astCurve( plot, start, finish );
```

Here, "start" and "finish" are arrays containing the starting and finishing coordinates of the curve. The astOffset and astDistance functions can often be useful for computing these (§7.11).

If you need to draw a series of curves end-to-end (when drawing a contour line, for example), then a more efficient alternative is to use astPolyCurve. This has the same effect as a sequence of invocations of astCurve, but allows you to supply a whole set of points at one time. astPolyCurve then joins them, in sequence, using geodesic curves:

```
#define NPOINT 100
double coords[ NCOORD ][ NPOINT ];

...

astPolyCurve( plot, NPOINT, NCOORD, NPOINT, coords );
```

Here, NPOINT specifies how many points are to be joined and NCOORD specifies how many coordinates are being supplied for each point. The array "coords" supplies the coordinates of the points in the Plot's physical coordinate system.

---

[34] Remember, the physical coordinate space need not necessarily be 2-dimensional, even if the plotting surface is.

21.4 Plotting Curves Parallel to Axes

As there is no Plot function to draw a "straight line", drawing axes and grid lines to represent coordinate systems requires a slightly different approach. The problem is that for some coordinate systems, these grid lines will not be geodesics, so astCurve and astPolyCurve (§21.3) cannot easily be used (you would have to resort to approximating grid lines by many small elements). Lines of constant celestial latitude provide an example of this, with the exception of the equator which is a geodesic.

The astGridLine function allows these curves to be drawn, as follows:

```
int axis;
double length;

...

astGridLine( plot, axis, start, length );
```

Here, "axis" specifies which physical coordinate axis we wish to draw parallel to. The "start" array contains the coordinates of the start of the curve and "length" specifies the distance to draw along the axis in physical coordinate space.

21.5 Plotting Generalized Curves

We have seen how geodesic curves and grid lines can be drawn. The Plot class includes another method, astGenCurve, which allows curves of *any* form to be drawn. The caller supplies a Mapping which maps offset along the curve[35] into the corresponding position in the current Frame of the Plot. astGenCurve, then takes care of Mapping these positions into graphics coordinates. The choice of exactly which positions along the curve are to be used to define the curve is also made by astGenCurve, using an adaptive algorithm which concentrates points around areas where the curve is bending sharply or is discontinuous in graphics coordinates.

The IntraMap class may be of particular use in this context since it allows you to code your own Mappings to do any transformation you choose.

21.6 Clipping

Like many graphics systems, a Plot allows you to *clip* the graphics you produce. This means that plotting is restricted to certain regions of the plotting surface so that anything drawn outside these regions will not appear. All Plots automatically clip at the edges of the plotting area specified when the Plot is created. This means that graphics are ultimately restricted to the rectangular region of plotting space to which you have attached the Plot.

In addition to this, you may also specify lower and upper limits on each axis at which clipping should occur. This permits you to further restrict the plotting region. Moreover, you may attach these clipping limits to *any* of the Frames in the Plot. This allows you to place restrictions on

---

[35] normalized so that the start of the curve is at offset 0.0 and the end of the curve is at offset 1.0 - offset need not be linearly related to distance.

where plotting will take place in either the physical coordinate system, the graphical coordinate system, or in any other coordinate system which is described by a Frame within the Plot.

For example, you could plot using equatorial coordinates and set up clipping limits in galactic coordinates. In general, you could set up arbitrary clipping regions by adding a new Frame to a Plot (in which clipping will be performed) and inter-relating this to the other Frames in a suitable way.

Clipping limits are defined using the astClip function, as follows:

```
#define NAXES 2
int iframe;
double lbnd[ NAXES ], ubnd[ NAXES ];

...
astClip( plot, iframe, lbnd, ubnd);
```

Here, the “iframe” value gives the index of the Frame within the Plot to which clipping is to be applied, while “lbnd” and “ubnd” give the limits on each axis of the selected Frame (NAXES is the number of axes in this Frame).

You can remove clipping by giving a value of AST__NOFRAME for “iframe”.

21.7 Using a Plot as a Mapping

All Plots are also Mappings (just like the FrameSets from which they are derived), so can be used to transform coordinates.

Like FrameSets, the forward transformation of a Plot will convert coordinates between the base and current Frames (*i.e.* between graphical and physical coordinates). This would be useful if you were (say) reading a cursor position in graphical coordinates and needed to convert this into physical coordinates for display.

Conversely, a Plot’s inverse transformation converts between its current and base Frames (*i.e.* from physical coordinates to graphical coordinates). This transformation is applied automatically whenever plotting operations are carried out by AST functions. It may also be useful to apply it directly, however, if you wish to perform additional plotting operations (*e.g.* those provided by the native graphics system) at positions specified in physical coordinates.

There is, however, one important difference between using a FrameSet and a Plot to transform coordinates, and this is that clipping may be applied by a Plot (if it has been enabled using astClip—§21.6). Any point which lies within the clipped region of a Plot will, when transformed, yield coordinates with the value AST__BAD. If you wish to avoid this clipping, you should extract the relevant Mapping from the Plot (using astGetMapping) and use this, instead of the Plot, to transform the coordinates.

21.8 Using a Plot as a Frame

Every Plot is also a Frame, so can be used to obtain the values of Frame attributes such as a Title, axis Labels, axis Units, *etc.,* which are typically used when displaying data and/or coordinates. These attributes are, as for any FrameSet, derived from the current Frame of the

Plot (§13.8). They are also used automatically when using the Plot to plot coordinate axes and coordinate grids (*e.g.* for labelling them—§21.12).

Because the current Frame of a Plot represents physical coordinates, any Frame operation applied to the Plot will effectively be working in this coordinate system. For example, the astDistance and astOffset functions will compute distances and offsets in physical coordinate space, while astFormat and astNorm will format physical coordinates in an appropriate way for display.

21.9 Regions of Valid Physical Coordinates

When points in physical coordinate space are transformed by a Plot into graphics coordinates for plotting, they may not always yield valid coordinates, irrespective of any clipping being applied (§21.6). To indicate this, the resulting coordinate values will be set to the value AST__BAD (§5.8).

There are a number of reasons why this may occur, but typically it will be because physical coordinates only map on to a subset of the graphics coordinate space. This situation is commonly encountered with all-sky projections where, typically, the celestial sphere appears, when plotted, as a distorted shape (*e.g.* an ellipse) which does not entirely fill the graphics space. In some cases, there may even be multiple regions of valid and invalid physical coordinates.

When plotting is performed *via* a Plot, graphical output will only appear in the regions of valid physical coordinates. Nothing will appear where invalid coordinates occur. Such output is effectively clipped. If you wish to plot in these areas, you must change coordinate system and use, say, graphical coordinates to address the plotting surface directly.

21.10 Plotting Borders

The astBorder function is provided to draw a (line) border around your graphical output. With most graphics systems, this would simply be a rectangular box around the plotting area. With a Plot, however, this boundary follows the edge of each region containing valid, unclipped physical coordinates (§21.9).

This means, for example, that if you were plotting an all-sky projection, this boundary would outline the perimeter of the celestial sphere when projected on to your plotting surface. Of course, if there is no clipping and all physical coordinates are valid, then you will get the traditional rectangular box. astBorder requires only a pointer to the Plot:

```
int holes;

...

holes = astBorder( plot );
```

It returns a boolean (integer) value to indicate if any invalid or clipped physical coordinates were found within the plotting area. If they were, it will draw around the valid unclipped regions and return a value of one. Otherwise, it will draw a simple rectangular border and return zero.

21.11 Plotting Text

Using a Plot to draw text involves supplying a string of text to be displayed and a position in physical coordinates where the text is to appear. The position is transformed into graphical coordinates to determine where the text should appear on the plotting surface. You must also provide a 2-element "up" vector which gives the upward direction of the text in graphical coordinates. This allows text to be drawn at any angle.

Plotting is performed by astText, for example:

```
char text[ 21 ];
double pos[ NCOORD ];
float up[ 2 ] = { 0.0f, 1.0f };

...

astText( plot, text, pos, up, "TL" );
```

Here, "text" contains the string to be drawn, "pos" is an array of physical coordinates and "up" specifies the upward vector. In this case, the text will be drawn horizontally. The final argument specifies the text justification, here indicating that the top left corner of the text should appear at the position given.

Further control over the appearance of the text is possible by setting values for various Plot attributes, for example Colour, Font and Size. Sub-strings within the displayed text can be given different appearances, or turned into super-scripts or sub-scripts, by the inclusion of escape sequences (see section §21.13) within the supplied text string.

21.12 Plotting a Grid

The most comprehensive plotting function available is astGrid, which can be used to draw labelled coordinate axes and, optionally, to overlay coordinate grids on the plotting area (Figure 8). The routine is straightforward to use, simply requiring a pointer to the Plot:

```
astGrid( plot );
```

It will draw both linear and curvilinear axes and grids, as required by the particular Plot. The appearance of the output can be modified in a wide variety of ways by setting various Plot attributes. The Label attributes of the current Frame are displayed as the axis labels in the grid, and the Title attribute as the plot title. Sub-strings within these strings can be given different appearances, or turned into super-scripts or sub-scripts, by the inclusion of escape sequences (see section §21.13) within the Label attributes.

21.13 Controlling the Appearance of Sub-strings

Normally, each string of characters displayed using a Plot will be plotted so that all characters in the string have the same font size, colour, *etc.,* specified by the appropriate attributes of the Plot. However, it is possible to include *escape sequences* within the text to modify the appearance of sub-strings. Escape sequences can be used to change, colour, font, size, width, to introduce extra horizontal space between characters, and to change the base line of characters (thus allowing super-scripts and sub-scripts to be created). See the entry for the Escape attribute in Appendix C for details.

As an example, if the character string "`10\%^50`%s70+0.5+" is plotted, it will be displayed as "$10^{0.5}$" - that is, with a super-scripted exponent. The exponent text will be 70% of the size of normal text (as determined by the Size attribute), and its baseline will be raised by 50% of the height of a normal character.

Such escape sequences can be used in the strings assigned to textual attributes of the Plot (such as the axis Labels), and may also be included in strings plotted using astText.

The Format attribute for the SkyAxis class includes the "g" option which will cause escape sequences to be included when formatting celestial positions so that super-script characters are used as delimiters for the various fields (a super-script "h" for hours, "m" for minutes, *etc*).

Note, the facility for interpreting escape sequences is only available if the graphics wrapper functions which provide the interface to the underlying graphics system support all the functions included in the `grf.h` file as of AST V3.2. Older grf interfaces may need to be extended by the addition of new functions before escape sequences can be interpretted.

21.14 Producing Logarithmic Axes

In certain situations you may wish for one or both of the plotted axes to be displayed logarithmically rather than linearly. For instance, you may wish to do this when using a Plot to represent a spectrum of, say, flux against frequency. In this case, you can cause the frequency axis to be drawn logarithmically simply by setting the boolean LogPlot attribute for the frequency axis to a non-zero value. This causes several things to happen:

1. The Mapping between the base Frame of the Plot (which represents the underlying graphics world coordinate system) and the base Frame of the FrameSet supplied when the Plot was created, is modified. By default, this mapping is linear on both axes, but setting LogPlot non-zero for an axis causes the Mapping to be modified so that it is logarithmic on the specified axis. This is only possible if the displayed section of the axis does not include the value zero (otherwise the attempt to set a new value for LogPlot is ignored,and it retains its default value of zero).

2. The major tick marks drawn as part of the annotated coordinate grid are spaced logarithmically rather than linearly. That is, major axis values are chosen so that there is a constant ratio between adjacent tick mark values. This ratio is constrained to be a power of ten. The minor tick marks are drawn at linearly distributed points between the adjoining major tick values. Thus if a pair of adjacent major tick values are drawn at axis values 10.0 and 100.0, minor ticks will be placed at 20.0, 30.0, 40.0, 50.0, 60.0, 70.0, 80.0 and 90.0 (note only 8 minor tick marks are drawn).

3. If possible, numerical axis labels are shown as powers of ten. This depends on the facilities implemented by the graphics wrapper functions (see the next section). Extra functions were introduced to this set of wrapper functions at AST V3.2 which enable super-scripts and sub-scripts to be produced. Some older wrappers may not yet have implemented

these functiosn and this will result in axis labels being drawn in usual scientific or decimal notation.

Whilst the LogPlot attribute can be used to control all three of the above facilities, it is possible to control them individually as well. The LogTicks and LogLabel attributes control the behaviour specified in items 2 and 3 above, but the default values for these attributes depend on the setting of the LogPlot attribute. This means that setting LogPlot non-zero will swicth all three facilites on, so long as zero values have not been assigned explicitly to LogTicks or LogLabel.

21.15 Choosing a Graphics Package

The Plot class itself does not include any code for actually drawing on a graphics device. Instead, it requires a set of functions to be provided which it uses to draw the required graphics. These include functions to draw a straight line, draw a text string, *etc.* You may choose to provide functions from your favorite graphics package, or you can even write your own! To accomodate variations in the calling interfaces of different graphics packages, AST defines a standard interface for these routines. If this interface differs from the interface provided by your graphics package (which in general it will), then you must write a set of *wrapper functions*, which provide the interface expected by AST but which then call functions from your graphics package to provide the required functionality. AST comes with wrapper functions suitable for the PGPLOT graphics package (see SUN/15).

There are two ways of indicating which wrapper functions are to be used by the Plot class:

1. A file containing C functions with pre-defined names can be written and linked with the application using options of the ast_link command. (see §3.3 and Appendix E). AST is distributed with such a file (called `grf_pgplot.c`) which calls PGPLOT functions to implement the required functionality. This file can be used as a template for writing your own.
2. The astGrfSet method of the Plot class can be used to "register" wrapper functions at run-time. This allows an application to switch between graphics systems if required. Graphics functions registered in this way do not need to have the pre-defined names used in the link-time method described above.

For details of the interfaces of the wrapper routines, see either the `grf_pgplot.c` file included in the AST source distribution, or the reference documentation for the astGrfSet method.

22 Compiling and Linking Software that Uses AST

A small number of UNIX commands are provided by AST to assist with the process of building software. A description of these can be found in Appendix E and their use is discussed here. Note that in order to access these commands, the appropriate directory (normally "/star/bin") should be on your PATH.[36]

22.1 Accessing the "ast.h" Header File

The "ast.h" header file defines the external interface to the AST library, including all constants, function prototypes, macros, *etc.*. This file should be located using the usual compiler options for finding C include files, for instance:

```
cc prog.c -I/star/include -o prog
```

This is preferable to specifying the file's absolute name within your software.

22.2 Linking with AST Facilities

C programs which use AST facilities may be linked by including execution of the command "ast_link" on the compiler command line. Thus, to compile and link a program called "prog", the following might be used:

```
cc prog.c -L/star/lib `ast_link` -o prog
```

Note the use of backward quote characters, which cause the "ast_link" command to be executed and its result substituted into the compiler command. An alternative is to save the output from "ast_link" in (say) a shell variable and use this instead. You may find this a little faster if you are building software repeatedly during development.

Programs which use AST can also be linked in a number of other ways, depending on the facilities they require. In the example above, we have used the default method which assumes that the program will not be generating graphical output, so that no graphics libraries need be linked. If you need other facilities, then various switches can be applied to the "ast_link" command in order to control the linking process.

For example, if you were producing graphical output using the PGPLOT graphics package, you could link with the AST/PGPLOT interface by using the "−pgplot" switch with "ast_link", as follows:[37]

```
cc prog.c -L/star/lib `ast_link -pgplot` -o prog
```

See the "ast_link" command description in Appendix E for details of the options available.

---

[36] If you have not installed AST in the usual location, then substitute the appropriate directory in place of "/star" wherever it occurs.

[37] Use the "−pgp" option instead if you wish to use the Starlink version of PGPLOT which uses GKS to generate its output.

22.3 Building ADAM Applications that Use AST

Users of Starlink's ADAM programming environment (SG/4) on UNIX should use the "alink" command (SUN/144) to compile and link applications and can access the AST library by including execution of the command "ast_link_adam" on the command line, as follows:

```
alink adamprog.c `ast_link_adam`
```

Note the use of backward quote characters.

By default, AST error messages produced by applications built in this way will be delivered *via* the Starlink EMS Error Message Service (SSN/4) so that error handling by AST is consistent with the *inherited status* error handling normally used in Starlink software.

Switches may be given to the "ast_link_adam" command (in a similar way to "ast_link"—§22.2) in order to link with additional AST-related facilities, such as a graphics interface. See the "ast_link_adam" command description in Appendix E for details of the options available.

A The AST Class Hierarchy

The following table shows the hierarchy of classes in the AST library. For a description of each class, you should consult Appendix D.

```
Object             - Base class for all AST Objects
   Axis            - Store axis information
      SkyAxis      - Store celestial axis information
   Channel         - Basic (textual) I/O channel
      FitsChan     - I/O Channel using FITS header cards
      XmlChan      - I/O Channel using XML
      StcsChan     - I/O Channel using IVOA STC-S descriptions
   KeyMap          - Store a set of key/value pairs
      Table        - Store a 2-dimensional table of values
   Mapping         - Inter-relate two coordinate systems
      CmpMap       - Compound Mapping
      DssMap       - Map points using Digitised Sky Survey plate solution
      Frame        - Coordinate system description
         CmpFrame  - Compound Frame
            SpecFluxFrame - Observed value versus spectral position
         FluxFrame - Observed value at a given fixed spectral position
         FrameSet  - Set of inter-related coordinate systems
            Plot   - Provide facilities for 2D graphical output
               Plot3D - Provide facilities for 3D graphical output
         Region    - Specify areas within a coordinate system
            Box    - A box region with sides parallel to the axes of a Frame
            Circle - A circular or spherical region within a Frame
            CmpRegion  - A combination of two regions within a single Frame
            Ellipse    - An elliptical region within a 2-dimensional Frame
            Interval   - Intervals on one or more axes of a Frame.
            NullRegion - A boundless region within a Frame
            PointList  - A collection of points in a Frame
            Polygon    - A polygonal region within a 2-dimensional Frame
            Prism  - An extrusion of a Region into orthogonal dimensions
            Stc    - Represents an generic instance of an IVOA STC-X description
               StcResourceProfile - Represents an an IVOA STC-X ResourceProfile
               StcSearchLocation  - Represents an an IVOA STC-X SearchLocation
               StcCatalogEntryLocation - Represents an an IVOA STC-X CatalogEntryLocation
               StcObsDataLocation - Represents an an IVOA STC-X ObsDataLocation
         SkyFrame  - Celestial coordinate system description
         SpecFrame - Spectral coordinate system description
            DSBSpecFrame - Dual sideband spectral coordinate system description
         TimeFrame - Time coordinate system description
      GrismMap     - Models the spectral dispersion produced by a grism
      IntraMap     - Map points using a private transformation function
      LutMap       - Transform 1-dimensional coordinates using a lookup table
      MathMap      - Transform coordinates using mathematical expressions
      MatrixMap    - Map positions by multiplying them by a matrix
      NormMap      - Normalise coordinates using a supplied Frame
      PcdMap       - Apply 2-dimensional pincushion/barrel distortion
      PermMap      - Coordinate permutation Mapping
      PolyMap      - General N-dimensional polynomial Mapping
      RateMap      - Calculates an element of a Mapping's Jacobian matrix
```

SelectorMap - Locates positions within a set of Regions
ShiftMap - Shifts each axis by a constant amount
SlaMap - Sequence of celestial coordinate conversions
SpecMap - Sequence of spectral coordinate conversions
SphMap - Map 3-d Cartesian to 2-d spherical coordinates
SwitchMap - Encapuslates a set of alternate Mappings
TimeMap - Sequence of time coordinate conversions
TranMap - Combine fwd. and inv. transformations from two Mappings
UnitMap - Unit (null) Mapping
WcsMap - Implement a FITS-WCS sky projection
WinMap - Match windows by scaling and shifting each axis
ZoomMap - Zoom coordinates about the origin

B AST Function Descriptions

---

astSet Set attribute values for an Object astSet

Description: This function assigns a set of attribute values to an Object, over-riding any previous values. The attributes and their new values are specified via a character string, which should contain a comma-separated list of the form:

"attribute_1 = value_1, attribute_2 = value_2, ... "

where "attribute_n" specifies an attribute name, and the value to the right of each "=" sign should be a suitable textual representation of the value to be assigned. This value will be interpreted according to the attribute's data type.

The string supplied may also contain "printf"-style format specifiers, identified by "%" signs in the usual way. If present, these will be substituted by values supplied as additional optional arguments (using the normal "printf" rules) before the string is used.

Synopsis: `void astSet( AstObject *this, const char *settings, ... )`

Parameters:

this

Pointer to the Object.

settings

Pointer to a null-terminated character string containing a comma-separated list of attribute settings in the form described above.

...

Optional additional arguments which supply values to be substituted for any "printf"-style format specifiers that appear in the "settings" string.

Class Applicability:

Object

This function applies to all Objects.

Examples:

`astSet( map, "Report = 1, Zoom = 25.0" );`

Sets the Report attribute for Object "map" to the value 1 and the Zoom attribute to 25.0.

`astSet( frame, "Label( %d ) =Offset along axis %d", axis, axis );`

Sets the Label(axis) attribute for Object "frame" to a suitable string, where the axis number is obtained from "axis", a variable of type int.

`astSet( frame, "Title =%s", mystring );`

Sets the Title attribute for Object "frame" to the contents of the string "mystring".

Notes:

- Attribute names are not case sensitive and may be surrounded by white space.
- White space may also surround attribute values, where it will generally be ignored (except for string-valued attributes where it is significant and forms part of the value to be assigned).
- To include a literal comma in the value assigned to an attribute, the whole attribute value should be enclosed in quotation markes. Alternatively, you can use "%s" format and supply the value as a separate additional argument to astSet (or use the astSetC function instead).
- The same procedure may be adopted if "%" signs are to be included and are not to be interpreted as format specifiers (alternatively, the "printf" convention of writing "%%" may be used).
- An error will result if an attempt is made to set a value for a read-only attribute.

---

astAddColumn Add a new column definition to a table astAddColumn

Description: Adds the definition of a new column to the supplied table. Initially, the column is empty. Values may be added subsequently using the methods of the KeyMap class.

Synopsis: `void astAddColumn( AstTable *this, const char *name, int type, int ndim, int *dims, const char *unit )`

Parameters:

this
Pointer to the Table.

name
The column name. Trailing spaces are ignored (all other spaces are significant). The supplied string is converted to upper case.

type
The data type associated with the column. See "Applicability:" below.

ndim
The number of dimensions spanned by the values stored in a single cell of the column. Zero if the column holds scalar values.

dims
An array holding the the lengths of each of the axes spanned by the values stored in a single cell of the column. Ignored if the column holds scalara values.

unit
A string specifying the units of the column. Supply a blank string if the column is unitless.

Class Applicability:

Table
Tables can hold columns with any of the following data types - AST__INTTYPE (for integer), AST__SINTTYPE (for short int), AST__BYTETYPE (for unsigned bytes - i.e. unsigned chars), AST__DOUBLETYPE (for double precision floating point), AST__FLOATTYPE (for single precision floating point), AST__STRINGTYPE (for character string), AST__OBJECTTYPE (for AST Object pointer), AST__POINTERTYPE (for arbitrary C pointer) or AST__UNDEFTYPE (for undefined values created by astMapPutU).

FitsTable
FitsTables can hold columns with any of the following data types - AST__INTTYPE (for integer), AST__SINTTYPE (for short int), AST__BYTETYPE (for unsigned bytes - i.e. unsigned chars), AST__DOUBLETYPE (for double precision floating point), AST__FLOATTYPE (for single precision floating point), AST__STRINGTYPE (for character string).

Notes:

- This function returns without action if a column already exists in the Table with the supplied name and properties. However an error is reported if any of the properties differ.

---

astAddFrame Add a Frame to a FrameSet to define a new coordinate system astAddFrame

Description: This function adds a new Frame and an associated Mapping to a FrameSet so as to define a new coordinate system, derived from one which already exists within the FrameSet. The new Frame then becomes the FrameSet's current Frame.

This function may also be used to merge two FrameSets, or to append extra axes to every Frame in a FrameSet.

Synopsis: `void astAddFrame( AstFrameSet *this, int iframe, AstMapping *map, AstFrame *frame )`

Parameters:

this

Pointer to the FrameSet.

iframe

The index of the Frame within the FrameSet which describes the coordinate system upon which the new one is to be based. This value should lie in the range from 1 to the number of Frames already in the FrameSet (as given by its Nframe attribute). As a special case, AST__ALLFRAMES may be supplied, in which case the axes defined by the supplied Frame are appended to every Frame in the FrameSet (see the Notes section for details).

map

Pointer to a Mapping which describes how to convert coordinates from the old coordinate system (described by the Frame with index "iframe") into coordinates in the new system. The Mapping's forward transformation should perform this conversion, and its inverse transformation should convert in the opposite direction. The supplied Mapping is ignored if parameter "iframe"is equal to AST__ALLFRAMES.

frame

Pointer to a Frame that describes the new coordinate system. Any class of Frame may be supplied (including Regions and FrameSets).

This function may also be used to merge two FrameSets by supplying a pointer to a second FrameSet for this parameter (see the Notes section for details).

Notes:

- A value of AST__BASE or AST__CURRENT may be given for the "iframe" parameter to specify the base Frame or the current Frame respectively.
- This function sets the value of the Current attribute for the FrameSet so that the new Frame subsequently becomes the current Frame.
- The number of input coordinate values accepted by the supplied Mapping (its Nin attribute) must match the number of axes in the Frame identified by the "iframe" parameter. Similarly, the number of output coordinate values generated by this Mapping (its Nout attribute) must match the number of axes in the new Frame.
- As a special case, if a pointer to a FrameSet is given for the "frame" parameter, this is treated as a request to merge a pair of FrameSets. This is done by appending all the new Frames (in the "frame" FrameSet) to the original FrameSet, while preserving their order and retaining all the inter-relationships (i.e. Mappings) between them. The two sets of Frames are inter-related within the merged FrameSet by using the Mapping supplied. This should convert between the Frame identified by the "iframe" parameter (in the original FrameSet) and the current Frame of the "frame" FrameSet. This latter Frame becomes the current Frame in the merged FrameSet.
- As another special case, if a value of AST__ALLFRAMES is supplied for parameter "iframe", then the supplied Mapping is ignored, and the axes defined by the supplied Frame are appended to each Frame in the FrameSet. In detail, each Frame in the FrameSet is replaced by a CmpFrame containing the original Frame and the Frame specified by parameter "frame". In addition, each Mapping in the FrameSet is replaced by a CmpMap containing the original Mapping and a UnitMap in parallel. The Nin and Nout attributes of the UnitMap are set equal to the number of axes in the supplied Frame. Each new CmpMap is simplified using astSimplify before being stored in the FrameSet.

---

astAddParameter — Add a new global parameter definition to a table — astAddParameter

Description: Adds the definition of a new global parameter to the supplied table. Note, this does not store a value for the parameter. To get or set the parameter value, the methods of the paremt KeyMap class should be used, using the name of the parameter as the key.

Synopsis: `void astAddParameter( AstTable *this, const char *name )`

Parameters:

this

Pointer to the Table.

name

The parameter name. Trailing spaces are ignored (all other spaces are significant). The supplied string is converted to upper case.

Notes:

- Unlike columns, the definition of a parameter does not specify its type, size or dimensionality.

---

astAddVariant — Store a new variant Mapping for the current Frame in a FrameSet — astAddVariant

Description: This function allows a new variant Mapping to be stored with the current Frame in a FrameSet. See the "Variant" attribute for more details. It can also be used to rename the currently selected variant Mapping.

Synopsis: `void astAddVariant( AstFrameSet *this, AstMapping *map, const char *name, int *status )`

Parameters:

this

Pointer to the FrameSet.

map

Pointer to a Mapping which describes how to convert coordinates from the current Frame to the new variant of the current Frame. If NULL is supplied, then the name associated with the currently selected variant of the current Frame is set to the value supplied for "name", but no new variant is added.

name

The name to associate with the new variant Mapping (or the currently selected variant Mapping if "map" is NULL).

Notes:

- The newly added Variant becomes the current variant on exit (this is equivalent to setting the Variant attribute to the value supplied for "name).
- An error is reported if a variant with the supplied name already exists in the current Frame.
- An error is reported if the current Frame is a mirror for the variant Mappings in another Frame. This is only the case if the astMirrorVariants function has been called to make the current Frame act as a mirror.

astAngle Calculate the angle subtended by two points at a third point astAngle

Description: This function finds the angle at point B between the line joining points A and B, and the line joining points C and B. These lines will in fact be geodesic curves appropriate to the Frame in use. For instance, in SkyFrame, they will be great circles.

Synopsis: `double astAngle( AstFrame *this, const double a[], const double b[], const double c[] )`

Parameters:

this

Pointer to the Frame.

a

An array of double, with one element for each Frame axis (Naxes attribute) containing the coordinates of the first point.

b

An array of double, with one element for each Frame axis (Naxes attribute) containing the coordinates of the second point.

c

An array of double, with one element for each Frame axis (Naxes attribute) containing the coordinates of the third point.

Returned Value:

astAngle

The angle in radians, from the line AB to the line CB. If the Frame is 2-dimensional, it will be in the range $\pm \pi$, and positive rotation is in the same sense as rotation from the positive direction of axis 2 to the positive direction of axis 1. If the Frame has more than 2 axes, a positive value will always be returned in the range zero to $\pi$.

Notes:

- A value of AST__BAD will also be returned if points A and B are co-incident, or if points B and C are co-incident.
- A value of AST__BAD will also be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

astAnnul Annul a pointer to an Object astAnnul

Description: This function annuls a pointer to an Object so that it is no longer recognised as a valid pointer by the AST library. Any resources associated with the pointer are released and made available for re-use.

This function also decrements the Object's RefCount attribute by one. If this attribute reaches zero (which happens when the last pointer to the Object is annulled), then the Object is deleted.

Synopsis: `AstObject *astAnnul( AstObject *this )`

Parameters:

this

The Object pointer to be annulled.

Class Applicability:

Object

This function applies to all Objects.

Returned Value:

astAnnul()

A null Object pointer (AST__NULL) is always returned.

Notes:

- This function will attempt to annul the pointer even if the Object is not currently locked by the calling thread (see astLock).
- This function attempts to execute even if the AST error status is set on entry, although no further error report will be made if it subsequently fails under these circumstances. In particular, it will fail if the pointer suppled is not valid, but this will only be reported if the error status is clear on entry.

---

astAxAngle Returns the angle from an axis, to a line through two points astAxAngle

Description: This function finds the angle, as seen from point A, between the positive direction of a specified axis, and the geodesic curve joining point A to point B.

Synopsis: `double astAxAngle( AstFrame *this, const double a[], const double b[], int axis )`

Parameters:

this

Pointer to the Frame.

a

An array of double, with one element for each Frame axis (Naxes attribute) containing the coordinates of the first point.

b

An array of double, with one element for each Frame axis (Naxes attribute) containing the coordinates of the second point.

axis

The number of the Frame axis from which the angle is to be measured (axis numbering starts at 1 for the first axis).

Returned Value:

astAxAngle

The angle in radians, from the positive direction of the specified axis, to the line AB. If the Frame is 2-dimensional, it will be in the range [-PI/2,+PI/2], and positive rotation is in the same sense as rotation from the positive direction of axis 2 to the positive direction of axis 1. If the Frame has more than 2 axes, a positive value will always be returned in the range zero to PI.

Notes:

- The geodesic curve used by this function is the path of shortest distance between two points, as defined by the astDistance function.
- This function will return "bad" coordinate values (AST__BAD) if any of the input coordinates has this value, or if the require position angle is undefined.

---

astAxDistance Find the distance between two axis values astAxDistance

Description: This function returns a signed value representing the axis increment from axis value v1 to axis value v2.

For a simple Frame, this is a trivial operation returning the difference between the two axis values. But for other derived classes of Frame (such as a SkyFrame) this is not the case.

Synopsis: `double astAxDistance( AstFrame *this, int axis, double v1, double v2 )`

Parameters:

this
Pointer to the Frame.

axis
The index of the axis to which the supplied values refer. The first axis has index 1.

v1
The first axis value.

v2
The second axis value.

Returned Value:

astAxDistance
The distance from the first to the second axis value.

Notes:

- This function will return a "bad" result value (AST__BAD) if any of the input values has this value.
- A "bad" value will also be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

---

astAxOffset Add an increment onto a supplied axis value astAxOffset

Description: This function returns an axis value formed by adding a signed axis increment onto a supplied axis value.

For a simple Frame, this is a trivial operation returning the sum of the two supplied values. But for other derived classes of Frame (such as a SkyFrame) this is not the case.

Synopsis: `double astAxOffset( AstFrame *this, int axis, double v1, double dist )`

Parameters:

this
Pointer to the Frame.

axis
The index of the axis to which the supplied values refer. The first axis has index 1.

v1
The original axis value.

dist
The axis increment to add to the original axis value.

Returned Value:

astAxOffset
The incremented axis value.

Notes:

- This function will return a "bad" result value (AST__BAD) if any of the input values has this value.
- A "bad" value will also be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

---

astBBuf Begin a new graphical buffering context astBBuf

Description: This function starts a new graphics buffering context. A matching call to the function astEBuf should be used to end the context.

Synopsis: `void astBBuf( AstPlot *this )`

Parameters:

this
Pointer to the Plot.

Notes:

- The nature of the buffering is determined by the underlying graphics system (as defined by the current grf module). Each call to this function to this function simply invokes the astGBBuf function in the grf module.

---

astBegin Begin a new AST context astBegin

Description: This macro invokes a function to begin a new AST context. Any Object pointers created within this context will be annulled when it is later ended using astEnd (just as if astAnnul had been invoked), unless they have first been exported using astExport or rendered exempt using astExempt. If annulling a pointer causes an Object's RefCount attribute to fall to zero (which happens when the last pointer to it is annulled), then the Object will be deleted.

Synopsis: `void astBegin`

Class Applicability:

Object
This macro applies to all Objects.

Notes:

- astBegin attempts to execute even if the AST error status is set on entry.
- Contexts delimited by astBegin and astEnd may be nested to any depth.

astBorder Draw a border around valid regions of a Plot astBorder

Description: This function draws a (line) border around regions of the plotting area of a Plot which correspond to valid, unclipped physical coordinates. For example, when plotting using an all-sky map projection, this function could be used to draw the boundary of the celestial sphere when it is projected on to the plotting surface.

If the entire plotting area contains valid, unclipped physical coordinates, then the boundary will just be a rectangular box around the edges of the plotting area.

If the Plot is a Plot3D, this method is applied individually to each of the three 2D Plots encapsulated within the Plot3D (each of these Plots corresponds to a single 2D plane in the 3D graphics system). In addition, if the entire plotting volume has valid coordinates in the 3D current Frame of the Plot3D, then additional lines are drawn along the edges of the 3D plotting volume so that the entire plotting volume is enclosed within a cuboid grid.

Synopsis: `int astBorder( AstPlot *this )`

Parameters:

this

Pointer to the Plot.

Returned Value:

astBorder()

Zero is returned if the plotting space is completely filled by valid, unclipped physical coordinates (so that only a rectangular box was drawn around the edge). Otherwise, one is returned.

Notes:

- A value of zero will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.
- An error results if either the current Frame or the base Frame of the Plot is not 2-dimensional or (for a Plot3D) 3-dimensional.
- An error also results if the transformation between the base and current Frames of the Plot is not defined (i.e. the Plot's TranForward attribute is zero).

astBoundingBox Return a bounding box for previously drawn graphics astBoundingBox

Description: This function returns the bounds of a box which just encompasess the graphics produced by the previous call to any of the Plot methods which produce graphical output. If no such previous call has yet been made, or if the call failed for any reason, then the bounding box returned by this function is undefined.

Synopsis: `void astBoundingBox( AstPlot *this, float lbnd[2], float ubnd[2] )`

Parameters:

this

Pointer to the Plot.

lbnd

A two element array in which is returned the lower limits of the bounding box on each of the two axes of the graphics coordinate system (the base Frame of the Plot).

ubnd

A two element array in which is returned the upper limits of the bounding box on each of the two axes of the graphics coordinate system (the base Frame of the Plot).

Notes:

- An error results if the base Frame of the Plot is not 2-dimensional.

---

astBox Create a Box astBox

Description: This function creates a new Box and optionally initialises its attributes.

The Box class implements a Region which represents a box with sides parallel to the axes of a Frame (i.e. an area which encloses a given range of values on each axis). A Box is similar to an Interval, the only real difference being that the Interval class allows some axis limits to be unspecified. Note, a Box will only look like a box if the Frame geometry is approximately flat. For instance, a Box centred close to a pole in a SkyFrame will look more like a fan than a box (the Polygon class can be used to create a box-like region close to a pole).

Synopsis: `AstBox *astBox( AstFrame *frame, int form, const double point1[], const double point2[], AstRegion *unc, const char *options, ... )`

Parameters:

frame

A pointer to the Frame in which the region is defined. A deep copy is taken of the supplied Frame. This means that any subsequent changes made to the Frame using the supplied pointer will have no effect the Region.

form

Indicates how the box is described by the remaining parameters. A value of zero indicates that the box is specified by a centre position and a corner position. A value of one indicates that the box is specified by a two opposite corner positions.

point1

An array of double, with one element for each Frame axis (Naxes attribute). If "form" is zero, this array should contain the coordinates at the centre of the box. If "form" is one, it should contain the coordinates at the corner of the box which is diagonally opposite the corner specified by "point2".

point2

An array of double, with one element for each Frame axis (Naxes attribute) containing the coordinates at any corner of the box.

unc

An optional pointer to an existing Region which specifies the uncertainties associated with the boundary of the Box being created. The uncertainty in any point on the boundary of the Box is found by shifting the supplied "uncertainty" Region so that it is centred at the boundary point being considered. The area covered by the shifted uncertainty Region then represents the uncertainty in the boundary position. The uncertainty is assumed to be the same for all points.

If supplied, the uncertainty Region must be of a class for which all instances are centro-symetric (e.g. Box, Circle, Ellipse, etc.) or be a Prism containing centro-symetric component Regions. A deep copy of the supplied Region will be taken, so subsequent changes to the uncertainty Region using the supplied pointer will have no effect on the created Box. Alternatively, a NULL Object pointer may be supplied, in which case a default uncertainty is used equivalent to a box 1.0E-6 of the size of the Box being created.

The uncertainty Region has two uses: 1) when the astOverlap function compares two Regions for equality the uncertainty Region is used to determine the tolerance on the comparison, and 2) when a Region is mapped into a different coordinate system and subsequently simplified (using astSimplify), the uncertainties are used to determine if the transformed boundary can be accurately represented by a specific shape of Region.

options

Pointer to a null-terminated string containing an optional comma-separated list of attribute assignments to be used for initialising the new Box. The syntax used is identical to that for the astSet function and may include "printf" format specifiers identified by "%" symbols in the normal way.

...

If the "options" string contains "%" format specifiers, then an optional list of additional arguments may follow it in order to supply values to be substituted for these specifiers. The rules for supplying these are identical to those for the astSet function (and for the C "printf" function).

Returned Value:

astBox()

A pointer to the new Box.

Notes:

- A null Object pointer (AST__NULL) will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

Status Handling:

The protected interface to this function includes an extra parameter at the end of the parameter list descirbed above. This parameter is a pointer to the integer inherited status variable: "int ∗status".

---

astChannel Create a Channel astChannel

Description: This function creates a new Channel and optionally initialises its attributes.

A Channel implements low-level input/output for the AST library. Writing an Object to a Channel (using astWrite) will generate a textual representation of that Object, and reading from a Channel (using astRead) will create a new Object from its textual representation.

Normally, when you use a Channel, you should provide "source" and "sink" functions which connect it to an external data store by reading and writing the resulting text. By default, however, a Channel will read from standard input and write to standard output. Alternatively, a Channel can be told to read or write from specific text files using the SinkFile and SourceFile attributes, in which case no sink or source function need be supplied.

Synopsis: `AstChannel *astChannel( const char *(* source)( void ), void (* sink)( const char * ), const char *options, ... )`

Parameters:

source

Pointer to a source function that takes no arguments and returns a pointer to a null-terminated string. If no value has been set for the SourceFile attribute, this function will be used by the Channel to obtain lines of input text. On each invocation, it should return a pointer to the next input line read from some external data store, and a NULL pointer when there are no more lines to read.

If "source" is NULL and no value has been set for the SourceFile attribute, the Channel will read from standard input instead.

sink

Pointer to a sink function that takes a pointer to a null-terminated string as an argument and returns void. If no value has been set for the SinkFile attribute, this function will be used by the Channel to deliver lines of output text. On each invocation, it should deliver the contents of the string supplied to some external data store.

If "sink" is NULL, and no value has been set for the SinkFile attribute, the Channel will write to standard output instead.

options

Pointer to a null-terminated string containing an optional comma-separated list of attribute assignments to be used for initialising the new Channel. The syntax used is identical to that for the astSet function and may include "printf" format specifiers identified by "%" symbols in the normal way.

...

If the "options" string contains "%" format specifiers, then an optional list of additional arguments may follow it in order to supply values to be substituted for these specifiers. The rules for supplying these are identical to those for the astSet function (and for the C "printf" function).

Returned Value:

astChannel()

A pointer to the new Channel.

Notes:

- Application code can pass arbitrary data (such as file descriptors, etc) to source and sink functions using the astPutChannelData function. The source or sink function should use the astChannelData macro to retrieve this data.
- A null Object pointer (AST__NULL) will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

---

astChannelData Return a pointer to user-supplied data stored with a Channel astChannelData

Description: This macro is intended to be used within the source or sink functions associated with a Channel. It returns any pointer previously stored in the Channel (that is, the Channel that has invoked the source or sink function) using astPutChannelData.

This mechanism is a thread-safe alternative to passing file descriptors, etc, via static global variables.

Synopsis: `void *astChannelData`

Class Applicability:

Channel

This macro applies to all Channels.

Returned Value:

astChannelData

The pointer previously stored with the Channel using astPutChannelData. A NULL pointer will be returned if no such pointer has been stored with the Channel.

Notes:

- This routine is not available in the Fortran 77 interface to the AST library.

astCircle Create a Circle astCircle

Description: This function creates a new Circle and optionally initialises its attributes.

A Circle is a Region which represents a circle or sphere within the supplied Frame.

Synopsis: `AstCircle *astCircle( AstFrame *frame, int form, const double centre[], const double point[], AstRegion *unc, const char *options, ... )`

Parameters:

frame

A pointer to the Frame in which the region is defined. A deep copy is taken of the supplied Frame. This means that any subsequent changes made to the Frame using the supplied pointer will have no effect the Region.

form

Indicates how the circle is described by the remaining parameters. A value of zero indicates that the circle is specified by a centre position and a position on the circumference. A value of one indicates that the circle is specified by a centre position and a scalar radius.

centre

An array of double, with one element for each Frame axis (Naxes attribute) containing the coordinates at the centre of the circle or sphere.

point

If "form" is zero, then this array should have one element for each Frame axis (Naxes attribute), and should be supplied holding the coordinates at a point on the circumference of the circle or sphere. If "form" is one, then this array should have one element only which should be supplied holding the scalar radius of the circle or sphere, as a geodesic distance within the Frame.

unc

An optional pointer to an existing Region which specifies the uncertainties associated with the boundary of the Circle being created. The uncertainty in any point on the boundary of the Circle is found by shifting the supplied "uncertainty" Region so that it is centred at the boundary point being considered. The area covered by the shifted uncertainty Region then represents the uncertainty in the boundary position. The uncertainty is assumed to be the same for all points.

If supplied, the uncertainty Region must be of a class for which all instances are centro-symetric (e.g. Box, Circle, Ellipse, etc.) or be a Prism containing centro-symetric component Regions. A deep copy of the supplied Region will be taken, so subsequent changes to the uncertainty Region using the supplied pointer will have no effect on the created Circle. Alternatively, a NULL Object pointer may be supplied, in which case a default uncertainty is used equivalent to a box 1.0E-6 of the size of the Circle being created.

The uncertainty Region has two uses: 1) when the astOverlap function compares two Regions for equality the uncertainty Region is used to determine the tolerance on the comparison, and 2) when a Region is mapped into a different coordinate system and subsequently simplified (using astSimplify), the uncertainties are used to determine if the transformed boundary can be accurately represented by a specific shape of Region.

options

Pointer to a null-terminated string containing an optional comma-separated list of attribute assignments to be used for initialising the new Circle. The syntax used is identical to that for the astSet function and may include "printf" format specifiers identified by "%" symbols in the normal way.

...

If the "options" string contains "%" format specifiers, then an optional list of additional arguments may follow it in order to supply values to be substituted for these specifiers. The rules for supplying these are identical to those for the astSet function (and for the C "printf" function).

Returned Value:

astCircle()
A pointer to the new Circle.

Notes:

- A null Object pointer (AST__NULL) will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

---

astCirclePars Returns the geometric parameters of an Circle astCirclePars

Description: This function returns the geometric parameters describing the supplied Circle.

Synopsis: `void astCirclePars( AstCircle *this, double *centre, double *radius, double *p1 )`

Parameters:

this
Pointer to the Region.

centre
Pointer to an array in which to return the coordinates of the Circle centre. The length of this array should be no less than the number of axes in the associated coordinate system.

radius
Returned holding the radius of the Circle, as an geodesic distance in the associated coordinate system.

p1
Pointer to an array in which to return the coordinates of a point on the circumference of the Circle. The length of this array should be no less than the number of axes in the associated coordinate system. A NULL pointer can be supplied if the circumference position is not needed.

Notes:

- If the coordinate system represented by the Circle has been changed since it was first created, the returned parameters refer to the new (changed) coordinate system, rather than the original coordinate system. Note however that if the transformation from original to new coordinate system is non-linear, the shape represented by the supplied Circle object may not be an accurate circle.

---

astClear Clear attribute values for an Object astClear

Description: This function clears the values of a specified set of attributes for an Object. Clearing an attribute cancels any value that has previously been explicitly set for it, so that the standard default attribute value will subsequently be used instead. This also causes the astTest function to return the value zero for the attribute, indicating that no value has been set.

Synopsis: `void astClear( AstObject *this, const char *attrib )`

Parameters:

this

Pointer to the Object.

attrib

Pointer to a null-terminated character string containing a comma-separated list of the names of the attributes to be cleared.

Class Applicability:

Object

This function applies to all Objects.

Notes:

- Attribute names are not case sensitive and may be surrounded by white space.
- It does no harm to clear an attribute whose value has not been set.
- An error will result if an attempt is made to clear the value of a read-only attribute.

---

astClearStatus Clear the AST error status astClearStatus

Description: This macro resets the AST error status to an OK value, indicating that an error condition (if any) has been cleared.

Synopsis: `void astClearStatus`

Notes:

- If the AST error status is set to an error value (after an error), most AST functions will not execute and will simply return without action. Using astClearStatus will restore normal behaviour.

---

astClip Set up or remove clipping for a Plot astClip

Description: This function defines regions of a Plot which are to be clipped. Any subsequent graphical output created using the Plot will then be visible only within the unclipped regions of the plotting area. See also the Clip attribute.

Synopsis: `void astClip( AstPlot *this, int iframe, const double lbnd[], const double ubnd[] )`

Parameters:

this

Pointer to the Plot.

iframe

The index of the Frame within the Plot to which the clipping limits supplied in "lbnd" and "ubnd" (below) refer. Clipping may be applied to any of the coordinate systems associated with a Plot (as defined by the Frames it contains), so this index may take any value from 1 to the number of Frames in the Plot (Nframe attribute). In addition, the values AST__BASE and AST__CURRENT may be used to specify the base and current Frames respectively.

For example, a value of AST__CURRENT causes clipping to be performed in physical coordinates, while a value of AST__BASE would clip in graphical coordinates. Clipping may also be removed completely by giving a value of AST__NOFRAME. In this case any clipping bounds supplied (below) are ignored.

lbnd

An array with one element for each axis of the clipping Frame (identified by the index `"iframe"`). This should contain the lower bound, on each axis, of the region which is to remain visible (unclipped).

ubnd

An array with one element for each axis of the clipping Frame (identified by the index `"iframe"`). This should contain the upper bound, on each axis, of the region which is to remain visible (unclipped).

Notes:

- Only one clipping Frame may be active at a time. This function will deactivate any previously-established clipping Frame before setting up new clipping limits.
- The clipping produced by this function is in addition to that specified by the Clip attribute which occurs at the edges of the plotting area established when the Plot is created (see astPlot). The underlying graphics system may also impose further clipping.
- When testing a graphical position for clipping, it is first transformed into the clipping Frame. The resulting coordinate on each axis is then checked against the clipping limits (given by `"lbnd"` and `"ubnd"`). By default, a position is clipped if any coordinate lies outside these limits. However, if a non-zero value is assigned to the Plot's ClipOp attribute, then a position is only clipped if the coordinates on all axes lie outside their clipping limits.
- If the lower clipping limit exceeds the upper limit for any axis, then the sense of clipping for that axis is reversed (so that coordinate values lying between the limits are clipped instead of those lying outside the limits). To produce a `"hole"` in a coordinate space (that is, an internal region where nothing is plotted), you should supply all the bounds in reversed order, and set the ClipOp attribute for the Plot to a non-zero value.
- Either clipping limit may be set to the value AST__BAD, which is equivalent to setting it to infinity (or minus infinity for a lower bound) so that it is not used.
- If a graphical position results in any bad coordinate values (AST__BAD) when transformed into the clipping Frame, then it is treated (for the purposes of producing graphical output) as if it were clipped.
- When a Plot is used as a Mapping to transform points (e.g. using astTran2), any clipped output points are assigned coordinate values of AST__BAD.
- An error results if the base Frame of the Plot is not 2-dimensional.

---

astClone Clone (duplicate) an Object pointer astClone

Description: This function returns a duplicate pointer to an existing Object. It also increments the Object's RefCount attribute to keep track of how many pointers have been issued.

Note that this function is NOT equivalent to an assignment statement, as in general the two pointers will not have the same value.

Synopsis: `AstObject *astClone( AstObject *this )`

Parameters:

this

Original pointer to the Object.

Class Applicability:

Object

This function applies to all Objects.

Returned Value:

astClone()
A duplicate pointer to the same Object.

Notes:

- A null Object pointer (AST__NULL) will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

---

astCmpFrame Create a CmpFrame astCmpFrame

Description: This function creates a new CmpFrame and optionally initialises its attributes.

A CmpFrame is a compound Frame which allows two component Frames (of any class) to be merged together to form a more complex Frame. The axes of the two component Frames then appear together in the resulting CmpFrame (those of the first Frame, followed by those of the second Frame).

Since a CmpFrame is itself a Frame, it can be used as a component in forming further CmpFrames. Frames of arbitrary complexity may be built from simple individual Frames in this way.

Also since a Frame is a Mapping, a CmpFrame can also be used as a Mapping. Normally, a CmpFrame is simply equivalent to a UnitMap, but if either of the component Frames within a CmpFrame is a Region (a sub-class of Frame), then the CmpFrame will use the Region as a Mapping when transforming values for axes described by the Region. Thus input axis values corresponding to positions which are outside the Region will result in bad output axis values.

Synopsis: `AstCmpFrame *astCmpFrame( AstFrame *frame1, AstFrame *frame2, const char *options, ... )`

Parameters:

frame1
Pointer to the first component Frame.

frame2
Pointer to the second component Frame.

options
Pointer to a null-terminated string containing an optional comma-separated list of attribute assignments to be used for initialising the new CmpFrame. The syntax used is identical to that for the astSet function and may include "printf" format specifiers identified by "%" symbols in the normal way.

...
If the "options" string contains "%" format specifiers, then an optional list of additional arguments may follow it in order to supply values to be substituted for these specifiers. The rules for supplying these are identical to those for the astSet function (and for the C "printf" function).

Returned Value:

astCmpFrame()
A pointer to the new CmpFrame.

Notes:

- A null Object pointer (AST__NULL) will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

Status Handling:

The protected interface to this function includes an extra parameter at the end of the parameter list descirbed above. This parameter is a pointer to the integer inherited status variable: "int ∗status".

---

astCmpMap Create a CmpMap astCmpMap

Description: This function creates a new CmpMap and optionally initialises its attributes.

A CmpMap is a compound Mapping which allows two component Mappings (of any class) to be connected together to form a more complex Mapping. This connection may either be "in series" (where the first Mapping is used to transform the coordinates of each point and the second mapping is then applied to the result), or "in parallel" (where one Mapping transforms the earlier coordinates for each point and the second Mapping simultaneously transforms the later coordinates).

Since a CmpMap is itself a Mapping, it can be used as a component in forming further CmpMaps. Mappings of arbitrary complexity may be built from simple individual Mappings in this way.

Synopsis: `AstCmpMap *astCmpMap( AstMapping *map1, AstMapping *map2, int series, const char *options, ... )`

Parameters:

map1

Pointer to the first component Mapping.

map2

Pointer to the second component Mapping.

series

If a non-zero value is given for this parameter, the two component Mappings will be connected in series. A zero value requests that they are connected in parallel.

options

Pointer to a null-terminated string containing an optional comma-separated list of attribute assignments to be used for initialising the new CmpMap. The syntax used is identical to that for the astSet function and may include "printf" format specifiers identified by "%" symbols in the normal way.

...

If the "options" string contains "%" format specifiers, then an optional list of additional arguments may follow it in order to supply values to be substituted for these specifiers. The rules for supplying these are identical to those for the astSet function (and for the C "printf" function).

Returned Value:

astCmpMap()

A pointer to the new CmpMap.

Notes:

- If the component Mappings are connected in series, then using the resulting CmpMap to transform coordinates will cause the first Mapping to be applied, followed by the second Mapping. If the inverse CmpMap transformation is requested, the two component Mappings will be applied in both the reverse order and the reverse direction.
- When connecting two component Mappings in series, the number of output coordinates generated by the first Mapping (its Nout attribute) must equal the number of input coordinates accepted by the second Mapping (its Nin attribute).

- If the component Mappings of a CmpMap are connected in parallel, then the first Mapping will be used to transform the earlier input coordinates for each point (and to produce the earlier output coordinates) and the second Mapping will be used simultaneously to transform the remaining input coordinates (to produce the remaining output coordinates for each point). If the inverse transformation is requested, each Mapping will still be applied to the same coordinates, but in the reverse direction.
- When connecting two component Mappings in parallel, there is no restriction on the number of input and output coordinates for each Mapping.
- Note that the component Mappings supplied are not copied by astCmpMap (the new CmpMap simply retains a reference to them). They may continue to be used for other purposes, but should not be deleted. If a CmpMap containing a copy of its component Mappings is required, then a copy of the CmpMap should be made using astCopy.
- A null Object pointer (AST__NULL) will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

---

astCmpRegion Create a CmpRegion **astCmpRegion**

Description: This function creates a new CmpRegion and optionally initialises its attributes.

A CmpRegion is a Region which allows two component Regions (of any class) to be combined to form a more complex Region. This combination may be performed a boolean AND, OR or XOR (exclusive OR) operator. If the AND operator is used, then a position is inside the CmpRegion only if it is inside both of its two component Regions. If the OR operator is used, then a position is inside the CmpRegion if it is inside either (or both) of its two component Regions. If the XOR operator is used, then a position is inside the CmpRegion if it is inside one but not both of its two component Regions. Other operators can be formed by negating one or both component Regions before using them to construct a new CmpRegion.

The two component Region need not refer to the same coordinate Frame, but it must be possible for the astConvert function to determine a Mapping between them (an error will be reported otherwise when the CmpRegion is created). For instance, a CmpRegion may combine a Region defined within an ICRS SkyFrame with a Region defined within a Galactic SkyFrame. This is acceptable because the SkyFrame class knows how to convert between these two systems, and consequently the astConvert function will also be able to convert between them. In such cases, the second component Region will be mapped into the coordinate Frame of the first component Region, and the Frame represented by the CmpRegion as a whole will be the Frame of the first component Region.

Since a CmpRegion is itself a Region, it can be used as a component in forming further CmpRegions. Regions of arbitrary complexity may be built from simple individual Regions in this way.

Synopsis: `AstCmpRegion *astCmpRegion( AstRegion *region1, AstRegion *region2, int oper, const char *options, ... )`

Parameters:

region1
Pointer to the first component Region.

region2
Pointer to the second component Region. This Region will be transformed into the coordinate Frame of the first region before use. An error will be reported if this is not possible.

oper
The boolean operator with which to combine the two Regions. This must be one of the symbolic constants AST__AND, AST__OR or AST__XOR.

options

Pointer to a null-terminated string containing an optional comma-separated list of attribute assignments to be used for initialising the new CmpRegion. The syntax used is identical to that for the astSet function and may include "printf" format specifiers identified by "%" symbols in the normal way.

...

If the "options" string contains "%" format specifiers, then an optional list of additional arguments may follow it in order to supply values to be substituted for these specifiers. The rules for supplying these are identical to those for the astSet function (and for the C "printf" function).

Returned Value:

astCmpRegion()

A pointer to the new CmpRegion.

Notes:

- If one of the supplied Regions has an associated uncertainty, that uncertainty will also be used for the returned CmpRegion. If both supplied Regions have associated uncertainties, the uncertainty associated with the first Region will be used for the returned CmpRegion.
- Deep copies are taken of the supplied Regions. This means that any subsequent changes made to the component Regions using the supplied pointers will have no effect on the CmpRegion.
- A null Object pointer (AST__NULL) will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

---

astColumnName Get the name of the column at a given index within the Table astColumnName

Description: This function returns a string holding the name of the column with the given index within the Table.

This function is intended primarily as a means of iterating round all the columns in a Table. For this purpose, the number of columns in the Table is given by the Ncolumn attribute of the Table. This function could then be called in a loop, with the index value going from zero to one less than Ncolumn.

Note, the index associated with a column decreases monotonically with the age of the column: the oldest Column in the Table will have index one, and the Column added most recently to the Table will have the largest index.

Synopsis: `const char *astColumnName( AstTable *this, int index )`

Parameters:

this

Pointer to the Table.

index

The index into the list of columns. The first column has index one, and the last has index "Ncolumn".

Returned Value:

astColumnName()

A pointer to a null-terminated string containing the upper case column name.

Notes:

- The returned pointer is guaranteed to remain valid and the string to which it points will not be over-written for a total of 50 successive invocations of this function. After this, the memory containing the string may be re-used, so a copy of the string should be made if it is needed for longer than this.
- A NULL pointer will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

---

astColumnNull Get or set the null value for an integer column of a FITS table astColumnNull

Description: This function allows a null value to be stored with a named integer-valued column in a FitsTable. The supplied null value is assigned to the TNULLn keyword in the FITS header associated with the FitsTable. A value in the named column is then considered to be null if 1) it equals the null value supplied to this function, or 2) no value has yet been stored in the cell.

As well as setting a new null value, this function also returns the previous null value. If no null value has been set previously, a default value will be returned. This default will be an integer value that does not currently occur anywhere within the named column. If no such value can be found, what happens depends on whether the column contains any cells in which no values have yet been stored. If so, an error will be reported. Otherwise (i.e. if there are no null values in the column), an arbitrary value of zero will be returned as the function value, and no TNULLn keyword will be stored in the FITS header.

A flag is returned indicating if the returned null value was set explicitly by a previous call to this function, or is a default value.

A second flag is returned indicating if the named column contains any null values (i.e. values equal to the supplied null value, or cells to which no value has yet been assigned).

Synopsis: `int astColumnNull( AstFitsTable *this, const char *column, int set, int newval, int *wasset, int *hasnull )`

Parameters:

this

Pointer to the Table.

column

The character string holding the name of the column. Trailing spaces are ignored.

set

If non-zero, the value supplied for parameter "newval" will be stored as the current null value, replacing any value set by a previous call to this function. If zero, the value supplied for parameter "newval" is ignored and the current null value is left unchanged.

newval

The new null value to use. Ignored if "set" is zero. An error will be reported if the supplied value is outside the range of values that can be stored in the integer data type associated with the column.

wasset

Pointer to an int that will be returned non-zero if the returned null value was set previously via an earlier invocation of this function. Zero is returned otherwise. If the named column does not exist, or an error occurs, a value of zero is returned.

hasnull

Pointer to an int that will be returned non-zero if and only if the named column currently contains any values equal to the null value on exit (i.e. "newval" if "set" is non-zero, or the returned function value otherwise), or contains any empty cells. If the named column does not exist, or an error occurs, a value of zero is returned. If a NULL pointer is supplied for "hasnull", no check on the presence of null values will be performed.

Returned Value:

astColumnNull()
The null value that was in use on entry to this function. If a null value has been set by a previous invocation of this function, it will be returned. Otherwise, if "set" is non-zero, the supplied "newval" value is returned. Otherwise, a default value is chosen (if possible) that does not currently occur in the named column. If all available values are in use in the column, an error is reported if and only if the column contains any empty cells. Otherwise, a value of zero is returned. A value of zero is also returned if the named column does not exist, or an error occurs.

Notes:

- The FITS binary table definition allows only integer-valued columns to have an associated null value. This routine will return without action if the column is not integer-valued.

---

astColumnShape Returns the shape of the values in a named column astColumnShape

Description: This function returns the number of dimensions spaned by each value in a named column of a Table, together with the length of each dimension. These are the values supplied when the column was created using astAddColumn.

Synopsis: `void astColumnShape( AstTable *this, const char *column, int mxdim, int *ndim, int *dims )`

Parameters:

this
Pointer to the Table.

column
The character string holding the upper case name of the column. Trailing spaces are ignored.

mxdim
The length of the "dims" array.

ndim
Pointer to an int in which to return the number of dimensions spanned by values in the named column. This will be zero if the column contains scalar values.

dims
Pointer to an array in which to return the length of each dimension. Any excess trailing elements will be filled with the value 1.

Notes:

- No error is reported if the requested column cannot be found in the given Table. A value of zero is returned for "ndim" and the supplied values in "dims" are left unchanged.
- A value of zero is returned for "ndim" if an error occurs.

---

astColumnSize Get the number of bytes needed to hold a full column of data astColumnSize

Description: This function returns the number of bytes of memory that must be allocated prior to retrieving the data from a column using astGetColumnData.

Synopsis: `size_t astColumnSize( AstFitsTable *this, const char *column, int *hasnull )`

Parameters:

this

Pointer to the Table.

column

The character string holding the name of the column. Trailing spaces are ignored.

Returned Value:

astColumnNull()

The number of bytes required to store the column data.

Notes:

- An error will be reported if the named column does not exist in the FitsTable.
- Zero will be returned as the function value in an error occurs.

---

astConvert Determine how to convert between two coordinate systems astConvert

Description: This function compares two Frames and determines whether it is possible to convert between the coordinate systems which they represent. If conversion is possible, it returns a FrameSet which describes the conversion and which may be used (as a Mapping) to transform coordinate values in either direction.

The same function may also be used to determine how to convert between two FrameSets (or between a Frame and a FrameSet, or vice versa). This mode is intended for use when (for example) two images have been calibrated by attaching a FrameSet to each. astConvert might then be used to search for a celestial coordinate system that both images have in common, and the result could then be used to convert between the pixel coordinates of both images – having effectively used their celestial coordinate systems to align them.

When using FrameSets, there may be more than one possible intermediate coordinate system in which to perform the conversion (for instance, two FrameSets might both have celestial coordinates, detector coordinates, pixel coordinates, etc.). A comma-separated list of coordinate system domains may therefore be given which defines a priority order to use when selecting the intermediate coordinate system. The path used for conversion must go via an intermediate coordinate system whose Domain attribute matches one of the domains given. If conversion cannot be achieved using the first domain, the next one is considered, and so on, until success is achieved.

Synopsis: `AstFrameSet *astConvert( AstFrame *from, AstFrame *to, const char *domainlist )`

Parameters:

from

Pointer to a Frame which represents the "source" coordinate system. This is the coordinate system in which you already have coordinates available.

If a FrameSet is given, its current Frame (as determined by its Current attribute) is taken to describe the source coordinate system. Note that the Base attribute of this FrameSet may be modified by this function to indicate which intermediate coordinate system was used (see under "FrameSets" in the "Applicability" section for details).

to

Pointer to a Frame which represents the "destination" coordinate system. This is the coordinate system into which you wish to convert your coordinates.

If a FrameSet is given, its current Frame (as determined by its Current attribute) is taken to describe the destination coordinate system. Note that the Base attribute of this FrameSet may be modified by this function to indicate which intermediate coordinate system was used (see under "FrameSets" in the "Applicability" section for details).

domainlist

Pointer to a null-terminated character string containing a comma-separated list of Frame domains. This may be used to define a priority order for the different intermediate coordinate systems that might be used to perform the conversion.

The function will first try to obtain a conversion by making use only of an intermediate coordinate system whose Domain attribute matches the first domain in this list. If this fails, the second domain in the list will be used, and so on, until conversion is achieved. A blank domain (e.g. two consecutive commas) indicates that all coordinate systems should be considered, regardless of their domains.

This list is case-insensitive and all white space is ignored. If you do not wish to restrict the domain in this way, you should supply an empty string. This is normally appropriate if either of the source or destination coordinate systems are described by Frames (rather than FrameSets), since there is then usually only one possible choice of intermediate coordinate system.

Class Applicability:

DSBSpecFrame

If the AlignSideBand attribute is non-zero, alignment occurs in the upper sideband expressed within the spectral system and standard of rest given by attributes AlignSystem and AlignStdOfRest. If AlignSideBand is zero, the two DSBSpecFrames are aligned as if they were simple SpecFrames (i.e. the SideBand is ignored).

Frame

This function applies to all Frames. Alignment occurs within the coordinate system given by attribute AlignSystem.

FrameSet

If either of the "from" or "to" parameters is a pointer to a FrameSet, then astConvert will attempt to convert from the coordinate system described by the current Frame of the "from" FrameSet to that described by the current Frame of the "to" FrameSet.

To achieve this, it will consider all of the Frames within each FrameSet as a possible way of reaching an intermediate coordinate system that can be used for the conversion. There is then the possibility that more than one conversion path may exist and, unless the choice is sufficiently restricted by the "domainlist" string, the sequence in which the Frames are considered can be important. In this case, the search for a conversion path proceeds as follows:

- Each field in the "domainlist" string is considered in turn.
- The Frames within each FrameSet are considered in a specific order: (1) the base Frame is always considered first, (2) after this come all the other Frames in Frame-index order (but omitting the base and current Frames), (3) the current Frame is always considered last. However, if either FrameSet's Invert attribute is set to a non-zero value (so that the FrameSet is inverted), then its Frames are considered in reverse order. (Note that this still means that the base Frame is considered first and the current Frame last, because the Invert value will also cause these Frames to swap places.)

- All source Frames are first considered (in the appropriate order) for conversion to the first destination Frame. If no suitable intermediate coordinate system emerges, they are then considered again for conversion to the second destination Frame (in the appropriate order), and so on.
- Generally, the first suitable intermediate coordinate system found is used. However, the overall Mapping between the source and destination coordinate systems is also examined. Preference is given to cases where both the forward and inverse transformations are defined (as indicated by the TranForward and TranInverse attributes). If only one transformation is defined, the forward one is preferred.
- If the domain of the intermediate coordinate system matches the current `"domainlist"` field, the conversion path is accepted. Otherwise, the next `"domainlist"` field is considered and the process repeated.

If conversion is possible, the Base attributes of the two FrameSets will be modified on exit to identify the Frames used to access the intermediate coordinate system which was finally accepted.

Note that it is possible to force a particular Frame within a FrameSet to be used as the basis for the intermediate coordinate system, if it is suitable, by (a) focussing attention on it by specifying its domain in the `"domainlist"` string, or (b) making it the base Frame, since this is always considered first.

SpecFrame

Alignment occurs within the spectral system and standard of rest given by attributes AlignSystem and AlignStdOfRest.

TimeFrame

Alignment occurs within the time system and time scale given by attributes AlignSystem and AlignTimeScale.

Returned Value:

astConvert()

If the requested coordinate conversion is possible, the function returns a pointer to a FrameSet which describes the conversion. Otherwise, a null Object pointer (AST__NULL) is returned without error.

If a FrameSet is returned, it will contain two Frames. Frame number 1 (its base Frame) will describe the source coordinate system, corresponding to the `"from"` parameter. Frame number 2 (its current Frame) will describe the destination coordinate system, corresponding to the `"to"` parameter. The Mapping which inter-relates these two Frames will perform the required conversion between their respective coordinate systems.

Note that a FrameSet may be used both as a Mapping and as a Frame. If the result is used as a Mapping (e.g. with astTran2), then it provides a means of converting coordinates from the source to the destination coordinate system (or vice versa if its inverse transformation is selected). If it is used as a Frame, its attributes will describe the destination coordinate system.

Examples:

`cvt = astConvert( a, b, "" );`

Attempts to convert between the coordinate systems represented by `"a"` and `"b"` (assumed to be Frames). If successful, a FrameSet is returned via the `"cvt"` pointer which may be used to apply the conversion to sets of coordinates (e.g. using astTran2).

`cvt = astConvert( astSkyFrame(""), astSkyFrame("Equinox=2005"), "" );`

Creates a FrameSet which describes precession in the default FK5 celestial coordinate system between equinoxes J2000 (also the default) and J2005. The returned `"cvt"` pointer may then be passed to astTran2 to apply this precession correction to any number of coordinate values given in radians.

Note that the returned FrameSet also contains information about how to format coordinate values. This means that setting its Report attribute to 1 is a simple way to obtain printed output (formatted in sexagesimal notation) to show the coordinate values before and after conversion.

`cvt = astConvert( a, b, "sky,detector," );`

Attempts to convert between the coordinate systems represented by the current Frames of "a" and "b" (now assumed to be FrameSets), via the intermediate "SKY" coordinate system. This, by default, is the Domain associated with a celestial coordinate system represented by a SkyFrame.

If this fails (for example, because either FrameSet lacks celestial coordinate information), then the user-defined "DETECTOR" coordinate system is used instead. If this also fails, then all other possible ways of achieving conversion are considered before giving up.

The returned pointer "cvt" indicates whether conversion was possible and will have the value AST__NULL if it was not. If conversion was possible, "cvt" will point at a new FrameSet describing the conversion.

The Base attributes of the two FrameSets will be set by astConvert to indicate which of their Frames was used for the intermediate coordinate system. This means that you can subsequently determine which coordinate system was used by enquiring the Domain attribute of either base Frame.

Notes:

- The Mapping represented by the returned FrameSet results in alignment taking place in the coordinate system specified by the AlignSystem attribute of the "to" Frame. See the description of the AlignSystem attribute for further details.
- When aligning (say) two images, which have been calibrated by attaching FrameSets to them, it is usually necessary to convert between the base Frames (representing "native" pixel coordinates) of both FrameSets. This may be achieved by inverting the FrameSets (e.g. using astInvert) so as to interchange their base and current Frames before using astConvert.
- A null Object pointer (AST__NULL) will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

---

astCopy Copy an Object astCopy

Description: This function creates a copy of an Object and returns a pointer to the resulting new Object. It makes a "deep" copy, which contains no references to any other Object (i.e. if the original Object contains references to other Objects, then the actual data are copied, not simply the references). This means that modifications may safely be made to the copy without indirectly affecting any other Object.

Synopsis: `AstObject *astCopy( const AstObject *this )`

Parameters:

this

Pointer to the Object to be copied.

Class Applicability:

Object

This function applies to all Objects.

Returned Value:

astCopy()

Pointer to the new Object.

Notes:

- A null Object pointer (AST__NULL) will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

---

astCurrentTime Return the current system time astCurrentTime

Description: This function returns the current system time, represented in the form specified by the supplied TimeFrame. That is, the returned floating point value should be interpreted using the attribute values of the TimeFrame. This includes System, TimeOrigin, LTOffset, TimeScale, and Unit.

Synopsis: `double astCurrentTime( AstTimeFrame *this )`

Parameters:

this
Pointer to the TimeFrame.

Returned Value:

astCurrentTime()
A TimeFrame axis value representing the current system time.

Notes:

- Values of AST__BAD will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.
- It is assumes that the system time (returned by the C time() function) follows the POSIX standard, representing a continuous monotonic increasing count of SI seconds since the epoch 00:00:00 UTC 1 January 1970 AD (equivalent to TAI with a constant offset). Resolution is one second.
- An error will be reported if the TimeFrame has a TimeScale value which cannot be converted to TAI (e.g. "angular" systems such as UT1, GMST, LMST and LAST).
- Any inaccuracy in the system clock will be reflected in the value returned by this function.

---

astCurve Draw a geodesic curve astCurve

Description: This function draws a geodesic curve between two points in the physical coordinate system of a Plot. The curve drawn is the path of shortest distance joining the two points (as defined by the astDistance function for the current Frame of the Plot). For example, if the current Frame is a basic Frame, then the curve joining the two points will be a straight line in physical coordinate space. If the current Frame is more specialised and describes, for instance, a sky coordinate system, then the geodesic curve would be a great circle in physical coordinate space passing through the two sky positions given.

Note that the geodesic curve is transformed into graphical coordinate space for plotting, so that a straight line in physical coordinates may result in a curved line being drawn if the Mapping involved is non-linear. Any discontinuities in the Mapping between physical and graphical coordinates are catered for, as is any clipping established using astClip.

If you need to draw many geodesic curves end-to-end, then the astPolyCurve function is equivalent to repeatedly using astCurve, but will usually be more efficient.

If you need to draw curves which are not geodesics, see astGenCurve or astGridLine.

Synopsis: `void astCurve( AstPlot *this, const double start[], const double finish[] )`

Parameters:

this

Pointer to the Plot.

start

An array, with one element for each axis of the Plot, giving the physical coordinates of the first point on the geodesic curve.

finish

An array, with one element for each axis of the Plot, giving the physical coordinates of the second point on the geodesic curve.

Notes:

- No curve is drawn if either of the "start" or "finish" arrays contains any coordinates with the value AST__BAD.
- An error results if the base Frame of the Plot is not 2-dimensional.
- An error also results if the transformation between the current and base Frames of the Plot is not defined (i.e. the Plot's TranInverse attribute is zero).

---

astDSBSpecFrame Create a DSBSpecFrame **astDSBSpecFrame**

Description: This function creates a new DSBSpecFrame and optionally initialises its attributes.

A DSBSpecFrame is a specialised form of SpecFrame which represents positions in a spectrum obtained using a dual sideband instrument. Such an instrument produces a spectrum in which each point contains contributions from two distinctly different frequencies, one from the "lower side band" (LSB) and one from the "upper side band" (USB). Corresponding LSB and USB frequencies are connected by the fact that they are an equal distance on either side of a fixed central frequency known as the "Local Oscillator" (LO) frequency.

When quoting a position within such a spectrum, it is necessary to indicate whether the quoted position is the USB position or the corresponding LSB position. The SideBand attribute provides this indication. Another option that the SideBand attribute provides is to represent a spectral position by its topocentric offset from the LO frequency.

In practice, the LO frequency is specified by giving the distance from the LO frequency to some "central" spectral position. Typically this central position is that of some interesting spectral feature. The distance from this central position to the LO frequency is known as the "intermediate frequency" (IF). The value supplied for IF can be a signed value in order to indicate whether the LO frequency is above or below the central position.

Synopsis: `AstDSBSpecFrame *astDSBSpecFrame( const char *options, ... )`

Parameters:

options

Pointer to a null-terminated string containing an optional comma-separated list of attribute assignments to be used for initialising the new DSBSpecFrame. The syntax used is identical to that for the astSet function and may include "printf" format specifiers identified by "%" symbols in the normal way.

...

If the "options" string contains "%" format specifiers, then an optional list of additional arguments may follow it in order to supply values to be substituted for these specifiers. The rules for supplying these are identical to those for the astSet function (and for the C "printf" function).

Returned Value:

astDSBSpecFrame()
A pointer to the new DSBSpecFrame.

Notes:

- A null Object pointer (AST__NULL) will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

---

astDecompose Decompose a Mapping into two component Mappings astDecompose

Description: This function returns pointers to two Mappings which, when applied either in series or parallel, are equivalent to the supplied Mapping.

Since the Frame class inherits from the Mapping class, Frames can be considered as special types of Mappings and so this method can be used to decompose either CmpMaps or CmpFrames.

Synopsis: `void astDecompose( AstMapping *this, AstMapping **map1, AstMapping **map2, int *series, int *invert1, int *invert2 )`

Parameters:

this
Pointer to the Mapping.

map1
Address of a location to receive a pointer to first component Mapping.

map2
Address of a location to receive a pointer to second component Mapping.

series
Address of a location to receive a value indicating if the component Mappings are applied in series or parallel. A non-zero value means that the supplied Mapping is equivalent to applying map1 followed by map2 in series. A zero value means that the supplied Mapping is equivalent to applying map1 to the lower numbered axes and map2 to the higher numbered axes, in parallel.

invert1
The value of the Invert attribute to be used with map1.

invert2
The value of the Invert attribute to be used with map2.

Class Applicability:

CmpMap
If the supplied Mapping is a CmpMap, then map1 and map2 will be returned holding pointers to the component Mappings used to create the CmpMap, either in series or parallel. Note, changing the Invert attribute of either of the component Mappings using the returned pointers will have no effect on the supplied CmpMap. This is because the CmpMap remembers and uses the original settings of the Invert attributes (that is, the values of the Invert attributes when the CmpMap was first created). These are the Invert values which are returned in invert1 and invert2.

TranMap
If the supplied Mapping is a TranMap, then map1 and map2 will be returned holding pointers to the forward and inverse Mappings represented by the TranMap (zero will be returned for series). Note, changing the Invert attribute of either of the component Mappings using the

returned pointers will have no effect on the supplied TranMap. This is because the TranMap remembers and uses the original settings of the Invert attributes (that is, the values of the Invert attributes when the TranMap was first created). These are the Invert values which are returned in invert1 and invert2.

Mapping

For any class of Mapping other than a CmpMap, map1 will be returned holding a clone of the supplied Mapping pointer, and map2 will be returned holding a NULL pointer. Invert1 will be returned holding the current value of the Invert attribute for the supplied Mapping, and invert2 will be returned holding zero.

CmpFrame

If the supplied Mapping is a CmpFrame, then map1 and map2 will be returned holding pointers to the component Frames used to create the CmpFrame. The component Frames are considered to be in applied in parallel.

Frame

For any class of Frame other than a CmpFrame, map1 will be returned holding a clone of the supplied Frame pointer, and map2 will be returned holding a NULL pointer.

Notes:

- The returned Invert values should be used in preference to the current values of the Invert attribute in map1 and map2. This is because the attributes may have changed value since the Mappings were combined.
- Any changes made to the component Mappings using the returned pointers will be reflected in the supplied Mapping.

---

astDelFits Delete the current FITS card in a FitsChan astDelFits

Description: This function deletes the current FITS card from a FitsChan. The current card may be selected using the Card attribute (if its index is known) or by using astFindFits (if only the FITS keyword is known).

After deletion, the following card becomes the current card.

Synopsis: `void astDelFits( AstFitsChan *this )`

Parameters:

this

Pointer to the FitsChan.

Notes:

- This function returns without action if the FitsChan is initially positioned at the "end-of-file" (i.e. if the Card attribute exceeds the number of cards in the FitsChan).
- If there are no subsequent cards in the FitsChan, then the Card attribute is left pointing at the "end-of-file" after deletion (i.e. is set to one more than the number of cards in the FitsChan).

astDelete Delete an Object astDelete

Description: This function deletes an Object, freeing all resources associated with it and rendering any remaining pointers to the Object invalid.

Note that deletion is unconditional, regardless of whether other pointers to the Object are still in use (possibly within other Objects). A safer approach is to defer deletion, until all references to an Object have expired, by using astBegin/astEnd (together with astClone and astAnnul if necessary).

Synopsis: `AstObject *astDelete( AstObject *this )`

Parameters:

this

Pointer to the Object to be deleted.

Class Applicability:

Object

This function applies to all Objects.

Returned Value:

astDelete()

A null Object pointer (AST__NULL) is always returned.

Notes:

- This function attempts to execute even if the AST error status is set on entry, although no further error report will be made if it subsequently fails under these circumstances.

astDistance Calculate the distance between two points in a Frame astDistance

Description: This function finds the distance between two points whose Frame coordinates are given. The distance calculated is that along the geodesic curve that joins the two points.

For example, in a basic Frame, the distance calculated will be the Cartesian distance along the straight line joining the two points. For a more specialised Frame describing a sky coordinate system, however, it would be the distance along the great circle passing through two sky positions.

Synopsis: `double astDistance( AstFrame *this, const double point1[], const double point2[] )`

Parameters:

this

Pointer to the Frame.

point1

An array of double, with one element for each Frame axis (Naxes attribute) containing the coordinates of the first point.

point2

An array of double, with one element for each Frame axis containing the coordinates of the second point.

Returned Value:

astDistance

The distance between the two points.

Notes:

- This function will return a "bad" result value (AST__BAD) if any of the input coordinates has this value.
- A "bad" value will also be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

---

astDownsize Reduce the number of vertices in a Polygon astDownsize

Description: This function returns a pointer to a new Polygon that contains a subset of the vertices in the supplied Polygon. The subset is chosen so that the returned Polygon is a good approximation to the supplied Polygon, within the limits specified by the supplied parameter values. That is, the density of points in the returned Polygon is greater at points where the curvature of the boundary of the supplied Polygon is greater.

Synopsis: `AstPolygon *astDownsize( AstPolygon *this, double maxerr, int maxvert )`

Parameters:

this

Pointer to the Polygon.

maxerr

The maximum allowed discrepancy between the supplied and returned Polygons, expressed as a geodesic distance within the Polygon's coordinate frame. If this is zero or less, the returned Polygon will have the number of vertices specified by maxvert.

maxvert

The maximum allowed number of vertices in the returned Polygon. If this is less than 3, the number of vertices in the returned Polygon will be the minimum needed to achieve the maximum discrepancy specified by maxerr.

Returned Value:

astDownsize()

Pointer to the new Polygon.

Notes:

- A null Object pointer (AST__NULL) will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

---

astEBuf End the current graphical buffering context astEBuf

Description: This function ends the current graphics buffering context. It should match a corresponding call to the astBBuf function.

Synopsis: `void astEBuf( AstPlot *this )`

Parameters:

this

Pointer to the Plot.

Notes:

- The nature of the buffering is determined by the underlying graphics system (as defined by the current grf module). Each call to this function simply invokes the astGEBuf function in the grf module.

---

astEllipse Create a Ellipse astEllipse

Description: This function creates a new Ellipse and optionally initialises its attributes.

A Ellipse is a Region which represents a elliptical area within the supplied 2-dimensional Frame.

Synopsis: `AstEllipse *astEllipse( AstFrame *frame, int form, const double centre[2], const double point1[2], const double point2[2], AstRegion *unc, const char *options, ... )`

Parameters:

frame

A pointer to the Frame in which the region is defined. It must have exactly 2 axes. A deep copy is taken of the supplied Frame. This means that any subsequent changes made to the Frame using the supplied pointer will have no effect the Region.

form

Indicates how the ellipse is described by the remaining parameters. A value of zero indicates that the ellipse is specified by a centre position and two positions on the circumference. A value of one indicates that the ellipse is specified by its centre position, the half-lengths of its two axes, and the orientation of its first axis.

centre

An array of 2 doubles, containing the coordinates at the centre of the ellipse.

point1

An array of 2 doubles. If "form" is zero, this array should contain the coordinates of one of the four points where an axis of the ellipse crosses the circumference of the ellipse. If "form" is one, it should contain the lengths of semi-major and semi-minor axes of the ellipse, given as geodesic distances within the Frame.

point2

An array of 2 doubles. If "form" is zero, this array should containing the coordinates at some other point on the circumference of the ellipse, distinct from "point1". If "form" is one, the first element of this array should hold the angle between the second axis of the Frame and the first ellipse axis (i.e. the ellipse axis which is specified first in the "point1" array), and the second element will be ignored. The angle should be given in radians, measured positive in the same sense as rotation from the positive direction of the second Frame axis to the positive direction of the first Frame axis.

unc

An optional pointer to an existing Region which specifies the uncertainties associated with the boundary of the Box being created. The uncertainty in any point on the boundary of the Box is found by shifting the supplied "uncertainty" Region so that it is centred at the boundary point being considered. The area covered by the shifted uncertainty Region then represents the uncertainty in the boundary position. The uncertainty is assumed to be the same for all points.

If supplied, the uncertainty Region must be of a class for which all instances are centro-symetric (e.g. Box, Circle, Ellipse, etc.) or be a Prism containing centro-symetric component Regions. A deep copy of the supplied Region will be taken, so subsequent changes to the uncertainty Region using the supplied pointer will have no effect on the created Box. Alternatively, a NULL Object pointer may be supplied, in which case a default uncertainty is used equivalent to a box 1.0E-6 of the size of the Box being created.

The uncertainty Region has two uses: 1) when the astOverlap function compares two Regions for equality the uncertainty Region is used to determine the tolerance on the comparison, and 2) when a Region is mapped into a different coordinate system and subsequently simplified (using astSimplify), the uncertainties are used to determine if the transformed boundary can be accurately represented by a specific shape of Region.

options
Pointer to a null-terminated string containing an optional comma-separated list of attribute assignments to be used for initialising the new Ellipse. The syntax used is identical to that for the astSet function and may include "printf" format specifiers identified by "%" symbols in the normal way.

...
If the "options" string contains "%" format specifiers, then an optional list of additional arguments may follow it in order to supply values to be substituted for these specifiers. The rules for supplying these are identical to those for the astSet function (and for the C "printf" function).

Returned Value:

astEllipse()
A pointer to the new Ellipse.

Notes:

- A null Object pointer (AST__NULL) will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

---

astEllipsePars Returns the geometric parameters of an Ellipse astEllipsePars

Description: This function returns the geometric parameters describing the supplied ellipse.

Synopsis: `void astEllipsePars( AstEllipse *this, double centre[2], double *a, double *b, double *angle, double p1[2], double p2[2] )`

Parameters:

this
Pointer to the Region.

centre
The coordinates of the Ellipse centre are returned in this arrays.

a
Returned holding the half-length of the first axis of the ellipse.

b
Returned holding the half-length of the second axis of the ellipse.

angle
If the coordinate system in which the Ellipse is defined has axes (X,Y), then "*angle" is returned holding the angle from the positive direction of the Y axis to the first axis of the ellipse, in radians. Positive rotation is in the same sense as rotation from the positive direction of Y to the positive direction of X.

p1
An array in which to return the coordinates at one of the two ends of the first axis of the ellipse. A NULL pointer can be supplied if these coordinates are not needed.

p2

An array in which to return the coordinates at one of the two ends of the second axis of the ellipse. A NULL pointer can be supplied if these coordinates are not needed.

Notes:

- If the coordinate system represented by the Ellipse has been changed since it was first created, the returned parameters refer to the new (changed) coordinate system, rather than the original coordinate system. Note however that if the transformation from original to new coordinate system is non-linear, the shape represented by the supplied Ellipse object may not be an accurate ellipse.
- Values of AST__BAD are returned for the parameters without error if the ellipse is degenerate or undefined.

---

astEmptyFits Delete all cards in a FitsChan astEmptyFits

Description: This function deletes all cards and associated information from a FitsChan.

Synopsis: `void astEmptyFits( AstFitsChan *this )`

Parameters:

this

Pointer to the FitsChan.

Notes:

- This method simply deletes the cards currently in the FitsChan. Unlike astWriteFits, they are not first written out to the sink function or sink file.
- Any Tables or warnings stored in the FitsChan are also deleted.
- This method attempt to execute even if an error has occurred previously.

---

astEnd End an AST context astEnd

Description: This macro invokes a function to end an AST context which was begun with a matching invocation of astBegin. Any Object pointers created within this context will be annulled (just as if astAnnul had been invoked) and will cease to be valid afterwards, unless they have previously been exported using astExport or rendered exempt using astExempt. If annulling a pointer causes an Object's RefCount attribute to fall to zero (which happens when the last pointer to it is annulled), then the Object will be deleted.

Synopsis: `void astEnd`

Class Applicability:

Object

This macro applies to all Objects.

Notes:

- astEnd attempts to execute even if the AST error status is set.
- Contexts delimited by astBegin and astEnd may be nested to any depth.

---

astEscapes Control whether graphical escape sequences are included in strings astEscapes

Description: The Plot class defines a set of escape sequences which can be included within a text string in order to control the appearance of sub-strings within the text. See the Escape attribute for a description of these escape sequences. It is usually inappropriate for AST to return strings containing such escape sequences when called by application code. For instance, an application which displays the value of the Title attribute of a Frame usually does not want the displayed string to include potentially long escape sequences which a human read would have difficuly interpreting. Therefore the default behaviour is for AST to strip out such escape sequences when called by application code. This default behaviour can be changed using this function.

Synopsis: `int astEscapes( int new_value )`

Parameters:

new_value

A flag which indicates if escapes sequences should be included in returned strings. If zero is supplied, escape sequences will be stripped out of all strings returned by any AST function. If a positive value is supplied, then any escape sequences will be retained in the value returned to the caller. If a negative value is supplied, the current value of the flag will be left unchanged.

Class Applicability:

Object

This macro applies to all Objects.

Returned Value:

astEscapes

The value of the flag on entry to this function.

Notes:

- This function also controls whether the astStripEscapes function removes escape sequences from the supplied string, or returns the supplied string without change.
- This function attempts to execute even if an error has already occurred.

---

astExempt Exempt an Object pointer from AST context handling astExempt

Description: This function exempts an Object pointer from AST context handling, as implemented by astBegin and astEnd. This means that the pointer will not be affected when astEnd is invoked and will remain active until the end of the program, or until explicitly annulled using astAnnul.

If possible, you should avoid using this function when writing applications. It is provided mainly for developers of other libraries, who may wish to retain references to AST Objects in internal data structures, and who therefore need to avoid the effects of astBegin and astEnd.

Synopsis: `void astExempt( AstObject *this )`

Parameters:

this

Object pointer to be exempted from context handling.

Class Applicability:

Object

This function applies to all Objects.

astExport Export an Object pointer to an outer context astExport

Description: This function exports an Object pointer from the current AST context into the context that encloses the current one. This means that the pointer will no longer be annulled when the current context is ended (with astEnd), but only when the next outer context (if any) ends.

Synopsis: `void astExport( AstObject *this )`

Parameters:

this
Object pointer to be exported.

Class Applicability:

Object
This function applies to all Objects.

Notes:

- It is only sensible to apply this function to pointers that have been created within (or exported to) the current context and have not been rendered exempt using astExempt. Applying it to an unsuitable Object pointer has no effect.

astFindFits Find a FITS card in a FitsChan by keyword astFindFits

Description: This function searches for a card in a FitsChan by keyword. The search commences at the current card (identified by the Card attribute) and ends when a card is found whose FITS keyword matches the template supplied, or when the last card in the FitsChan has been searched.

If the search is successful (i.e. a card is found which matches the template), the contents of the card are (optionally) returned and the Card attribute is adjusted to identify the card found or, if required, the one following it. If the search is not successful, the function returns zero and the Card attribute is set to the "end-of-file".

Synopsis: `int astFindFits( AstFitsChan *this, const char *name, char card[ 81 ], int inc )`

Parameters:

this
Pointer to the FitsChan.

name
Pointer to a null-terminated character string containing a template for the keyword to be found. In the simplest case, this should simply be the keyword name (the search is case insensitive and trailing spaces are ignored). However, this template may also contain "field specifiers" which are capable of matching a range of characters (see the "Keyword Templates" section for details). In this case, the first card with a keyword which matches the template will be found. To find the next FITS card regardless of its keyword, you should use the template "%f".

card
An array of at least 81 characters (to allow room for a terminating null) in which the FITS card which is found will be returned. If the search is not successful (or a NULL pointer is given), a card will not be returned.

inc

If this value is zero (and the search is successful), the FitsChan's Card attribute will be set to the index of the card that was found. If it is non-zero, however, the Card attribute will be incremented to identify the card which follows the one found.

Returned Value:

astFindFits()

One if the search was successful, otherwise zero.

Examples:

`result = astFindFits( fitschan, "%f", card, 1 );`

Returns the current card in a FitsChan and advances the Card attribute to identify the card that follows (the "%f" template matches any keyword).

`result = astFindFits( fitschan, "BITPIX", card, 1 );`

Searches a FitsChan for a FITS card with the "BITPIX" keyword and returns that card. The Card attribute is then incremented to identify the card that follows it.

`result = astFindFits( fitschan, "COMMENT", NULL, 0 );`

Sets the Card attribute of a FitsChan to identify the next COMMENT card (if any). The card itself is not returned.

`result = astFindFits( fitschan, "CRVAL%1d", card, 1 );`

Searches a FitsChan for the next card with a keyword of the form "CRVALi" (for example, any of the keywords "CRVAL1", "CRVAL2" or "CRVAL3" would be matched). The card found (if any) is returned, and the Card attribute is then incremented to identify the following card (ready to search for another keyword with the same form, perhaps).

Notes:

- The search always starts with the current card, as identified by the Card attribute. To ensure you search the entire contents of a FitsChan, you should first clear the Card attribute (using astClear). This effectively "rewinds" the FitsChan.
- If a search is unsuccessful, the Card attribute is set to the "end-of-file" (i.e. to one more than the number of cards in the FitsChan). No error occurs.
- A value of zero will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

Keyword Templates:

The templates used to match FITS keywords are normally composed of literal characters, which must match the keyword exactly (apart from case). However, a template may also contain "field specifiers" which can match a range of possible characters. This allows you to search for keywords that contain (for example) numbers, where the digits comprising the number are not known in advance.

A field specifier starts with a "%" character. This is followed by an optional single digit (0 to 9) specifying a field width. Finally, there is a single character which specifies the

type of character to be matched, as follows:

- "c": matches all upper case letters,
- "d": matches all decimal digits,
- "f": matches all characters which are permitted within a FITS keyword (upper case letters, digits, underscores and hyphens).

If the field width is omitted, the field specifier matches one or more characters. If the field width is zero, it matches zero or more characters. Otherwise, it matches exactly the number of characters specified. In addition to this:

- The template "%f" will match a blank FITS keyword consisting of 8 spaces (as well as matching all other keywords).
- A template consisting of 8 spaces will match a blank keyword (only).

For example:

- The template "BitPix" will match the keyword "BITPIX" only.
- The template "crpix%1d" will match keywords consisting of "CRPIX" followed by one decimal digit.
- The template "P%c" will match any keyword starting with "P" and followed by one or more letters.
- The template "E%0f" will match any keyword beginning with "E".
- The template "%f" will match any keyword at all (including a blank one).

---

astFindFrame Find a coordinate system with specified characteristics astFindFrame

Description: This function uses a "template" Frame to search another Frame (or FrameSet) to identify a coordinate system which has a specified set of characteristics. If a suitable coordinate system can be found, the function returns a pointer to a FrameSet which describes the required coordinate system and how to convert coordinates to and from it.

This function is provided to help answer general questions about coordinate systems, such as typically arise when coordinate information is imported into a program as part of an initially unknown dataset. For example:

- Is there a wavelength scale?
- Is there a 2-dimensional coordinate system?
- Is there a celestial coordinate system?
- Can I plot the data in ecliptic coordinates?

You can also use this function as a means of reconciling a user's preference for a particular coordinate system (for example, what type of axes to draw) with what is actually possible given the coordinate information available.

To perform a search, you supply a "target" Frame (or FrameSet) which represents the set of coordinate systems to be searched. If a basic Frame is given as the target, this set of coordinate systems consists of the one described by this Frame, plus all other "virtual" coordinate systems which can potentially be reached from it by applying built-in conversions (for example, any of the celestial coordinate conversions known to the AST library would constitute a "built-in" conversion). If a FrameSet is given as the target, the set of coordinate systems to be searched consists of the union of those represented by all the individual Frames within it.

To select from this large set of possible coordinate systems, you supply a "template" Frame which is an instance of the type of Frame you are looking for. Effectively, you then ask the function to "find a coordinate system that looks like this".

You can make your request more or less specific by setting attribute values for the template Frame. If a particular attribute is set in the template, then the function will only find coordinate systems which have exactly the same value for that attribute. If you leave a template attribute un-set, however, then the function has discretion about the value the attribute should have in any coordinate system it finds. The attribute will then take its value from one of the actual (rather than virtual) coordinate systems in the target. If the target is a FrameSet, its Current attribute will be modified to indicate which of its Frames was used for this purpose.

The result of this process is a coordinate system represented by a hybrid Frame which acquires some attributes from the template (but only if they were set) and the remainder from the target. This represents the "best compromise" between what you asked for and what was available. A Mapping is then generated which converts from the target coordinate system to this hybrid one, and the returned FrameSet encapsulates all of this information.

Synopsis: `AstFrameSet *astFindFrame( AstFrame *target, AstFrame *template, const char *domainlist )`

Parameters:

target

Pointer to the target Frame (or FrameSet).

Note that if a FrameSet is supplied (and a suitable coordinate system is found), then its Current attribute will be modified to indicate which Frame was used to obtain attribute values which were not specified by the template. This Frame will, in some sense, represent the "closest" non-virtual coordinate system to the one you requested.

template

Pointer to the template Frame, which should be an instance of the type of Frame you wish to find. If you wanted to find a Frame describing a celestial coordinate system, for example, then you might use a SkyFrame here. See the "Examples" section for more ideas.

domainlist

Pointer to a null-terminated character string containing a comma-separated list of Frame domains. This may be used to establish a priority order for the different types of coordinate system that might be found.

The function will first try to find a suitable coordinate system whose Domain attribute equals the first domain in this list. If this fails, the second domain in the list will be used, and so on, until a result is obtained. A blank domain (e.g. two consecutive commas) indicates that any coordinate system is acceptable (subject to the template) regardless of its domain.

This list is case-insensitive and all white space is ignored. If you do not wish to restrict the domain in this way, you should supply an empty string.

Class Applicability:

Frame

This function applies to all Frames.

FrameSet

If the target is a FrameSet, the possibility exists that several of the Frames within it might be matched by the template. Unless the choice is sufficiently restricted by the "domainlist" string, the sequence in which Frames are searched can then become important. In this case, the search proceeds as follows:

- Each field in the "domainlist" string is considered in turn.
- An attempt is made to match the template to each of the target's Frames in the order: (1) the current Frame, (2) the base Frame, (3) each remaining Frame in the order of being added to the target FrameSet.
- Generally, the first match found is used. However, the Mapping between the target coordinate system and the resulting Frame is also examined. Preference is given to cases where both the forward and inverse transformations are defined (as indicated by the TranForward and TranInverse attributes). If only one transformation is defined, the forward one is preferred.

- If a match is found and the domain of the resulting Frame also matches the current "domainlist" field, it is accepted. Otherwise, the next "domainlist" field is considered and the process repeated.

If a suitable coordinate system is found, the Current attribute of the target FrameSet will be modified on exit to identify the Frame whose match with the target was eventually accepted.

Returned Value:

astFindFrame()

If the search is successful, the function returns a pointer to a FrameSet which contains the Frame found and a description of how to convert to (and from) the coordinate system it represents. Otherwise, a null Object pointer (AST__NULL) is returned without error.

If a FrameSet is returned, it will contain two Frames. Frame number 1 (its base Frame) represents the target coordinate system and will be the same as the (base Frame of the) target. Frame number 2 (its current Frame) will be a Frame representing the coordinate system which the function found. The Mapping which inter-relates these two Frames will describe how to convert between their respective coordinate systems.

Note that a FrameSet may be used both as a Mapping and as a Frame. If the result is used as a Mapping (e.g. with astTran2), then it provides a means of converting coordinates from the target coordinate system into the new coordinate system that was found (and vice versa if its inverse transformation is selected). If it is used as a Frame, its attributes will describe the new coordinate system.

Examples:

`result = astFindFrame( target, astFrame( 3, "" ), "" );`

Searches for a 3-dimensional coordinate system in the target Frame (or FrameSet). No attributes have been set in the template Frame (created by astFrame), so no restriction has been placed on the required coordinate system, other than that it should have 3 dimensions. The first suitable Frame found will be returned as part of the "result" FrameSet.

`result = astFindFrame( target, astSkyFrame( "" ), "" );`

Searches for a celestial coordinate system in the target Frame (or FrameSet). The type of celestial coordinate system is unspecified, so astFindFrame will return the first one found as part of the "result" FrameSet. If the target is a FrameSet, then its Current attribute will be updated to identify the Frame that was used.

If no celestial coordinate system can be found, a value of AST__NULL will be returned without error.

`result = astFindFrame( target, astSkyFrame( "MaxAxes=100" ), "" );`

This is like the last example, except that in the event of the target being a CmpFrame, the component Frames encapsulated by the CmpFrame will be searched for a SkyFrame. If found, the returned Mapping will included a PermMap which selects the required axes from the target CmpFrame.

This is acomplished by setting the MaxAxes attribute of the template SkyFrame to a large number (larger than or equal to the number of axes in the target CmpFrame). This allows the SkyFrame to be used as a match for Frames containing from 2 to 100 axes.

`result = astFindFrame( target, astSkyFrame( "System=FK5" ), "" );`

Searches for an equatorial (FK5) coordinate system in the target. The Equinox value for the coordinate system has not been specified, so will be obtained from the target. If the target is a FrameSet, its Current attribute will be updated to indicate which SkyFrame was used to obtain this value.

`result = astFindFrame( target, astFrame( 2, "" ), "sky,pixel," );`

Searches for a 2-dimensional coordinate system in the target. Initially, a search is made for a suitable coordinate system whose Domain attribute has the value "SKY". If this search fails, a search is then made for one with the domain "PIXEL". If this also fails, then any 2-dimensional coordinate system is returned as part of the "result" FrameSet.

Only if no 2-dimensional coordinate systems can be reached by applying built-in conversions to any of the Frames in the target will a value of AST__NULL be returned.

`result = astFindFrame( target, astFrame( 1, "Domain=WAVELENGTH" ), "" );`

Searches for any 1-dimensional coordinate system in the target which has the domain "WAVELENGTH".

`result = astFindFrame( target, astFrame( 1, "" ), "wavelength" );`

This example has exactly the same effect as that above. It illustrates the equivalence of the template's Domain attribute and the fields in the "domainlist" string.

`result = astFindFrame( target, astFrame( 1, "MaxAxes=3" ), "" );`

This is a more advanced example which will search for any coordinate system in the target having 1, 2 or 3 dimensions. The Frame returned (as part of the "result" FrameSet) will always be 1-dimensional, but will be related to the coordinate system that was found by a suitable Mapping (e.g. a PermMap) which simply extracts the first axis.

If we had wanted a Frame representing the actual (1, 2 or 3-dimensional) coordinate system found, we could set the PreserveAxes attribute to a non-zero value in the template.

`result = astFindFrame( target, astSkyFrame( "Permute=0" ), "" );`

Searches for any celestial coordinate system in the target, but only finds one if its axes are in the conventional (longitude,latitude) order and have not been permuted (e.g. with astPermAxes).

Notes:

- The Mapping represented by the returned FrameSet results in alignment taking place in the coordinate system specified by the AlignSystem attribute of the "template" Frame. See the description of the AlignSystem attribute for further details.
- Beware of setting the Domain attribute of the template and then using a "domainlist" string which does not include the template's domain (or a blank field). If you do so, no coordinate system will be found.
- A null Object pointer (AST__NULL) will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

More on Using Templates:

A Frame (describing a coordinate system) will be found by this function if (a) it is "matched" by the template you supply, and (b) the value of its Domain attribute appears in the "domainlist" string (except that a blank field in this string permits any domain). A successful match by the template depends on a number of criteria, as outlined below:

- In general, a template will only match another Frame which belongs to the same class as the template, or to a derived (more specialised) class. For example, a SkyFrame template will match any other SkyFrame, but will not match a basic Frame. Conversely, a basic Frame template will match any class of Frame.
- The exception to this is that a Frame of any class can be used to match a CmpFrame, if that CmpFrame contains a Frame of the same class as the template. Note however, the MaxAxes and MinAxes attributes of the template must be set to suitable values to allow it to match the CmpFrame. That is, the MinAxes attribute must be less than or equal to the number of axes in the target, and the MaxAxes attribute must be greater than or equal to the number of axes in the target.
- If using a CmpFrame as a template frame, the MinAxes and MaxAxes for the template are determined by the MinAxes and MaxAxes values of the component Frames within the template. So if you want a template CmpFrame to be able to match Frames with different numbers of axes, then you must set the MaxAxes and/or MinAxes attributes in the component template Frames, before combining them together into the template CmpFrame.

- If a template has a value set for any of its main attributes, then it will only match Frames which have an identical value for that attribute (or which can be transformed, using a built-in conversion, so that they have the required value for that attribute). If any attribute in the template is un-set, however, then Frames are matched regardless of the value they may have for that attribute. You may therefore make a template more or less specific by choosing the attributes for which you set values. This requirement does not apply to 'descriptive' attributes such as titles, labels, symbols, etc.
- An important application of this principle involves the Domain attribute. Setting the Domain attribute of the template has the effect of restricting the search to a particular type of Frame (with the domain you specify). Conversely, if the Domain attribute is not set in the template, then the domain of the Frame found is not relevant, so all Frames are searched. Note that the "domainlist" string provides an alternative way of restricting the search in the same manner, but is a more convenient interface if you wish to search automatically for another domain if the first search fails.
- Normally, a template will only match a Frame which has the same number of axes as itself. However, for some classes of template, this default behaviour may be changed by means of the MinAxes, MaxAxes and MatchEnd attributes. In addition, the behaviour of a template may be influenced by its Permute and PreserveAxes attributes, which control whether it matches Frames whose axes have been permuted, and whether this permutation is retained in the Frame which is returned (as opposed to returning the axes in the order specified in the template, which is the default behaviour). You should consult the descriptions of these attributes for details of this more advanced use of templates.

---

astFitsChan Create a FitsChan astFitsChan

Description: This function creates a new FitsChan and optionally initialises its attributes.

A FitsChan is a specialised form of Channel which supports I/O operations involving the use of FITS (Flexible Image Transport System) header cards. Writing an Object to a FitsChan (using astWrite) will, if the Object is suitable, generate a description of that Object composed of FITS header cards, and reading from a FitsChan will create a new Object from its FITS header card description.

While a FitsChan is active, it represents a buffer which may contain zero or more 80-character "header cards" conforming to FITS conventions. Any sequence of FITS-conforming header cards may be stored, apart from the "END" card whose existence is merely implied. The cards may be accessed in any order by using the FitsChan's integer Card attribute, which identifies a "current" card, to which subsequent operations apply. Searches based on keyword may be performed (using astFindFits), new cards may be inserted (astPutFits, astPutCards, astSetFits<X>) and existing ones may be deleted (astDelFits) or changed (astSetFits<X>).

When you create a FitsChan, you have the option of specifying "source" and "sink" functions which connect it to external data stores by reading and writing FITS header cards. If you provide a source function, it is used to fill the FitsChan with header cards when it is accessed for the first time. If you do not provide a source function, the FitsChan remains empty until you explicitly enter data into it (e.g. using astPutFits, astPutCards, astWrite or by using the SourceFile attribute to specifying a text file from which headers should be read). When the FitsChan is deleted, any remaining header cards in the FitsChan can be saved in either of two ways: 1) by specifying a value for the SinkFile attribute (the name of a text file to which header cards should be written), or 2) by providing a sink function (used to to deliver header cards to an external data store). If you do not provide a sink function or a value for SinkFile, any header cards remaining when the FitsChan is deleted will be lost, so you should arrange to extract them first if necessary (e.g. using astFindFits or astRead).

Coordinate system information may be described using FITS header cards using several different conventions, termed "encodings". When an AST Object is written to (or read from) a FitsChan, the value of the FitsChan's Encoding attribute determines how the Object is converted to (or from) a description involving FITS header cards. In general, different encodings will result in different sets of header cards to describe the same Object. Examples of encodings include the DSS encoding (based on conventions used by the STScI Digitised Sky Survey data), the FITS-WCS encoding (based on a proposed FITS standard) and the NATIVE encoding (a near loss-less way of storing AST Objects in FITS headers).

The available encodings differ in the range of Objects they can represent, in the number of Object descriptions that can coexist in the same FitsChan, and in their accessibility to other (external) astronomy applications (see the Encoding attribute for details). Encodings are not necessarily mutually exclusive and it may sometimes be possible to describe the same Object in several ways within a particular set of FITS header cards by using several different encodings.

The detailed behaviour of astRead and astWrite, when used with a FitsChan, depends on the encoding in use. In general, however, all use of astRead is destructive, so that FITS header cards are consumed in the process of reading an Object, and are removed from the FitsChan (this deletion can be prevented for specific cards by calling the astRetainFits function).

If the encoding in use allows only a single Object description to be stored in a FitsChan (e.g. the DSS, FITS-WCS and FITS-IRAF encodings), then write operations using astWrite will over-write any existing Object description using that encoding. Otherwise (e.g. the NATIVE encoding), multiple Object descriptions are written sequentially and may later be read back in the same sequence.

Synopsis: `AstFitsChan *astFitsChan( const char *(* source)( void ), void (* sink)( const char * ), const char *options, ... )`

Parameters:

source

Pointer to a source function which takes no arguments and returns a pointer to a null-terminated string. This function will be used by the FitsChan to obtain input FITS header cards. On each invocation, it should read the next input card from some external source (such as a FITS file), and return a pointer to the (null-terminated) contents of the card. It should return a NULL pointer when there are no more cards to be read.

If "source" is NULL, the FitsChan will remain empty until cards are explicitly stored in it (e.g. using astPutCards, astPutFits or via the SourceFile attribute).

sink

Pointer to a sink function that takes a pointer to a null-terminated string as an argument and returns void. If no value has been set for the SinkFile attribute, this function will be used by the FitsChan to deliver any FITS header cards it contains when it is finally deleted. On each invocation, it should deliver the contents of the character string passed to it as a FITS header card to some external data store (such as a FITS file).

If "sink" is NULL, and no value has been set for the SinkFile attribute, the contents of the FitsChan will be lost when it is deleted.

options

Pointer to a null-terminated string containing an optional comma-separated list of attribute assignments to be used for initialising the new FitsChan. The syntax used is identical to that for the astSet function and may include "printf" format specifiers identified by "%" symbols in the normal way.

...

If the "options" string contains "%" format specifiers, then an optional list of additional arguments may follow it in order to supply values to be substituted for these specifiers. The rules for supplying these are identical to those for the astSet function (and for the C "printf" function).

Returned Value:

astFitsChan()

A pointer to the new FitsChan.

Notes:

- No FITS "END" card will be written via the sink function. You should add this card yourself after the FitsChan has been deleted.
- A null Object pointer (AST__NULL) will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

Status Handling:

The protected interface to this function includes an extra parameter at the end of the parameter list descirbed above. This parameter is a pointer to the integer inherited status variable: "int *status".

---

astFitsTable Create a FitsTable astFitsTable

Description: This function creates a new FitsTable and optionally initialises its attributes.

The FitsTable class is a representation of a FITS binary table. It inherits from the Table class. The parent Table is used to hold the binary data of the main table, and a FitsChan is used to hold the FITS header. Note, there is no provision for binary data following the main table (such data is referred to as a "heap" in the FITS standard).

Note - it is not recommended to use the FitsTable class to store very large tables.

Synopsis: `AstFitsTable *astFitsTable( AstFitsChan *header, const char *options, ... )`

Parameters:

header

Pointer to an optional FitsChan containing headers to be stored in the FitsTable. NULL may be supplied if the new FitsTable is to be left empty. If supplied, and if the headers describe columns of a FITS binary table, then equivalent (empty) columns are added to the FitsTable. Each column has the same index in the FitsTable that it has in the supplied header.

options

Pointer to a null-terminated string containing an optional comma-separated list of attribute assignments to be used for initialising the new FitsTable. The syntax used is identical to that for the astSet function and may include "printf" format specifiers identified by "%" symbols in the normal way.

...

If the "options" string contains "%" format specifiers, then an optional list of additional arguments may follow it in order to supply values to be substituted for these specifiers. The rules for supplying these are identical to those for the astSet function (and for the C "printf" function).

Returned Value:

astFitsTable()

A pointer to the new FitsTable.

Notes:

- A null Object pointer (AST__NULL) will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

Status Handling:

The protected interface to this function includes an extra parameter at the end of the parameter list described above. This parameter is a pointer to the integer inherited status variable: "int *status".

---

astFluxFrame Create a FluxFrame astFluxFrame

Description: This function creates a new FluxFrame and optionally initialises its attributes.

A FluxFrame is a specialised form of one-dimensional Frame which represents various systems used to represent the signal level in an observation. The particular coordinate system to be used is specified by setting the FluxFrame's System attribute qualified, as necessary, by other attributes such as the units, etc (see the description of the System attribute for details).

All flux values are assumed to be measured at the same frequency or wavelength (as given by the SpecVal attribute). Thus this class is more appropriate for use with images rather than spectra.

Synopsis: `AstFluxFrame *astFluxFrame( double specval, AstSpecFrame *specfrm, const char *options, ... )`

Parameters:

specval

The spectral value to which the flux values refer, given in the spectral coordinate system specified by "specfrm". The value supplied for the "specval" parameter becomes the default value for the SpecVal attribute. A value of AST__BAD may be supplied if the spectral position is unknown, but this may result in it not being possible for the astConvert function to determine a Mapping between the new FluxFrame and some other FluxFrame.

specfrm

A pointer to a SpecFrame describing the spectral coordinate system in which the "specval" parameter is given. A deep copy of this object is taken, so any subsequent changes to the SpecFrame using the supplied pointer will have no effect on the new FluxFrame. A NULL pointer can be supplied if AST__BAD is supplied for "specval".

options

Pointer to a null-terminated string containing an optional comma-separated list of attribute assignments to be used for initialising the new FluxFrame. The syntax used is identical to that for the astSet function and may include "printf" format specifiers identified by "%" symbols in the normal way. If no initialisation is required, a zero-length string may be supplied.

...

If the "options" string contains "%" format specifiers, then an optional list of additional arguments may follow it in order to supply values to be substituted for these specifiers. The rules for supplying these are identical to those for the astSet function (and for the C "printf" function).

Returned Value:

astFluxFrame()

A pointer to the new FluxFrame.

Notes:

- When conversion between two FluxFrames is requested (as when supplying FluxFrames to astConvert), account will be taken of the nature of the flux coordinate systems they represent, together with any qualifying attribute values, including the AlignSystem attribute. The results will therefore fully reflect the relationship between positions measured in the two systems. In addition, any difference in the Unit attributes of the two systems will also be taken into account.

- A null Object pointer (AST__NULL) will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

---

astFormat Format a coordinate value for a Frame axis astFormat

Description: This function returns a pointer to a string containing the formatted (character) version of a coordinate value for a Frame axis. The formatting applied is determined by the Frame's attributes and, in particular, by any Format attribute string that has been set for the axis. A suitable default format (based on the Digits attribute value) will be applied if necessary.

Synopsis: `const char *astFormat( AstFrame *this, int axis, double value )`

Parameters:

this

Pointer to the Frame.

axis

The number of the Frame axis for which formatting is to be performed (axis numbering starts at 1 for the first axis).

value

The coordinate value to be formatted.

Returned Value:

astFormat()

A pointer to a null-terminated string containing the formatted value.

Notes:

- The returned pointer is guaranteed to remain valid and the string to which it points will not be over-written for a total of 50 successive invocations of this function. After this, the memory containing the string may be re-used, so a copy of the string should be made if it is needed for longer than this.
- A formatted value may be converted back into a numerical (double) value using astUnformat.
- A NULL pointer will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

---

astFrame Create a Frame astFrame

Description: This function creates a new Frame and optionally initialises its attributes.

A Frame is used to represent a coordinate system. It does this in rather the same way that a frame around a graph describes the coordinate space in which data are plotted. Consequently, a Frame has a Title (string) attribute, which describes the coordinate space, and contains axes which in turn hold information such as Label and Units strings which are used for labelling (e.g.) graphical output. In general, however, the number of axes is not restricted to two.

Functions are available for converting Frame coordinate values into a form suitable for display, and also for calculating distances and offsets between positions within the Frame.

Frames may also contain knowledge of how to transform to and from related coordinate systems.

Synopsis: `AstFrame *astFrame( int naxes, const char *options, ... )`

Parameters:

naxes

The number of Frame axes (i.e. the number of dimensions of the coordinate space which the Frame describes).

options

Pointer to a null-terminated string containing an optional comma-separated list of attribute assignments to be used for initialising the new Frame. The syntax used is identical to that for the astSet function and may include "printf" format specifiers identified by "%" symbols in the normal way. If no initialisation is required, a zero-length string may be supplied.

...

If the "options" string contains "%" format specifiers, then an optional list of additional arguments may follow it in order to supply values to be substituted for these specifiers. The rules for supplying these are identical to those for the astSet function (and for the C "printf" function).

Returned Value:

astFrame()

A pointer to the new Frame.

Examples:

`frame = astFrame( 2, "Title=Energy Spectrum:  Plot %d", n );`

Creates a new 2-dimensional Frame and initialises its Title attribute to the string "Energy Spectrum: Plot <n>", where <n> takes the value of the int variable "n".

`frame = astFrame( 2, "Label(1)=Energy, Label(2)=Response" );`

Creates a new 2-dimensional Frame and initialises its axis Label attributes to suitable string values.

Notes:

- A null Object pointer (AST__NULL) will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

---

astFrameSet Create a FrameSet astFrameSet

Description: This function creates a new FrameSet and optionally initialises its attributes.

A FrameSet consists of a set of one or more Frames (which describe coordinate systems), connected together by Mappings (which describe how the coordinate systems are inter-related). A FrameSet makes it possible to obtain a Mapping between any pair of these Frames (i.e. to convert between any of the coordinate systems which it describes). The individual Frames are identified within the FrameSet by an integer index, with Frames being numbered consecutively from one as they are added to the FrameSet.

Every FrameSet has a "base" Frame and a "current" Frame (which are allowed to be the same). Any of the Frames may be nominated to hold these positions, and the choice is determined by the values of the FrameSet's Base and Current attributes, which hold the indices of the relevant Frames. By default, the first Frame added to a FrameSet is its base Frame, and the last one added is its current Frame.

The base Frame describes the "native" coordinate system of whatever the FrameSet is used to calibrate (e.g. the pixel coordinates of an image) and the current Frame describes the "apparent" coordinate system in which it should be viewed (e.g. displayed, etc.). Any further Frames represent a library of alternative coordinate systems, which may be selected by making them current.

When a FrameSet is used in a context that requires a Frame, (e.g. obtaining its Title value, or number of axes), the current Frame is used. A FrameSet may therefore be used in place of its current Frame in most situations.

When a FrameSet is used in a context that requires a Mapping, the Mapping used is the one between its base Frame and its current Frame. Thus, a FrameSet may be used to convert "native" coordinates into "apparent" ones, and vice versa. Like any Mapping, a FrameSet may also be inverted (see astInvert), which has the effect of interchanging its base and current Frames and hence of reversing the Mapping between them.

Regions may be added into a FrameSet (since a Region is a type of Frame), either explicitly or as components within CmpFrames. In this case the Mapping between a pair of Frames within a FrameSet will include the effects of the clipping produced by any Regions included in the path between the Frames.

Synopsis: `AstFrameSet *astFrameSet( AstFrame *frame, const char *options, ... )`

Parameters:

frame

Pointer to the first Frame to be inserted into the FrameSet. This initially becomes both the base and the current Frame. (Further Frames may be added using the astAddFrame function.)

options

Pointer to a null-terminated string containing an optional comma-separated list of attribute assignments to be used for initialising the new FrameSet. The syntax used is identical to that for the astSet function and may include "printf" format specifiers identified by "%" symbols in the normal way. If no initialisation is required, a zero-length string may be supplied.

...

If the "options" string contains "%" format specifiers, then an optional list of additional arguments may follow it in order to supply values to be substituted for these specifiers. The rules for supplying these are identical to those for the astSet function (and for the C "printf" function).

Returned Value:

astFrameSet()

A pointer to the new FrameSet.

Notes:

- If a pointer to an existing FrameSet is given for the "frame" parameter, then the new FrameSet will (as a special case) be initialised to contain the same Frames and Mappings, and to have the same attribute values, as the one supplied. This process is similar to making a copy of a FrameSet (see astCopy), except that the Frames and Mappings contained in the original are not themselves copied, but are shared by both FrameSets.
- A null Object pointer (AST__NULL) will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

---

astFromString Re-create an Object from an in-memory serialisation astFromString

Description: This function returns a pointer to a new Object created from the supplied text string, which should have been created by astToString.

Synopsis: `AstObject *astFromString( const char *string )`

Parameters:

string

Pointer to a text string holding an Object serialisation created previously by astToString.

Returned Value:

astFromString()
Pointer to a new Object created from the supplied serialisation, or NULL if the serialisation was invalid, or an error occurred.

---

astGenCurve Draw a generalized curve astGenCurve

Description: This function draws a general user-defined curve defined by the supplied Mapping. Note that the curve is transformed into graphical coordinate space for plotting, so that a straight line in physical coordinates may result in a curved line being drawn if the Mapping involved is non-linear. Any discontinuities in the Mapping between physical and graphical coordinates are catered for, as is any clipping established using astClip.

If you need to draw simple straight lines (geodesics), astCurve or astPolyCurve will usually be easier to use and faster.

Synopsis: `void astGenCurve( AstPlot *this, astMapping *map )`

Parameters:

this
Pointer to the Plot.

map
Pointer to a Mapping. This Mapping should have 1 input coordinate representing offset along the required curve, normalized so that the start of the curve is at offset 0.0, and the end of the curve is at offset 1.0. Note, this offset does not need to be linearly related to distance along the curve. The number of output coordinates should equal the number of axes in the current Frame of the Plot. The Mapping should map a specified offset along the curve, into the corresponding coordinates in the current Frame of the Plot. The inverse transformation need not be defined.

Notes:

- An error results if the base Frame of the Plot is not 2-dimensional.
- An error also results if the transformation between the current and base Frames of the Plot is not defined (i.e. the Plot's TranInverse attribute is zero).

---

astGet<X> Get an attribute value for an Object astGet<X>

Description: This is a family of functions which return a specified attribute value for an Object using one of several different data types. The type is selected by replacing <X> in the function name by C, D, F, I or L, to obtain a result in const char∗ (i.e. string), double, float, int, or long format, respectively.

If possible, the attribute value is converted to the type you request. If conversion is not possible, an error will result.

Synopsis: `<X>type astGet<X>( AstObject *this, const char *attrib )`

Parameters:

this
Pointer to the Object.

attrib
Pointer to a null-terminated string containing the name of the attribute whose value is required.

Class Applicability:

Object
These functions apply to all Objects.

Returned Value:

astGet<X>()
The attribute value, in the data type corresponding to <X> (or, in the case of astGetC, a pointer to a constant null-terminated character string containing this value).

Examples:

`printf( "RefCount = %d\n", astGetI( z, "RefCount" ) );`
Prints the RefCount attribute value for Object "z" as an int.

`title = astGetC( axis, "Title" );`
Obtains a pointer to a null-terminated character string containing the Title attribute of Object "axis".

Notes:

- Attribute names are not case sensitive and may be surrounded by white space.
- An appropriate "null" value will be returned if this function is invoked with the AST error status set, or if it should fail for any reason. This null value is zero for numeric values and NULL for pointer values.
- The pointer returned by astGetC is guaranteed to remain valid and the string to which it points will not be over-written for a total of 50 successive invocations of this function. After this, the memory containing the string may be re-used, so a copy of the string should be made if it is needed for longer than this.

---

astGetActiveUnit Determines how the Unit attribute will be used astGetActiveUnit

Description: This function returns the current value of the ActiveUnit flag for a Frame. See the description of the astSetActiveUnit function for a description of the ActiveUnit flag.

Synopsis: `int astGetActiveUnit( AstFrame *this )`

Parameters:

this
Pointer to the Frame.

Returned Value:

astGetActiveUnit
The current value of the ActiveUnit flag.

Notes:

- A zero value will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

---

astGetColumnData Retrieve all the data values stored in a column astGetColumnData

Description: This function copies all data values from a named column into a supplied buffer

Synopsis: `void astGetColumnData( AstFitsTable *this, const char *column, float fnull, double dnull, size_t mxsize, void *coldata, int *nelem )`

Parameters:

this
Pointer to the FitsTable.

column
The character string holding the name of the column. Trailing spaces are ignored.

fnull
The value to return in "coldata" for any cells for which no value has been stored in the FitsTable. Ignored if the column's data type is not AST__FLOATTYPE. Supplying AST__NANF will cause a single precision IEEE NaN value to be used.

dnull
The value to return in "coldata" for any cells for which no value has been stored in the FitsTable. Ignored if the column's data type is not AST__DOUBLETYPE. Supplying AST__NAN will cause a double precision IEEE NaN value to be used.

mxsize
The size of the "coldata" array, in bytes. The amount of memory needed to hold the data from a column may be determined using astColumnSize. If the supplied array is too small to hold all the column data, trailing column values will be omitted from the returned array, but no error will be reported.

coldata
A pointer to an area of memory in which to return the data values currently stored in the column. The values are stored in row order. If the column holds non-scalar values, the elements of each value are stored in "Fortran" order. No data type conversion is performed - the data type of each returned value is the data type associated with the column when the column was added to the table. If the column holds strings, the returned strings will be null terminated. Any excess room at the end of the array will be left unchanged.

nelem
The number of elements returned in the "coldata" array. This is the product of the number of rows returned and the number of elements in each column value.

Notes:

- The "fnull" and "dnull" parameters specify the value to be returned for any empty cells within columns holding floating point values. For columns holding integer values, the value returned for empty cells is the value returned by the astColumNull function. For columns holding string values, the ASCII NULL character is returned for empty cells.

---

astGetFits<X> Get a named keyword value from a FitsChan astGetFits<X>

Description: This is a family of functions which gets a value for a named keyword, or the value of the current card, from a FitsChan using one of several different data types. The data type of the returned value is selected by replacing <X> in the function name by one of the following strings representing the recognised FITS data types:

- CF - Complex floating point values.
- CI - Complex integer values.
- F - Floating point values.
- I - Integer values.
- L - Logical (i.e. boolean) values.
- S - String values.
- CN - A "CONTINUE" value, these are treated like string values, but are encoded without an equals sign.

The data type of the "value" parameter

depends on <X> as follows:

- CF - "double *" (a pointer to a 2 element array to hold the real and imaginary parts of the complex value).
- CI - "int *" (a pointer to a 2 element array to hold the real and imaginary parts of the complex value).
- F - "double *".
- I - "int *".
- L - "int *".
- S - "char **" (a pointer to a static "char" array is returned at the location given by the "value" parameter, Note, the stored string may change on subsequent invocations of astGetFitsS so a permanent copy should be taken of the string if necessary).
- CN - Like"S".

Synopsis: `int astGetFits<X>( AstFitsChan *this, const char *name, <X>type *value )`

Parameters:

this

Pointer to the FitsChan.

name

Pointer to a null-terminated character string containing the FITS keyword name. This may be a complete FITS header card, in which case the keyword to use is extracted from it. No more than 80 characters are read from this string. If NULL is supplied, the value of the current card is returned.

value

A pointer to a buffer to receive the keyword value. The data type depends on <X> as described above. The conents of the buffer on entry are left unchanged if the keyword is not found.

Returned Value:

astGetFits<X><X>()

A value of zero is returned if the keyword was not found in the FitsChan (no error is reported). Otherwise, a value of one is returned.

Notes:

- If a name is supplied, the card following the current card is checked first. If this is not the required card, then the rest of the FitsChan is searched, starting with the first card added to the FitsChan. Therefore cards should be accessed in the order they are stored in the FitsChan (if possible) as this will minimise the time spent searching for cards.
- If the requested card is found, it becomes the current card, otherwise the current card is left pointing at the "end-of-file".
- If the stored keyword value is not of the requested type, it is converted into the requested type.
- If the keyword is found in the FitsChan, but has no associated value, an error is reported. If necessary, the astTestFits function can be used to determine if the keyword has a defined value in the FitsChan prior to calling this function.
- An error will be reported if the keyword name does not conform to FITS requirements.
- Zero
- .FALSE. is returned as the function value if an error has already occurred, or if this function should fail for any reason.
- The FITS standard says that string keyword values should be padded with trailing spaces if they are shorter than 8 characters. For this reason, trailing spaces are removed from the string returned by astGetFitsS if the original string (including any trailing spaces) contains 8 or fewer characters. Trailing spaces are not removed from longer strings.

---

astGetFrame Obtain a pointer to a specified Frame in a FrameSet astGetFrame

Description: This function returns a pointer to a specified Frame in a FrameSet.

Synopsis: `AstFrame *astGetFrame( AstFrameSet *this, int iframe )`

Parameters:

this
Pointer to the FrameSet.

iframe
The index of the required Frame within the FrameSet. This value should lie in the range from 1 to the number of Frames in the FrameSet (as given by its Nframe attribute).

Returned Value:

astGetFrame()
A pointer to the requested Frame.

Notes:

- A value of AST__BASE or AST__CURRENT may be given for the "iframe" parameter to specify the base Frame or the current Frame respectively.
- This function increments the RefCount attribute of the selected Frame by one.
- A null Object pointer (AST__NULL) will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

---

astGetGrfContext Return the KeyMap that describes a Plot's graphics context astGetGrfContext

Description: This function returns a reference to a KeyMap that will be passed to any drawing functions registered using astGrfSet. This KeyMap can be used by an application to pass information to the drawing functions about the context in which they are being called. The contents of the KeyMap are never accessed byt the Plot class itself.

Synopsis: `AstKeyMap *astGetGrfContext( AstPlot *this )`

Parameters:

this
Pointer to the Plot.

Returned Value:

astGetGrfContext()
A pointer to the graphics context KeyMap. The returned pointer should be annulled when it is no longer needed.

---

astGetMapping Obtain a Mapping that converts between two Frames in a FrameSet astGetMapping

Description: This function returns a pointer to a Mapping that will convert coordinates between the coordinate systems represented by two Frames in a FrameSet.

Synopsis: `AstMapping *astGetMapping( AstFrameSet *this, int iframe1, int iframe2 )`

Parameters:

this
Pointer to the FrameSet.

iframe1
The index of the first Frame in the FrameSet. This Frame describes the coordinate system for the "input" end of the Mapping.

iframe2
The index of the second Frame in the FrameSet. This Frame describes the coordinate system for the "output" end of the Mapping.

Returned Value:

astGetMapping()
Pointer to a Mapping whose forward transformation converts coordinates from the first coordinate system to the second one, and whose inverse transformation converts coordinates in the opposite direction.

Notes:

- The returned Mapping will include the clipping effect of any Regions which occur on the path between the two supplied Frames (this includes the two supplied Frames themselves).
- The values given for the "iframe1" and "iframe2" parameters should lie in the range from 1 to the number of Frames in the FrameSet (as given by its Nframe attribute). A value of AST__BASE or AST__CURRENT may also be given to identify the FrameSet's base Frame or current Frame respectively. It is permissible for both these parameters to have the same value, in which case a unit Mapping (UnitMap) is returned.

- It should always be possible to generate the Mapping requested, but this does necessarily guarantee that it will be able to perform the required coordinate conversion. If necessary, the TranForward and TranInverse attributes of the returned Mapping should be inspected to determine if the required transformation is available.
- A null Object pointer (AST__NULL) will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

---

astGetRefPos Return the reference position in a specified celestial coordinate system astGetRefPos

Description: This function returns the reference position (specified by attributes RefRA and RefDec) converted to the celestial coordinate system represented by a supplied SkyFrame. The celestial longitude and latitude values are returned in radians.

Synopsis: `void astGetRefPos( AstSpecFrame *this, AstSkyFrame *frm, double *lon, double *lat )`

Parameters:

this
Pointer to the SpecFrame.

frm
Pointer to the SkyFrame which defines the required celestial coordinate system. If NULL is supplied, then the longitude and latitude values are returned as FK5 J2000 RA and Dec values.

lon
A pointer to a double in which to store the longitude of the reference point, in the coordinate system represented by the supplied SkyFrame (radians).

lat
A pointer to a double in which to store the latitude of the reference point, in the coordinate system represented by the supplied SkyFrame (radians).

Notes:

- Values of AST__BAD will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

---

astGetRegionBounds Returns the bounding box of Region astGetRegionBounds

Description: This function returns the upper and lower limits of a box which just encompasses the supplied Region. The limits are returned as axis values within the Frame represented by the Region. The value of the Negated attribute is ignored (i.e. it is assumed that the Region has not been negated).

Synopsis: `void astGetRegionBounds( AstRegion *this, double *lbnd, double *ubnd )`

Parameters:

this
Pointer to the Region.

lbnd

Pointer to an array in which to return the lower axis bounds covered by the Region. It should have at least as many elements as there are axes in the Region. If an axis has no lower limit, the returned value will be the largest possible negative value.

ubnd

Pointer to an array in which to return the upper axis bounds covered by the Region. It should have at least as many elements as there are axes in the Region. If an axis has no upper limit, the returned value will be the largest possible positive value.

Notes:

- The value of the Negated attribute is ignored (i.e. it is assumed that the Region has not been negated).
- If an axis has no extent on an axis then the lower limit will be returned larger than the upper limit. Note, this is different to an axis which has a constant value (in which case both lower and upper limit will be returned set to the constant value).
- If the bounds on an axis cannot be determined, AST__BAD is returned for both upper and lower bounds

---

astGetRegionFrame Obtain a pointer to the encapsulated Frame within a Region astGetRegionFrame

Description: This function returns a pointer to the Frame represented by a Region.

Synopsis: `AstFrame *astGetRegionFrame( AstRegion *this )`

Parameters:

this

Pointer to the Region.

Returned Value:

astGetRegionFrame()

A pointer to a deep copy of the Frame represented by the Region. Using this pointer to modify the Frame will have no effect on the Region. To modify the Region, use the Region pointer directly.

Notes:

- A null Object pointer (AST__NULL) will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

---

astGetRegionMesh Return a mesh of points covering the surface or volume of a Region astGetRegionMesh

Description: This function returns the axis values at a mesh of points either covering the surface (i.e. boundary) of the supplied Region, or filling the interior (i.e. volume) of the Region. The number of points in the mesh is approximately equal to the MeshSize attribute.

Synopsis: `void astGetRegionMesh( AstRegion *this, int surface, int maxpoint, int maxcoord, int *npoint, double *points )`

Parameters:

this

Pointer to the Region.

surface

If non-zero, the returned points will cover the surface or the Region. Otherwise, they will fill the interior of the Region.

maxpoint

If zero, the number of points in the mesh is returned in "*npoint", but no axis values are returned and all other parameters are ignored. If not zero, the supplied value should be the length of the second dimension of the "points" array. An error is reported if the number of points in the mesh exceeds this number.

maxcoord

The length of the first dimension of the "points" array. An error is reported if the number of axes in the supplied Region exceeds this number.

npoint

A pointer to an integer in which to return the number of points in the returned mesh.

points

The address of the first element in a 2-dimensional array of shape "[maxcoord][maxpoint]", in which to return the coordinate values at the mesh positions. These are stored such that the value of coordinate number "coord" for point number "point" is found in element "points[coord][point]".

Notes:

- An error is reported if the Region is unbounded.
- If the coordinate system represented by the Region has been changed since it was first created, the returned axis values refer to the new (changed) coordinate system, rather than the original coordinate system. Note however that if the transformation from original to new coordinate system is non-linear, the shape within the new coordinate system may be distorted, and so may not match that implied by the name of the Region subclass (Circle, Box, etc).

---

astGetRegionPoints Returns the positions that define the given Region astGetRegionPoints

Description: This function returns the axis values at the points that define the supplied Region. The particular meaning of these points will depend on the type of class supplied, as listed below under "Applicability:".

Synopsis: `void astGetRegionPoints( AstRegion *this, int maxpoint, int maxcoord, int *npoint, double *points )`

Parameters:

this

Pointer to the Region.

maxpoint

If zero, the number of points needed to define the Region is returned in "*npoint", but no axis values are returned and all other parameters are ignored. If not zero, the supplied value should be the length of the second dimension of the "points" array. An error is reported if the number of points needed to define the Region exceeds this number.

maxcoord

The length of the first dimension of the "points" array. An error is reported if the number of axes in the supplied Region exceeds this number.

npoint

A pointer to an integer in which to return the number of points defining the Region.

points

The address of the first element in a 2-dimensional array of shape "[maxcoord][maxpoint]", in which to return the coordinate values at the positions that define the Region. These are stored such that the value of coordinate number "coord" for point number "point" is found in element "points[coord][point]".

Class Applicability:

Region

All Regions have this attribute.

Box

The first returned position is the Box centre, and the second is a Box corner.

Circle

The first returned position is the Circle centre, and the second is a point on the circumference.

CmpRegion

Returns a value of zero for "∗npoint" and leaves the supplied array contents unchanged. To find the points defining a CmpRegion, use this method on the component Regions, which can be accessed by invoking astDecompose on the CmpRegion.

Ellipse

The first returned position is the Ellipse centre. The second is the end of one of the axes of the ellipse. The third is some other point on the circumference of the ellipse, distinct from the second point.

Interval

The first point corresponds to the lower bounds position, and the second point corresponds to the upper bounds position. These are reversed to indicate an extcluded interval rather than an included interval. See the Interval constructor for more information.

NullRegion

Returns a value of zero for "∗npoint" and leaves the supplied array contents unchanged.

PointList

The positions returned are those that were supplied when the PointList was constructed.

Polygon

The positions returned are the vertex positions that were supplied when the Polygon was constructed.

Prism

Returns a value of zero for "∗npoint" and leaves the supplied array contents unchanged. To find the points defining a Prism, use this method on the component Regions, which can be accessed by invoking astDecompose on the CmpRegion.

Notes:

- If the coordinate system represented by the Region has been changed since it was first created, the returned axis values refer to the new (changed) coordinate system, rather than the original coordinate system. Note however that if the transformation from original to new coordinate system is non-linear, the shape within the new coordinate system may be distorted, and so may not match that implied by the name of the Region subclass (Circle, Box, etc).

---

astGetStcCoord Return information about an AstroCoords element stored in an Stc astGetStcCoord

Description: When any sub-class of Stc is created, the constructor function allows one or more AstroCoords elements to be stored within the Stc. This function allows any one of these AstroCoords elements to be retrieved. The format of the returned information is the same as that used to pass the original information to the Stc constructor. That is, the information is returned in a KeyMap structure containing elements with one or more of the keys given by symbolic constants AST__STCNAME, AST__STCVALUE, AST__STCERROR, AST__STCRES, AST__STCSIZE and AST__STCPIXSZ.

If the coordinate system represented by the Stc has been changed since it was created (for instance, by changing its System attribute), then the sizes and positions in the returned KeyMap will reflect the change in coordinate system.

Synopsis: `AstKeyMap *astGetStcCoord( AstStc *this, int icoord )`

Parameters:

this
Pointer to the Stc.

icoord
The index of the AstroCoords element required. The first has index one. The number of AstroCoords elements in the Stc can be found using function astGetStcNcoord.

Returned Value:

astGetStcCoord()
A pointer to a new KeyMap containing the required information.

Notes:

- A null Object pointer (AST__NULL) will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

---

astGetStcNCoord Return the number of AstroCoords elements stored in an Stc astGetStcNCoord

Description: This function returns the number of AstroCoords elements stored in an Stc.

Synopsis: `int astGetStcNCoord( AstStc *this )`

Parameters:

this
Pointer to the Stc.

Returned Value:

astGetStcNCoord()
The number of AstroCoords elements stored in the Stc.

Notes:

- Zero will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

astGetStcRegion Obtain a copy of the encapsulated Region within a Stc astGetStcRegion

Description: This function returns a pointer to a deep copy of the Region supplied when the Stc was created.

Synopsis: `AstRegion *astGetStcRegion( AstStc *this )`

Parameters:

this
Pointer to the Stc.

Returned Value:

astGetStcRegion()
A pointer to a deep copy of the Region encapsulated within the supplied Stc.

Notes:

- A null Object pointer (AST__NULL) will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

astGetTableHeader Get the FITS headers from a FitsTable astGetTableHeader

Description: This function returns a pointer to a FitsChan holding copies of the FITS headers associated with a FitsTable.

Synopsis: `AstFitsChan *astGetTableHeader( AstFitsTable *this )`

Parameters:

this
Pointer to the FitsTable.

Returned Value:

astGetTableHeader()
A pointer to a deep copy of the FitsChan stored within the FitsTable.

Notes:

- The returned pointer should be annulled using astAnnul when it is no longer needed.
- Changing the contents of the returned FitsChan will have no effect on the FitsTable. To modify the FitsTable, the modified FitsChan must be stored in the FitsTable using astPutTableHeader.

---

astGetTables Retrieve any FitsTables currently in a FitsChan astGetTables

Description: If the supplied FitsChan currently contains any tables, then this function returns a pointer to a KeyMap. Each entry in the KeyMap is a pointer to a FitsTable holding the data for a FITS binary table. The key used to access each entry is the FITS extension name in which the table should be stored.

Tables can be present in a FitsChan as a result either of using the astPutTable (or astPutTables) method to store existing tables in the FitsChan, or of using the astWrite method to write a FrameSet to the FitsChan. For the later case, if the FitsChan "TabOK" attribute is positive and the FrameSet requires a look-up table to describe one or more axes, then the "-TAB" algorithm code described in FITS-WCS paper III is used and the table values are stored in the FitsChan in the form of a FitsTable object (see the documentation for the "TabOK" attribute).

Synopsis: `AstKeyMap *astGetTables( AstFitsChan *this )`

Parameters:

this
Pointer to the FitsChan.

Returned Value:

astGetTables()
A pointer to a deep copy of the KeyMap holding the tables currently in the FitsChan, or NULL if the FitsChan does not contain any tables. The returned pointer should be annulled using astAnnul when no longer needed.

Notes:

- A null Object pointer (AST__NULL) will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

---

astGetUnc Obtain uncertainty information from a Region astGetUnc

Description: This function returns a Region which represents the uncertainty associated with positions within the supplied Region. See astSetUnc for more information about Region uncertainties and their use.

Synopsis: `AstRegion *astGetUnc( AstRegion *this, int def )`

Parameters:

this
Pointer to the Region.

def
Controls what is returned if no uncertainty information has been associated explicitly with the supplied Region. If a non-zero value is supplied, then the default uncertainty Region used internally within AST is returned (see "Applicability" below). If zero is supplied, then NULL will be returned (without error).

Class Applicability:

CmpRegion

The default uncertainty for a CmpRegion is taken from one of the two component Regions. If the first component Region has a non-default uncertainty, then it is used as the default uncertainty for the parent CmpRegion. Otherwise, if the second component Region has a non-default uncertainty, then it is used as the default uncertainty for the parent CmpRegion. If neither of the component Regions has non-default uncertainty, then the default uncertainty for the CmpRegion is 1.0E-6 of the bounding box of the CmpRegion.

Prism

The default uncertainty for a Prism is formed by combining the uncertainties from the two component Regions. If a component Region does not have a non-default uncertainty, then its default uncertainty will be used to form the default uncertainty of the parent Prism.

Region

For other classes of Region, the default uncertainty is 1.0E-6 of the bounding box of the Region. If the bounding box has zero width on any axis, then the uncertainty will be 1.0E-6 of the axis value.

Returned Value:

astGetUnc()

A pointer to a Region describing the uncertainty in the supplied Region.

Notes:

- If uncertainty information is associated with a Region, and the coordinate system described by the Region is subsequently changed (e.g. by changing the value of its System attribute, or using the astMapRegion function), then the uncertainty information returned by this function will be modified so that it refers to the coordinate system currently described by the supplied Region.
- A null Object pointer (NULL) will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

---

astGrfPop Restore previously saved graphics functions used by a Plot astGrfPop

Description: This function restores a snapshot of the graphics functions stored previously by calling astGrfPush. The restored graphics functions become the current graphics functions used by the Plot.

The astGrfPush and astGrfPop functions are intended for situations where it is necessary to make temporary changes to the graphics functions used by the Plot. The current functions should first be saved by calling astGrfPush. New functions should then be registered using astGrfSet. The required graphics should then be produced. Finally, astGrfPop should be called to restore the original graphics functions.

Synopsis: `void astGrfPop( AstPlot *this )`

Parameters:

this

Pointer to the Plot.

Notes:

- This function returns without action if there are no snapshots to restore. No error is reported in this case.

---

astGrfPush Save the current graphics functions used by a Plot astGrfPush

Description: This function takes a snapshot of the graphics functions which are currently registered with the supplied Plot, and saves the snapshot on a first-in-last-out stack within the Plot. The snapshot can be restored later using function astGrfPop.

The astGrfPush and astGrfPop functions are intended for situations where it is necessary to make temporary changes to the graphics functions used by the Plot. The current functions should first be saved by calling astGrfPush. New functions should then be registered using astGrfSet. The required graphics should then be produced. Finally, astGrfPop should be called to restore the original graphics functions.

Synopsis: `void astGrfPush( AstPlot *this )`

Parameters:

this

Pointer to the Plot.

---

astGrfSet Register a graphics function for use by a Plot astGrfSet

Description: This function can be used to select the underlying graphics functions to be used when the supplied Plot produces graphical output. If this function is not called prior to producing graphical output, then the underlying graphics functions selected at link-time (using the ast_link command) will be used. To use alternative graphics functions, call this function before the graphical output is created, specifying the graphics functions to be used. This will register the function for future use, but the function will not actually be used until the Grf attribute is given a non-zero value.

Synopsis: `void astGrfSet( AstPlot *this, const char *name, AstGrfFun fun )`

Parameters:

this

Pointer to the Plot.

name

A name indicating the graphics function to be replaced. Various graphics functions are used by the Plot class, and any combination of them may be supplied by calling this function once for each function to be replaced. If any of the graphics functions are not replaced in this way, the corresponding functions in the graphics interface selected at link-time (using the ast_link command) are used. The allowed names are:

- Attr - Enquire or set a graphics attribute value
- BBuf - Start a new graphics buffering context
- Cap - Inquire a capability
- EBuf - End the current graphics buffering context
- Flush - Flush all pending graphics to the output device
- Line - Draw a polyline (i.e. a set of connected lines)
- Mark - Draw a set of markers
- Qch - Return the character height in world coordinates
- Scales - Get the axis scales
- Text - Draw a character string
- TxExt - Get the extent of a character string

The string is case insensitive. For details of the interface required for each, see the sections below.

fun

A Pointer to the function to be used to provide the functionality indicated by parameter name. The interface for each function is described below, but the function pointer should be cast to a type of AstGrfFun when calling astGrfSet.

Once a function has been provided, a null pointer can be supplied in a subsequent call to astGrfSet to reset the function to the corresponding function in the graphics interface selected at link-time.

Function Interfaces:

All the functions listed below (except for "Cap") should return an integer value of 0 if an error occurs, and 1 otherwise. All x and y values refer to "graphics cordinates" as defined by the graphbox parameter of the astPlot call which created the Plot.

The first parameter ("grfcon") for each function is an AST KeyMap pointer that can be used by the called function to establish the context in which it is being called. The contents of the KeyMap are determined by the calling application, which should obtain a pointer to the KeyMap using the astGetGrfContext function, and then store any necessary information in the KeyMap using the methods of the KeyMap class. Note, the functions listed below should never annul or delete the supplied KeyMap pointer.

Attr:

The "Attr" function returns the current value of a specified graphics attribute, and optionally establishes a new value. The supplied value is converted to an integer value if necessary before use. It requires the following interface:

int Attr(AstObject *grfcon, int attr, double value, double *old_value, int prim)

- grfcon - A KeyMap containing information passed from the calling application.
- attr - An integer value identifying the required attribute. The following symbolic values are defined in grf.h: GRF__STYLE (Line style), GRF__WIDTH (Line width), GRF__SIZE (Character and marker size scale factor), GRF__FONT (Character font), GRF__COLOUR (Colour index).
- value - A new value to store for the attribute. If this is AST__BAD no value is stored.
- old_value - A pointer to a double in which to return the attribute value. If this is NULL, no value is returned.
- prim - The sort of graphics primitive to be drawn with the new attribute. Identified by the following values defined in grf.h: GRF__LINE, GRF__MARK, GRF__TEXT.

BBuf:

The "BBuf" function should start a new graphics buffering context. A matching call to the function "EBuf" should be used to end the context. The nature of the buffering is determined by the underlying graphics system.

int BBuf(AstObject *grfcon)

- grfcon - A KeyMap containing information passed from the calling application.

Cap:

The "Cap" function is called to determine if the grf module has a given capability, as indicated by the "cap" argument:

int Cap(AstObject *grfcon, int cap, int value)

- grfcon - A KeyMap containing information passed from the calling application.
- cap - The capability being inquired about. This will be one of the following constants defined in grf.h:

GRF__SCALES: This function should return a non-zero value if the "Scales" function is implemented, and zero otherwise. The supplied "value" argument should be ignored.

GRF__MJUST: This function should return a non-zero value if the "Text" and "TxExt" functions recognise "M" as a character in the justification string. If the first character of a justification string is "M", then the text should be justified with the given reference point at the bottom of the bounding box. This is different to "B" justification, which requests that the reference point be put on the baseline of the text, since some characters hang down below the baseline. If the "Text" or "TxExt" function cannot differentiate between "M" and "B", then this function should return zero, in which case "M" justification will never be requested by Plot. The supplied "value" argument should be ignored.

GRF__ESC: This function should return a non-zero value if the "Text" and "TxExt" functions can recognise and interpret graphics escape sequences within the supplied string (see attribute Escape). Zero should be returned if escape sequences cannot be interpreted (in which case the Plot class will interpret them itself if needed). The supplied "value" argument should be ignored only if escape sequences cannot be interpreted by "Text" and "TxExt". Otherwise, "value" indicates whether "Text" and "TxExt" should interpret escape sequences in subsequent calls. If "value" is non-zero then escape sequences should be interpreted by "Text" and "TxExt". Otherwise, they should be drawn as literal text.

- value - The use of this parameter depends on the value of "cap" as described above.
- Returned Function Value: The value returned by the function depends on the value of "cap" as described above. Zero should be returned if the supplied capability is not recognised.

EBuf:

The "EBuf" function should end the current graphics buffering context. See the description of "BBuf" above for further details. It requires the following interface:

int EBuf(AstObject ∗grfcon)

- grfcon - A KeyMap containing information passed from the calling application.

Flush:

The "Flush" function ensures that the display device is up-to-date, by flushing any pending graphics to the output device. It requires the following interface:

int Flush(AstObject ∗grfcon)

- grfcon - A KeyMap containing information passed from the calling application.

Line:

The "Line" function displays lines joining the given positions and requires the following interface:

int Line(AstObject ∗grfcon, int n, const float ∗x, const float ∗y)

- grfcon - A KeyMap containing information passed from the calling application.
- n - The number of positions to be joined together.
- x - A pointer to an array holding the "n" x values.
- y - A pointer to an array holding the "n" y values.

Mark:

The "Mark" function displays markers at the given positions. It requires the following interface:

int Mark(AstObject ∗grfcon, int n, const float ∗x, const float ∗y, int type)

- grfcon - A KeyMap containing information passed from the calling application.
- n - The number of positions to be marked.
- x - A pointer to an array holding the "n" x values.
- y - A pointer to an array holding the "n" y values.
- type - An integer which can be used to indicate the type of marker symbol required.

Qch:

The "Qch" function returns the heights of characters drawn vertically and horizontally in graphics coordinates. It requires the following interface:

int Qch(AstObject ∗grfcon, float ∗chv, float ∗chh)

- grfcon - A KeyMap containing information passed from the calling application.
- chv - A pointer to the float which is to receive the height of characters drawn with a vertical baseline. This will be an increment in the X axis.
- chh - A pointer to the float which is to receive the height of characters drawn with a horizontal baseline. This will be an increment in the Y axis.

Scales:

The "Scales" function returns two values (one for each axis) which scale increments on the corresponding axis into a "normal" coordinate system in which: 1) the axes have equal scale in terms of (for instance) millimetres per unit distance, 2) X values increase from left to right, and 3) Y values increase from bottom to top. It requires the following interface:

int Scales(AstObject ∗grfcon, float ∗alpha, float ∗beta)

- grfcon - A KeyMap containing information passed from the calling application.
- alpha - A pointer to the float which is to receive the scale for the X axis (i.e. Xnorm = alpha∗Xworld).
- beta - A pointer to the float which is to receive the scale for the Y axis (i.e. Ynorm = beta∗Yworld).

Text:

The "Text" function displays a character string at a given position using a specified justification and up-vector. It requires the following interface:

int Text(AstObject ∗grfcon, const char ∗text, float x, float y, const char ∗just, float upx, float upy)

- grfcon - A KeyMap containing information passed from the calling application.
- text - Pointer to a null-terminated character string to be displayed.
- x - The reference x coordinate.
- y - The reference y coordinate.

- just - A character string which specifies the location within the text string which is to be placed at the reference position given by x and y. The first character may be 'T' for "top", 'C' for "centre", or 'B' for "bottom", and specifies the vertical location of the reference position. Note, "bottom" corresponds to the base-line of normal text. Some characters (eg "y", "g", "p", etc) descend below the base-line. The second character may be 'L' for "left", 'C' for "centre", or 'R' for "right", and specifies the horizontal location of the reference position. If the string has less than 2 characters then 'C' is used for the missing characters.
- upx - The x component of the up-vector for the text. If necessary the supplied value should be negated to ensure that positive values always refer to displacements from left to right on the screen.
- upy - The y component of the up-vector for the text. If necessary the supplied value should be negated to ensure that positive values always refer to displacements from bottom to top on the screen.

TxExt:

The "TxExt" function returns the corners of a box which would enclose the supplied character string if it were displayed using the Text function described above. The returned box includes any leading or trailing spaces. It requires the following interface:

int TxExt(AstObject *grfcon, const char *text, float x, float y, const char *just, float upx, float upy, float *xb, float *yb)

- grfcon - A KeyMap containing information passed from the calling application.
- text - Pointer to a null-terminated character string to be displayed.
- x - The reference x coordinate.
- y - The reference y coordinate.
- just - A character string which specifies the location within the text string which is to be placed at the reference position given by x and y. See "Text" above.
- upx - The x component of the up-vector for the text. See "Text" above.
- upy - The y component of the up-vector for the text. See "Text" above.
- xb - An array of 4 elements in which to return the x coordinate of each corner of the bounding box.
- yb - An array of 4 elements in which to return the y coordinate of each corner of the bounding box.

---

astGrid Draw a set of labelled coordinate axes astGrid

Description: This function draws a complete annotated set of coordinate axes for a Plot with (optionally) a coordinate grid superimposed. Details of the axes and grid can be controlled by setting values for the various attributes defined by the Plot class (q.v.).

Synopsis: `void astGrid( AstPlot *this )`

Parameters:

this

Pointer to the Plot.

Notes:

- If the supplied Plot is a Plot3D, the axes will be annotated using three 2-dimensional Plots, one for each 2D plane in the 3D current coordinate system. The plots will be "pasted" onto 3 faces of the cuboid graphics volume specified when the Plot3D was constructed. The faces to be used can be controlled by the "RootCorner" attribute.
- An error results if either the current Frame or the base Frame of the Plot is not 2-dimensional or (for a Plot3D) 3-dimensional.
- An error also results if the transformation between the base and current Frames of the Plot is not defined in either direction (i.e. the Plot's TranForward or TranInverse attribute is zero).

---

astGridLine Draw a grid line (or axis) for a Plot astGridLine

Description: This function draws a curve in the physical coordinate system of a Plot by varying only one of the coordinates along the length of the curve. It is intended for drawing coordinate axes, coordinate grids, and tick marks on axes (but note that these are also available via the more comprehensive astGrid function).

The curve is transformed into graphical coordinate space for plotting, so that a straight line in physical coordinates may result in a curved line being drawn if the Mapping involved is non-linear. Any discontinuities in the Mapping between physical and graphical coordinates are catered for, as is any clipping established using astClip.

Synopsis:

```
void astGridLine( AstPlot *this, int axis, const double start[], double length
)
```

Parameters:

this
Pointer to the Plot.

axis
The index of the Plot axis whose physical coordinate value is to be varied along the length of the curve (all other coordinates will remain fixed). This value should lie in the range from 1 to the number of Plot axes (Naxes attribute).

start
An array, with one element for each axis of the Plot, giving the physical coordinates of the start of the curve.

length
The length of curve to be drawn, given as an increment along the selected physical axis. This may be positive or negative.

Notes:

- No curve is drawn if the "start" array contains any coordinates with the value AST__BAD, nor if "length" has this value.
- An error results if the base Frame of the Plot is not 2-dimensional.
- An error also results if the transformation between the current and base Frames of the Plot is not defined (i.e. the Plot's TranInverse attribute is zero).

---

astGrismMap Create a GrismMap astGrismMap

Description: This function creates a new GrismMap and optionally initialises its attributes.

A GrismMap is a specialised form of Mapping which transforms 1-dimensional coordinates using the spectral dispersion equation described in FITS-WCS paper III "Representation of spectral coordinates in FITS". This describes the dispersion produced by gratings, prisms and grisms.

When initially created, the forward transformation of a GrismMap transforms input "grism parameter" values into output wavelength values. The "grism parameter" is a dimensionless value which is linearly related to position on the detector. It is defined in FITS-WCS paper III as "the offset on the detector from the point of intersection of the camera axis, measured in units of the effective local length". The units in which wavelength values are expected or returned is determined by the values supplied for the GrismWaveR, GrismNRP and GrismG attribute: whatever units are used for these attributes will also be used for the wavelength values.

Synopsis: `AstGrismMap *astGrismMap( const char *options, ... )`

Parameters:

options

Pointer to a null-terminated string containing an optional comma-separated list of attribute assignments to be used for initialising the new GrismMap. The syntax used is identical to that for the astSet function and may include "printf" format specifiers identified by "%" symbols in the normal way.

...

If the "options" string contains "%" format specifiers, then an optional list of additional arguments may follow it in order to supply values to be substituted for these specifiers. The rules for supplying these are identical to those for the astSet function (and for the C "printf" function).

Returned Value:

astGrismMap()

A pointer to the new GrismMap.

Notes:

- A null Object pointer (AST__NULL) will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

---

astHasAttribute Test if an Object has a named attribute astHasAttribute

Description: This function returns a boolean result (0 or 1) to indicate whether the supplied Object has an attribute with the supplied name.

Synopsis: `int astHasAttribute( AstObject *this, const char *attrib )`

Parameters:

this

Pointer to the first Object.

attrib

Pointer to a string holding the name of the attribute to be tested.

Class Applicability:

Object
This function applies to all Objects.

Returned Value:

astHasAttribute()
One if the Object has the named attribute, otherwise zero.

Notes:

- A value of zero will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

---

astHasColumn Returns a flag indicating if a column is present in a Table astHasColumn

Description: This function returns a flag indicating if a named column exists in a Table, for instance, by having been added to to the Table using astAddColumn.

Synopsis: `int astHasColumn( AstTable *this, const char *column )`

Parameters:

this
Pointer to the Table.

column
The character string holding the upper case name of the column. Trailing spaces are ignored.

Notes:

- A value of zero is returned for if an error occurs.

---

astHasParameter Returns a flag indicating if a named global parameter is present in a Table astHasParameter

Description: This function returns a flag indicating if a named parameter exists in a Table, for instance, by having been added to to the Table using astAddParameter.

Synopsis: `int astHasParameter( AstTable *this, const char *parameter )`

Parameters:

this
Pointer to the Table.

parameter
The character string holding the upper case name of the parameter. Trailing spaces are ignored.

Notes:

- A value of zero is returned for if an error occurs.

---

astImport Import an Object pointer to the current context astImport

Description: This function imports an Object pointer that was created in a higher or lower level context, into the current AST context. This means that the pointer will be annulled when the current context is ended (with astEnd).

Synopsis: `void astImport( AstObject *this )`

Parameters:

this
Object pointer to be imported.

Class Applicability:

Object
This function applies to all Objects.

---

astIntersect Find the point of intersection between two geodesic curves astIntersect

Description: This function finds the coordinate values at the point of intersection between two geodesic curves. Each curve is specified by two points on the curve. It can only be used with 2-dimensional Frames.

For example, in a basic Frame, it will find the point of intersection between two straight lines. But for a SkyFrame it will find an intersection of two great circles.

Synopsis: `void astIntersect( AstFrame *this, const double a1[2], const double a2[2], const double b1[2], const double b2[2], double cross[2] )`

Parameters:

this
Pointer to the Frame.

a1
An array of double, with one element for each Frame axis (Naxes attribute). This should contain the coordinates of the first point on the first geodesic curve.

a2
An array of double, with one element for each Frame axis (Naxes attribute). This should contain the coordinates of a second point on the first geodesic curve. It should not be co-incident with the first point.

b1
An array of double, with one element for each Frame axis (Naxes attribute). This should contain the coordinates of the first point on the second geodesic curve.

b2
An array of double, with one element for each Frame axis (Naxes attribute). This should contain the coordinates of a second point on the second geodesic curve. It should not be co-incident with the first point.

cross
An array of double, with one element for each Frame axis in which the coordinates of the required intersection will be returned.

Notes:

- For SkyFrames each curve will be a great circle, and in general each pair of curves will intersect at two diametrically opposite points on the sky. The returned position is the one which is closest to point "a1".
- This function will return "bad" coordinate values (AST__BAD) if any of the input coordinates has this value, or if the two points defining either geodesic are co-incident, or if the two curves do not intersect.
- The geodesic curve used by this function is the path of shortest distance between two points, as defined by the astDistance function.
- An error will be reported if the Frame is not 2-dimensional.

---

astInterval Create a Interval astInterval

Description: This function creates a new Interval and optionally initialises its attributes.

A Interval is a Region which represents upper and/or lower limits on one or more axes of a Frame. For a point to be within the region represented by the Interval, the point must satisfy all the restrictions placed on all the axes. The point is outside the region if it fails to satisfy any one of the restrictions. Each axis may have either an upper limit, a lower limit, both or neither. If both limits are supplied but are in reverse order (so that the lower limit is greater than the upper limit), then the interval is an excluded interval, rather than an included interval.

At least one axis limit must be supplied.

Note, The Interval class makes no allowances for cyclic nature of some coordinate systems (such as SkyFrame coordinates). A Box should usually be used in these cases since this requires the user to think about suitable upper and lower limits,

Synopsis: `AstInterval *astInterval( AstFrame *frame, const double lbnd[], const double ubnd[], AstRegion *unc, const char *options, ... )`

Parameters:

frame

A pointer to the Frame in which the region is defined. A deep copy is taken of the supplied Frame. This means that any subsequent changes made to the Frame using the supplied pointer will have no effect the Region.

lbnd

An array of double, with one element for each Frame axis (Naxes attribute) containing the lower limits on each axis. Set a value to AST__BAD to indicate that the axis has no lower limit.

ubnd

An array of double, with one element for each Frame axis (Naxes attribute) containing the upper limits on each axis. Set a value to AST__BAD to indicate that the axis has no upper limit.

unc

An optional pointer to an existing Region which specifies the uncertainties associated with the boundary of the Box being created. The uncertainty in any point on the boundary of the Box is found by shifting the supplied "uncertainty" Region so that it is centred at the boundary point being considered. The area covered by the shifted uncertainty Region then represents the uncertainty in the boundary position. The uncertainty is assumed to be the same for all points.

If supplied, the uncertainty Region must be of a class for which all instances are centro-symetric (e.g. Box, Circle, Ellipse, etc.) or be a Prism containing centro-symetric component Regions. A deep copy of the supplied Region will be taken, so subsequent changes to the uncertainty Region using the supplied pointer will have no effect on the created Box. Alternatively, a NULL Object pointer may be supplied, in which case a default uncertainty is used equivalent to a box 1.0E-6 of the size of the Box being created.

The uncertainty Region has two uses: 1) when the astOverlap function compares two Regions for equality the uncertainty Region is used to determine the tolerance on the comparison, and 2) when a Region is mapped into a different coordinate system and subsequently simplified (using astSimplify), the uncertainties are used to determine if the transformed boundary can be accurately represented by a specific shape of Region.

options

Pointer to a null-terminated string containing an optional comma-separated list of attribute assignments to be used for initialising the new Interval. The syntax used is identical to that for the astSet function and may include "printf" format specifiers identified by "%" symbols in the normal way.

...

If the "options" string contains "%" format specifiers, then an optional list of additional arguments may follow it in order to supply values to be substituted for these specifiers. The rules for supplying these are identical to those for the astSet function (and for the C "printf" function).

Returned Value:

astInterval()

A pointer to the new Interval.

Notes:

- A null Object pointer (AST__NULL) will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

Status Handling:

The protected interface to this function includes an extra parameter at the end of the parameter list descirbed above. This parameter is a pointer to the integer inherited status variable: "int *status".

---

astIntraMap Create an IntraMap astIntraMap

Description: This function creates a new IntraMap and optionally initialises its attributes.

An IntraMap is a specialised form of Mapping which encapsulates a privately-defined coordinate transformation function (e.g. written in C) so that it may be used like any other AST Mapping. This allows you to create Mappings that perform any conceivable coordinate transformation.

However, an IntraMap is intended for use within a single program or a private suite of software, where all programs have access to the same coordinate transformation functions (i.e. can be linked against them). IntraMaps should not normally be stored in datasets which may be exported for processing by other software, since that software will not have the necessary transformation functions available, resulting in an error.

You must register any coordinate transformation functions to be used using astIntraReg before creating an IntraMap.

Synopsis: `AstIntraMap *astIntraMap( const char *name, int nin, int nout, const char *options, ... )`

Parameters:

name

Pointer to a null-terminated string containing the name of the transformation function to use (which should previously have been registered using astIntraReg). This name is case sensitive. All white space will be removed before use.

nin

The number of input coordinates. This must be compatible with the number of input coordinates accepted by the transformation function (as specified when this function was registered using astIntraReg).

nout

The number of output coordinates. This must be compatible with the number of output coordinates produced by the transformation function (as specified when this function was registered using astIntraReg).

options

Pointer to a null-terminated string containing an optional comma-separated list of attribute assignments to be used for initialising the new IntraMap. The syntax used is identical to that for the astSet function and may include "printf" format specifiers identified by "%" symbols in the normal way.

...

If the "options" string contains "%" format specifiers, then an optional list of additional arguments may follow it in order to supply values to be substituted for these specifiers. The rules for supplying these are identical to those for the astSet function (and for the C "printf" function).

Returned Value:

astIntraMap()

A pointer to the new IntraMap.

Notes:

- A null Object pointer (AST__NULL) will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

---

astIntraReg Register a transformation function for use by an IntraMap astIntraReg

Description: This function registers a privately-defined coordinate transformation function written in C so that it may be used to create an IntraMap. An IntraMap is a specialised form of Mapping which encapsulates the C function so that it may be used like any other AST Mapping. This allows you to create Mappings that perform any conceivable coordinate transformation.

Registration of relevant transformation functions is required before using the astIntraMap constructor function to create an IntraMap or reading an external representation of an IntraMap from a Channel.

Synopsis: `astIntraReg( const char *name, int nin, int nout, void (* tran)( AstMapping *, int, int, const double *[], int, int, double *[] ), unsigned int flags, const char *purpose, const char *author, const char *contact )`

Parameters:

name

Pointer to a null-terminated string containing a unique name to be associated with the transformation function in order to identify it. This name is case sensitive. All white space will be removed before use.

nin

The number of input coordinates accepted by the transformation function (i.e. the number of dimensions of the space in which the input points reside). A value of AST__ANY may be given if the function is able to accommodate a variable number of input coordinates.

nout

The number of output coordinates produced by the transformation function (i.e. the number of dimensions of the space in which the output points reside). A value of AST__ANY may be given if the function is able to produce a variable number of output coordinates.

tran

Pointer to the transformation function to be registered. This function should perform whatever coordinate transformations are required and should have an interface like astTranP (q.v.).

flags

This value may be used to supply a set of flags which describe the transformation function and which may affect the behaviour of any IntraMap which uses it. Often, a value of zero will be given here, but you may also supply the bitwise OR of a set of flags as described in the "Transformation Flags" section (below).

purpose

Pointer to a null-terminated string containing a short (one line) textual comment to describe the purpose of the transformation function.

author

Pointer to a null-terminated string containing the name of the author of the transformation function.

contact

Pointer to a null-terminated string containing contact details for the author of the transformation function (e.g. an e-mail or WWW address). If any IntraMap which uses this transformation function is exported as part of a dataset to an external user who does not have access to the function, then these contact details should allow them to obtain the necessary code.

Notes:

- Beware that an external representation of an IntraMap (created by writing it to a Channel) will not include the coordinate transformation function which it uses, so will only refer to the function by its name (as assigned using astIntraReg). Consequently, the external representation cannot be utilised by another program unless that program has also registered the same transformation function with the same name using an identical invocation of astIntraReg. If no such registration has been performed, then attempting to read the external representation will result in an error.
- You may use astIntraReg to register a transformation function with the same name more than once, but only if the arguments supplied are identical on each occasion (i.e there is no way of changing things once a function has been successfully registered under a given name, and attempting to do so will result in an error). This feature simply allows registration to be performed independently, but consistently, at several places within your program, without having to check whether it has already been done.
- If an error occurs in the transformation function, this may be indicated by setting the AST error status to an error value (using astSetStatus) before it returns. This will immediately terminate the current AST operation. The error value AST__ITFER is available for this purpose, but other values may also be used (e.g. if you wish to distinguish different types of error).

Transformation Flags:

The following flags are defined in the "ast.h" header file and allow you to provide further information about the nature of the transformation function. Having selected the set of flags which apply, you should supply the bitwise OR of their values as the "flags" argument to astIntraReg.

- AST__NOFWD: If this flag is set, it indicates that the transformation function does not implement a forward coordinate transformation. In this case, any IntraMap which uses it will have a TranForward attribute value of zero and the transformation function itself will not be invoked with its "forward" argument set to a non-zero value. By default, it is assumed that a forward transformation is provided.
- AST__NOINV: If this flag is set, it indicates that the transformation function does not implement an inverse coordinate transformation. In this case, any IntraMap which uses it will have a TranInverse attribute value of zero and the transformation function itself will not be invoked with its "forward" argument set to zero. By default, it is assumed that an inverse transformation is provided.
- AST__SIMPFI: You may set this flag if applying the transformation function's forward coordinate transformation, followed immediately by the matching inverse transformation, should always restore the original set of coordinates. It indicates that AST may replace such a sequence of operations by an identity Mapping (a UnitMap) if it is encountered while simplifying a compound Mapping (e.g. using astSimplify). It is not necessary that both transformations have actually been implemented.
- AST__SIMPIF: You may set this flag if applying the transformation function's inverse coordinate transformation, followed immediately by the matching forward transformation, should always restore the original set of coordinates. It indicates that AST may replace such a sequence of operations by an identity Mapping (a UnitMap) if it is encountered while simplifying a compound Mapping (e.g. using astSimplify). It is not necessary that both transformations have actually been implemented.

---

astInvert Invert a Mapping astInvert

Description: This function inverts a Mapping by reversing the boolean sense of its Invert attribute. If this attribute is zero (the default), the Mapping will transform coordinates in the way specified when it was created. If it is non-zero, the input and output coordinates will be inter-changed so that the direction of the Mapping is reversed. This will cause it to display the inverse of its original behaviour.

Synopsis: `void astInvert( AstMapping *this )`

Parameters:

this

Pointer to the Mapping.

---

astIsA<Class> Test membership of a class by an Object astIsA<Class>

Description: This is a family of functions which test whether an Object is a member of the class called <Class>, or of any class derived from it.

Synopsis: `int astIsA<Class>( const Ast<Class> *this )`

Parameters:

this

Pointer to the Object.

Class Applicability:

Object

These functions apply to all Objects.

Returned Value:

astIsA<Class>()

One if the Object belongs to the class called <Class> (or to a class derived from it), otherwise zero.

Examples:

`member = astIsAFrame( obj );`

Tests whether Object "obj" is a member of the Frame class, or of any class derived from a Frame.

Notes:

- Every AST class provides a function (astIsA<Class>) of this form, where <Class> should be replaced by the class name.
- This function attempts to execute even if the AST error status is set on entry, although no further error report will be made if it subsequently fails under these circumstances.
- A value of zero will be returned if this function should fail for any reason. In particular, it will fail if the pointer supplied does not identify an Object of any sort.

---

astKeyMap Create a KeyMap astKeyMap

Description: This function creates a new empty KeyMap and optionally initialises its attributes. Entries can then be added to the KeyMap using the astMapPut0<X> and astMapPut1<X> functions.

The KeyMap class is used to store a set of values with associated keys which identify the values. The keys are strings. These may be case sensitive or insensitive as selected by the KeyCase attribute, and trailing spaces are ignored. The value associated with a key can be integer (signed 4 and 2 byte, or unsigned 1 byte), floating point (single or double precision), void pointer, character string or AST Object pointer. Each value can be a scalar or a one-dimensional vector. A KeyMap is conceptually similar to a Mapping in that a KeyMap transforms an input into an output - the input is the key, and the output is the value associated with the key. However, this is only a conceptual similarity, and it should be noted that the KeyMap class inherits from the Object class rather than the Mapping class. The methods of the Mapping class cannot be used with a KeyMap.

Synopsis: `AstKeyMap *astKeyMap( const char *options, ... )`

Parameters:

options

Pointer to a null-terminated string containing an optional comma-separated list of attribute assignments to be used for initialising the new KeyMap. The syntax used is identical to that for the astSet function and may include "printf" format specifiers identified by "%" symbols in the normal way.

...

If the "options" string contains "%" format specifiers, then an optional list of additional arguments may follow it in order to supply values to be substituted for these specifiers. The rules for supplying these are identical to those for the astSet function (and for the C "printf" function).

Returned Value:

astKeyMap()

A pointer to the new KeyMap.

Notes:

- A null Object pointer (AST__NULL) will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

Status Handling:

The protected interface to this function includes an extra parameter at the end of the parameter list descirbed above. This parameter is a pointer to the integer inherited status variable: "int ∗status".

---

astLinearApprox Obtain a linear approximation to a Mapping, if appropriate astLinearApprox

Description: This function tests the forward coordinate transformation implemented by a Mapping over a given range of input coordinates. If the transformation is found to be linear to a specified level of accuracy, then an array of fit coefficients is returned. These may be used to implement a linear approximation to the Mapping's forward transformation within the specified range of output coordinates. If the transformation is not sufficiently linear, no coefficients are returned.

Synopsis: `int astLinearApprox( AstMapping *this, const double *lbnd, const double *ubnd, double tol, double *fit )`

Parameters:

this

Pointer to the Mapping.

lbnd

Pointer to an array of doubles containing the lower bounds of a box defined within the input coordinate system of the Mapping. The number of elements in this array should equal the value of the Mapping's Nin attribute. This box should specify the region over which linearity is required.

ubnd

Pointer to an array of doubles containing the upper bounds of the box specifying the region over which linearity is required.

tol

The maximum permitted deviation from linearity, expressed as a positive Cartesian displacement in the output coordinate space of the Mapping. If a linear fit to the forward transformation of the Mapping deviates from the true transformation by more than this amount at any point which is tested, then no fit coefficients will be returned.

fit

Pointer to an array of doubles in which to return the co-efficients of the linear approximation to the specified transformation. This array should have at least "(Nin + 1) ∗ Nout", elements. The first Nout elements hold the constant offsets for the transformation outputs. The remaining elements hold the gradients. So if the Mapping has 2 inputs and 3 outputs the linear approximation to the forward transformation is:

X_out = fit[0] + fit[3]∗X_in + fit[4]∗Y_in

Y_out = fit[1] + fit[5]∗X_in + fit[6]∗Y_in

Z_out = fit[2] + fit[7]∗X_in + fit[8]∗Y_in

Returned Value:

astLinearApprox()

If the forward transformation is sufficiently linear, a non-zero value is returned. Otherwise zero is returned and the fit co-efficients are set to AST__BAD.

Notes:

- This function fits the Mapping's forward transformation. To fit the inverse transformation, the Mapping should be inverted using astInvert before invoking this function.
- A value of zero will be returned if this function is invoked with the global error status set, or if it should fail for any reason.

---

astLock Lock an Object for exclusive use by the calling thread astLock

Description: The thread-safe public interface to AST is designed so that an error is reported if any thread attempts to use an Object that it has not previously locked for its own exclusive use using this function. When an Object is created, it is initially locked by the thread that creates it, so newly created objects do not need to be explicitly locked. However, if an Object pointer is passed to another thread, the original thread must first unlock it (using astUnlock) and the new thread must then lock it (using astLock) before the new thread can use the Object.

The "wait" parameter determines what happens if the supplied Object is curently locked by another thread when this function is invoked.

Synopsis: `void astLock( AstObject *this, int wait )`

Parameters:

this

Pointer to the Object to be locked.

wait

If the Object is curently locked by another thread then this function will either report an error or block. If a non-zero value is supplied for "wait", the calling thread waits until the object is available for it to use. Otherwise, an error is reported and the function returns immediately without locking the Object.

Class Applicability:

Object

This function applies to all Objects.

Notes:

- The astAnnul function is exceptional in that it can be used on pointers for Objects that are not currently locked by the calling thread. All other AST functions will report an error.
- The Locked object will belong to the current AST context.
- This function returns without action if the Object is already locked by the calling thread.
- If simultaneous use of the same object is required by two or more threads, astCopy should be used to to produce a deep copy of the Object for each thread. Each copy should then be unlocked by the parent thread (i.e. the thread that created the copy), and then locked by the child thread (i.e. the thread that wants to use the copy).
- This function is only available in the C interface.
- This function returns without action if the AST library has been built without POSIX thread support (i.e. the "-with-pthreads" option was not specified when running the "configure" script).

astLutMap Create a LutMap astLutMap

Description: This function creates a new LutMap and optionally initialises its attributes.

A LutMap is a specialised form of Mapping which transforms 1-dimensional coordinates by using linear interpolation in a lookup table. Each input coordinate value is first scaled to give the index of an entry in the table by subtracting a starting value (the input coordinate corresponding to the first table entry) and dividing by an increment (the difference in input coordinate value between adjacent table entries).

The resulting index will usually contain a fractional part, so the output coordinate value is then generated by interpolating linearly between the appropriate entries in the table. If the index lies outside the range of the table, linear extrapolation is used based on the two nearest entries (i.e. the two entries at the start or end of the table, as appropriate).

If the lookup table entries increase or decrease monotonically, then the inverse transformation may also be performed.

Synopsis: `AstLutMap *astLutMap( int nlut, const double lut[], double start, double inc, const char *options, ... )`

Parameters:

nlut

The number of entries in the lookup table. This value must be at least 2.

lut

An array containing the "nlut" lookup table entries.

start

The input coordinate value which corresponds to the first lookup table entry.

inc

The lookup table spacing (the increment in input coordinate value between successive lookup table entries). This value may be positive or negative, but must not be zero.

options

Pointer to a null-terminated string containing an optional comma-separated list of attribute assignments to be used for initialising the new LutMap. The syntax used is identical to that for the astSet function and may include "printf" format specifiers identified by "%" symbols in the normal way.

...

If the "options" string contains "%" format specifiers, then an optional list of additional arguments may follow it in order to supply values to be substituted for these specifiers. The rules for supplying these are identical to those for the astSet function (and for the C "printf" function).

Returned Value:

astLutMap()

A pointer to the new LutMap.

Notes:

- If the entries in the lookup table either increase or decrease monotonically, then the new LutMap's TranInverse attribute will have a value of one, indicating that the inverse transformation can be performed. Otherwise, it will have a value of zero, so that any attempt to use the inverse transformation will result in an error.
- A null Object pointer (AST__NULL) will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

Status Handling:

The protected interface to this function includes an extra parameter at the end of the parameter list descirbed above. This parameter is a pointer to the integer inherited status variable: "int *status".

---

astMapBox Find a bounding box for a Mapping astMapBox

Description: This function allows you to find the "bounding box" which just encloses another box after it has been transformed by a Mapping (using either its forward or inverse transformation). A typical use might be to calculate the size of an image after being transformed by a Mapping.

The function works on one dimension at a time. When supplied with the lower and upper bounds of a rectangular region (box) of input coordinate space, it finds the lowest and highest values taken by a nominated output coordinate within that region. Optionally, it also returns the input coordinates where these bounding values are attained. It should be used repeatedly to obtain the extent of the bounding box in more than one dimension.

Synopsis:

```
void astMapBox( AstMapping *this, const double lbnd_in[], const double ubnd_in[],
int forward, int coord_out, double *lbnd_out, double *ubnd_out, double xl[], double
xu[] );
```

Parameters:

this

Pointer to the Mapping.

lbnd_in

Pointer to an array of double, with one element for each Mapping input coordinate. This should contain the lower bound of the input box in each input dimension.

ubnd_in

Pointer to an array of double, with one element for each Mapping input coordinate. This should contain the upper bound of the input box in each input dimension.

Note that it is permissible for the upper bound to be less than the corresponding lower bound, as the values will simply be swapped before use.

forward

If this value is non-zero, then the Mapping's forward transformation will be used to transform the input box. Otherwise, its inverse transformation will be used.

(If the inverse transformation is selected, then references to "input" and "output" coordinates in this description should be transposed. For example, the size of the "lbnd_in" and "ubnd_in" arrays should match the number of output coordinates, as given by the Mapping's Nout attribute. Similarly, the "coord_out" parameter, below, should nominate one of the Mapping's input coordinates.)

coord_out

The index of the output coordinate for which the lower and upper bounds are required. This value should be at least one, and no larger than the number of Mapping output coordinates.

lbnd_out

Pointer to a double in which to return the lowest value taken by the nominated output coordinate within the specified region of input coordinate space.

ubnd_out

Pointer to a double in which to return the highest value taken by the nominated output coordinate within the specified region of input coordinate space.

xl

An optional pointer to an array of double, with one element for each Mapping input coordinate. If given, this array will be filled with the coordinates of an input point (although not necessarily a unique one) for which the nominated output coordinate attains the lower bound value returned in `"*lbnd_out"`.

If these coordinates are not required, a NULL pointer may be supplied.

xu

An optional pointer to an array of double, with one element for each Mapping input coordinate. If given, this array will be filled with the coordinates of an input point (although not necessarily a unique one) for which the nominated output coordinate attains the upper bound value returned in `"*ubnd_out"`.

If these coordinates are not required, a NULL pointer may be supplied.

Notes:

- Any input points which are transformed by the Mapping to give output coordinates containing the value AST__BAD are regarded as invalid and are ignored. They will make no contribution to determining the output bounds, even although the nominated output coordinate might still have a valid value at such points.
- An error will occur if the required output bounds cannot be found. Typically, this might happen if all the input points which the function considers turn out to be invalid (see above). The number of points considered before generating such an error is quite large, so this is unlikely to occur by accident unless valid points are restricted to a very small subset of the input coordinate space.
- The values returned via `"lbnd_out"`, `"ubnd_out"`, `"xl"` and `"xu"` will be set to the value AST__BAD if this function should fail for any reason. Their initial values on entry will not be altered if the function is invoked with the AST error status set.

---

astMapCopy Copy entries from one KeyMap into another astMapCopy

Description: This function copies all entries from one KeyMap into another.

Synopsis: `void astMapCopy( AstKeyMap *this, AstKeyMap *that )`

Parameters:

this

Pointer to the destination KeyMap.

that

Pointer to the source KeyMap.

Notes:

- Entries from the source KeyMap will replace any existing entries in the destination KeyMap that have the same key.
- The one exception to the above rule is that if a source entry contains a scalar KeyMap entry, and the destination contains a scalar KeyMap entry with the same key, then the source KeyMap entry will be copied into the destination KeyMap entry using this function, rather than simply replacing the destination KeyMap entry.
- If the destination entry has a non-zero value for its MapLocked attribute, then an error will be reported if the source KeyMap contains any keys that do not already exist within the destination KeyMap.

---

astMapDefined<X> Check if a KeyMap contains a defined value for a key astMapDefined<X>

Description: This function checks to see if a KeyMap contains a defined value for a given key. If the key is present in the KeyMap but has an undefined value it returns zero (unlike astMapHasKey which would return non-zero).

Synopsis: `int astMapDefined( AstKeyMap *this, const char *key );`

Parameters:

this

Pointer to the KeyMap.

key

The character string identifying the value to be retrieved. Trailing spaces are ignored. The supplied string is converted to upper case before use if the KeyCase attribute is currently set to zero.

Returned Value:

astMapDefined()

A non-zero value is returned if the requested key name is present in the KeyMap and has a defined value.

---

astMapGet0<X> Get a scalar value from a KeyMap astMapGet0<X>

Description: This is a set of functions for retrieving a scalar value from a KeyMap. You should replace <X> in the generic function name astMapGet0<X> by an appropriate 1-character type code (see the "Data Type Codes" section below for the code appropriate to each supported data type). The stored value is converted to the data type indiced by <X> before being returned (an error is reported if it is not possible to convert the stored value to the requested data type).

Synopsis: `int astMapGet0<X>( AstKeyMap *this, const char *key, <X>type *value );`

Parameters:

this

Pointer to the KeyMap.

key

The character string identifying the value to be retrieved. Trailing spaces are ignored. The supplied string is converted to upper case before use if the KeyCase attribute is currently set to zero.

value

A pointer to a buffer in which to return the requested value. If the requested key is not found, or if it is found but has an undefined value (see astMapPutU), then the contents of the buffer on entry to this function will be unchanged on exit. For pointer types ("A" and "C"), the buffer should be a suitable pointer, and the address of this pointer should be supplied as the "value" parameter.

Returned Value:

astMapGet0<X>()

A non-zero value is returned if the requested key name was found, and does not have an undefined value (see astMapPutU). Zero is returned otherwise.

Notes:

- No error is reported if the requested key cannot be found in the given KeyMap, but a zero value will be returned as the function value. The supplied buffer will be returned unchanged.
- If the stored value is a vector value, then the first value in the vector will be returned.
- A string pointer returned by astMapGet0C is guaranteed to remain valid and the string to which it points will not be over-written for a total of 50 successive invocations of this function. After this, the memory containing the string may be re-used, so a copy of the string should be made if it is needed for longer than this.
- If the returned value is an AST Object pointer, the Object's reference count is incremented by this call. Any subsequent changes made to the Object using the returned pointer will be reflected in any any other active pointers for the Object. The returned pointer should be annulled using astAnnul when it is no longer needed.

Data Type Codes:

To select the appropriate function, you should replace <X> in the generic function name astMapGet0<X> with a 1-character data type code, so as to match the data type <X>type of the data you are processing, as follows:

- F: float
- D: double
- I: int
- C: "const" pointer to null terminated character string
- A: Pointer to AstObject
- P: Generic "void *" pointer
- S: short int
- B: Unsigned byte (i.e. word)

For example, astMapGet0D would be used to get a "double" value, while astMapGet0I would be used to get an "int", etc.

---

astMapGet1<X> Get a vector value from a KeyMap astMapGet1<X>

Description: This is a set of functions for retrieving a vector value from a KeyMap. You should replace <X> in the generic function name astMapGet1<X> by an appropriate 1-character type code (see the "Data Type Codes" section below for the code appropriate to each supported data type). The stored value is converted to the data type indiced by <X> before being returned (an error is reported if it is not possible to convert the stored value to the requested data type). Note, the astMapGet1C function has an extra parameter "l" which specifies the maximum length of each string to be stored in the "value" buffer (see the "astMapGet1C" section below).

Synopsis: `int astMapGet1<X>( AstKeyMap *this, const char *key, int mxval, int *nval, <X>type *value ) int astMapGet1C( AstKeyMap *this, const char *key, int l, int mxval, int *nval, const char *value )`

Parameters:

this

Pointer to the KeyMap.

key

The character string identifying the value to be retrieved. Trailing spaces are ignored. The supplied string is converted to upper case before use if the KeyCase attribute is currently set to zero.

mxval

The number of elements in the "value" array.

nval

The address of an integer in which to put the number of elements stored in the "value" array. Any unused elements of the array are left unchanged.

value

A pointer to an array in which to return the requested values. If the requested key is not found, or if it is found but has an undefined value (see astMapPutU), then the contents of the buffer on entry to this function will be unchanged on exit.

Returned Value:

astMapGet1<X>()

A non-zero value is returned if the requested key name was found, and does not have an undefined value (see astMapPutU). Zero is returned otherwise.

Notes:

- No error is reported if the requested key cannot be found in the given KeyMap, but a zero value will be returned as the function value. The supplied array will be returned unchanged.
- If the stored value is a scalar value, then the value will be returned in the first element of the supplied array, and "nval" will be returned set to 1.

astMapGet1C:

The "value" buffer supplied to the astMapGet1C function should be a pointer to a character array with "mxval∗l" elements, where "l" is the maximum length of a string to be returned. The value of "l" should be supplied as an extra parameter following "key" when invoking astMapGet1C, and should include space for a terminating null character.

Data Type Codes:

To select the appropriate function, you should replace <X> in the generic function name astMapGet1<X> with a 1-character data type code, so as to match the data type <X>type of the data you are processing, as follows:

- D: double
- F: float
- I: int
- C: "const" pointer to null terminated character string
- A: Pointer to AstObject
- P: Generic "void ∗" pointer
- S: short int
- B: Unsigned byte (i.e. char)

For example, astMapGet1D would be used to get "double" values, while astMapGet1I would be used to get "int" values, etc. For D or I, the supplied "value" parameter should be a pointer to an array of doubles or ints, with "mxval" elements. For C, the supplied "value" parameter should be a pointer to a character string with "mxval∗l" elements. For A, the supplied "value" parameter should be a pointer to an array of AstObject pointers.

astMapGetElem<X> Get a single element of a vector value from a KeyMap astMapGetElem<X>

Description: This is a set of functions for retrieving a single element of a vector value from a KeyMap. You should replace <X> in the generic function name astMapGetElem<X> by an appropriate 1-character type code (see the "Data Type Codes" section below for the code appropriate to each supported data type). The stored value is converted to the data type indiced by <X> before being returned (an error is reported if it is not possible to convert the stored value to the requested data type). Note, the astMapGetElemC function has an extra parameter "l" which specifies the maximum length of the string to be stored in the "value" buffer (see the "astMapGetElemC" section below).

Synopsis: `int astMapGetElem<X>( AstKeyMap *this, const char *key, int elem, <X>type *value ) int astMapGetElemC( AstKeyMap *this, const char *key, int l, int elem, char *value )`

Parameters:

this
Pointer to the KeyMap.

key
The character string identifying the value to be retrieved. Trailing spaces are ignored. The supplied string is converted to upper case before use if the KeyCase attribute is currently set to zero.

elem
The index of the required vector element, starting at zero. An error will be reported if the value is outside the range of the vector.

value
A pointer to a buffer in which to return the requested value. If the requested key is not found, or if it is found but has an undefined value (see astMapPutU), then the contents of the buffer on entry to this function will be unchanged on exit.

Returned Value:

astMapGetElem<X>()
A non-zero value is returned if the requested key name was found, and does not have an undefined value (see astMapPutU). Zero is returned otherwise.

Notes:

- No error is reported if the requested key cannot be found in the given KeyMap, or if it has an undefined value, but a zero value will be returned as the function value.

astMapGetElemC:

The "value" buffer supplied to the astMapGetElemC function should be a pointer to a character array with "l" elements, where "l" is the maximum length of the string to be returned. The value of "l" should be supplied as an extra parameter following "key" when invoking astMapGetElemC, and should include space for a terminating null character.

Data Type Codes:

To select the appropriate function, you should replace <X> in the generic function name astMapGetElem<X> with a 1-character data type code, so as to match the data type <X>type of the data you are processing, as follows:

- D: double
- F: float
- I: int
- C: "const" pointer to null terminated character string
- A: Pointer to AstObject
- P: Generic "void *" pointer
- S: short int
- B: Unsigned byte (i.e. char)

For example, astMapGetElemD would be used to get a "double" value, while astMapGetElemI would be used to get an "int" value, etc. For D or I, the supplied "value" parameter should be a pointer to a double or int. For C, the supplied "value" parameter should be a pointer to a character string with "l" elements. For A, the supplied "value" parameter should be a pointer to an AstObject pointer.

---

astMapHasKey Check if an entry with a given key exists in a KeyMap astMapHasKey

Description: This function returns a flag indicating if the KeyMap contains an entry with the given key.

Synopsis: `int astMapHasKey( AstKeyMap *this, const char *key )`

Parameters:

this
Pointer to the KeyMap.

key
The character string identifying the KeyMap entry. Trailing spaces are ignored. The supplied string is converted to upper case before use if the KeyCase attribute is currently set to zero.

Returned Value:

astMapHasKey()
Non-zero if the key was found, and zero otherwise.

Notes:

- A non-zero function value is returned if the key exists but has an undefined value (that is, the returned value does not depend on whether the entry has a defined value or not). See also astMapDefined, which returns zero in such a case.
- A function value of zero will be returned if an error has already occurred, or if this function should fail for any reason.

---

astMapKey Get the key at a given index within the KeyMap astMapKey

Description: This function returns a string holding the key for the entry with the given index within the KeyMap.

This function is intended primarily as a means of iterating round all the elements in a KeyMap. For this purpose, the number of entries in the KeyMap should first be found using astMapSize and this function should then be called in a loop, with the index value going from zero to one less than the size of the KeyMap. The index associated with a given entry is determined by the SortBy attribute.

Synopsis: `const char *astMapKey( AstKeyMap *this, int index )`

Parameters:

this
Pointer to the KeyMap.

index
The index into the KeyMap. The first entry has index zero, and the last has index "size-1", where "size" is the value returned by the astMapSize function.

Returned Value:

astMapKey()
A pointer to a null-terminated string containing the key.

Notes:

- The returned pointer is guaranteed to remain valid and the string to which it points will not be over-written for a total of 50 successive invocations of this function. After this, the memory containing the string may be re-used, so a copy of the string should be made if it is needed for longer than this.
- A NULL pointer will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

---

astMapLenC Get the number of characters in a character entry in a KeyMap astMapLenC

Description: This function returns the minimum length which a character variable which must have in order to be able to store a specified entry in the supplied KeyMap. If the named entry is a vector entry, then the returned value is the length of the longest element of the vector value.

Synopsis: `int astMapLenC( AstKeyMap *this, const char *key )`

Parameters:

this
Pointer to the KeyMap.

key
The character string identifying the KeyMap entry. Trailing spaces are ignored. The supplied string is converted to upper case before use if the KeyCase attribute is currently set to zero.

Returned Value:

astMapLenC()
The length (i.e. number of characters) of the longest formatted value associated with the named entry. This does not include the trailing null character.

Notes:

- A function value of zero will be returned without error if the named entry cannot be formatted as a character string.
- A function value of zero will be returned if an error has already occurred, or if this function should fail for any reason.

---

astMapLength Get the vector length of an entry in a KeyMap astMapLength

Description: This function returns the vector length of a named entry in a KeyMap, (that is, how many values are associated with the entry).

Synopsis: `int astMapLength( AstKeyMap *this, const char *key )`

Parameters:

this
Pointer to the KeyMap.

key
The character string identifying the KeyMap entry. Trailing spaces are ignored. The supplied string is converted to upper case before use if the KeyCase attribute is currently set to zero.

Returned Value:

astMapLength()
The length of the entry. One for a scalar, greater than one for a vector. A value of zero is returned if the KeyMap does not contain the named entry.

Notes:

- A function value of zero will be returned if an error has already occurred, or if this function should fail for any reason.

---

astMapPut0<X> Add a scalar value to a KeyMap astMapPut0<X>

Description: This is a set of functions for adding scalar values to a KeyMap. You should use a function which matches the data type of the data you wish to add to the KeyMap by replacing <X> in the generic function name astMapPut0<X> by an appropriate 1-character type code (see the "Data Type Codes" section below for the code appropriate to each supported data type).

Synopsis: `void astMapPut0<X>( AstKeyMap *this, const char *key, <X>type value, const char *comment );`

Parameters:

this
Pointer to the KeyMap in which to store the supplied value.

key
A character string to be stored with the value, which can later be used to identify the value. Trailing spaces are ignored. The supplied string is converted to upper case before use if the KeyCase attribute is currently set to zero.

value
The value to be stored. The data type of this value should match the 1-character type code appended to the function name (e.g. if you are using astMapPut0A, the type of this value should be "pointer to AstObject").

comment
A pointer to a null-terminated comment string to be stored with the value. A NULL pointer may be supplied, in which case no comment is stored.

Notes:

- If the supplied key is already in use in the KeyMap, the new value will replace the old value.
- If the stored value is an AST Object pointer, the Object's reference count is incremented by this call. Any subsequent changes made to the Object using the returned pointer will be reflected in any any other active pointers for the Object, including any obtained later using astMapget0A. The reference count for the Object will be decremented when the KeyMap is destroyed, or the entry is removed or over-written with a different pointer.

Data Type Codes:

To select the appropriate function, you should replace <X> in the generic function name astMapPut0<X> with a 1-character data type code, so as to match the data type <X>type of the data you are processing, as follows:

- D: double
- F: float
- I: int
- C: "const" pointer to null terminated character string
- A: Pointer to AstObject
- P: Generic "void *" pointer
- S: short int
- B: unsigned byte (i.e. unsigned char)

For example, astMapPut0D would be used to store a "double" value, while astMapPut0I would be used to store an "int", etc.

Note that KeyMaps containing generic "void *" pointers cannot be written out using astShow or astWrite. An error will be reported if this is attempted.

---

astMapPut1<X> Add a vector value to a KeyMap astMapPut1<X>

Description: This is a set of functions for adding vector values to a KeyMap. You should use a function which matches the data type of the data you wish to add to the KeyMap by replacing <X> in the generic function name astMapPut1<X> by an appropriate 1-character type code (see the "Data Type Codes" section below for the code appropriate to each supported data type).

Synopsis: `void astMapPut1<X>( AstKeyMap *this, const char *key, int size, const <X>type value[], const char *comment );`

Parameters:

this
Pointer to the KeyMap in which to store the supplied values.

key
A character string to be stored with the values, which can later be used to identify the values. Trailing spaces are ignored. The supplied string is converted to upper case before use if the KeyCase attribute is currently set to zero.

size
The number of elements in the supplied array of values.

value
The array of values to be stored. The data type of this value should match the 1-character type code appended to the function name (e.g. if you are using astMapPut1A, the type of this value should be "array of pointers to AstObject").

comment

A pointer to a null-terminated comment string to be stored with the values. A NULL pointer may be supplied, in which case no comment is stored.

Notes:

- If the supplied key is already in use in the KeyMap, the new values will replace the old values.

Data Type Codes:

To select the appropriate function, you should replace <X> in the generic function name astMapPut1<X> with a 1-character data type code, so as to match the data type <X>type of the data you are processing, as follows:

- D: double
- F: float
- I: int
- C: "const" pointer to null terminated character string
- A: Pointer to AstObject
- P: Generic "void *" pointer
- S: short int
- B: Unsigned byte (i.e. char)

For example, astMapPut1D would be used to store "double" values, while astMapPut1I would be used to store "int", etc.

Note that KeyMaps containing generic "void *" pointers cannot be written out using astShow or astWrite. An error will be reported if this is attempted.

---

astMapPutElem<X> Put a value into an element of a vector value in a KeyMap astMapPutElem<X>

Description: This is a set of functions for storing a value in a single element of a vector value in a KeyMap. You should replace <X> in the generic function name astMapPutElem<X> by an appropriate 1-character type code (see the "Data Type Codes" section below for the code appropriate to each supported data type). The supplied value is converted from the data type indicated by <X> to the data type of the KeyMap entry before being stored (an error is reported if it is not possible to convert the value to the required data type).

Synopsis: `void astMapPutElem<X>( AstKeyMap *this, const char *key, int elem, <X>type *value )`

Parameters:

this

Pointer to the KeyMap.

key

The character string identifying the value to be retrieved. Trailing spaces are ignored. The supplied string is converted to upper case before use if the KeyCase attribute is currently set to zero.

elem

The index of the vector element to modify, starting at zero.

value

The value to store.

Class Applicability:

KeyMap

If the "elem" index is outside the range of the vector, the length of the vector will be increased by one element and the supplied value will be stored at the end of the vector in the new element.

Table

If the "elem" index is outside the range of the vector, an error will be reported. The number of elements in each cell of a column is specified when the column is created using astAddColumn.

Notes:

- If the entry originally holds a scalar value, it will be treated like a vector entry of length 1.
- If the specified key cannot be found in the given KeyMap, or is found but has an undefined value, a new vector entry with the given name, and data type implied by <X>, is created and the supplied value is stored in its first entry.

Data Type Codes:

To select the appropriate function, you should replace <X> in the generic function name astMapPutElem<X> with a 1-character data type code, so as to match the data type <X>type of the data you are processing, as follows:

- D: double
- F: float
- I: int
- C: "const" pointer to null terminated character string
- A: Pointer to AstObject
- P: Generic "void *" pointer
- S: short int
- B: Unsigned byte (i.e. char)

For example, astMapPutElemD would be used to put a "double" value, while astMapPutElemI would be used to put an "int" value, etc. For D or I, the supplied "value" parameter should be a double or int. For C, the supplied "value" parameter should be a pointer to a character string. For A, the supplied "value" parameter should be an AstObject pointer.

---

astMapPutU — Add an entry to a KeyMap with an undefined value — astMapPutU

Description: This function adds a new entry to a KeyMap, but no value is stored with the entry. The entry therefore has a special data type represented by symbolic constant AST__UNDEFTYPE.

An example use is to add entries with undefined values to a KeyMap prior to locking them with the MapLocked attribute. Such entries can act as placeholders for values that can be added to the KeyMap later.

Synopsis: `void astMapPutU( AstKeyMap *this, const char *key, const char *comment );`

Parameters:

this

Pointer to the KeyMap in which to store the supplied value.

key

A character string to be stored with the value, which can later be used to identify the value. Trailing spaces are ignored. The supplied string is converted to upper case before use if the KeyCase attribute is currently set to zero.

comment

A pointer to a null-terminated comment string to be stored with the value. A NULL pointer may be supplied, in which case no comment is stored.

Notes:

- If the supplied key is already in use in the KeyMap, the value associated with the key will be removed.

---

astMapRegion Transform a Region into a new Frame using a given Mapping astMapRegion

Description: This function returns a pointer to a new Region which corresponds to supplied Region described by some other specified coordinate system. A Mapping is supplied which transforms positions between the old and new coordinate systems. The new Region may not be of the same class as the original region.

Synopsis: `AstRegion *astMapRegion( AstRegion *this, AstMapping *map, AstFrame *frame )`

Parameters:

this

Pointer to the Region.

map

Pointer to a Mapping which transforms positions from the coordinate system represented by the supplied Region to the coordinate system specified by "frame". The supplied Mapping should define both forward and inverse transformations, and these transformations should form a genuine inverse pair. That is, transforming a position using the forward transformation and then using the inverse transformation should produce the original input position. Some Mapping classes (such as PermMap, MathMap, SphMap) can result in Mappings for which this is not true.

frame

Pointer to a Frame describing the coordinate system in which the new Region is required.

Returned Value:

astMapRegion()

A pointer to a new Region. This Region will represent the area within the coordinate system specified by "frame" which corresponds to the supplied Region.

Notes:

- The uncertainty associated with the supplied Region is modified using the supplied Mapping.
- A null Object pointer (AST__NULL) will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

astMapRemove Removed a named entry from a KeyMap astMapRemove

Description: This function removes a named entry from a KeyMap. It returns without action if the KeyMap does not contain the specified key.

Synopsis: `void astMapRemove( AstKeyMap *this, const char *key )`

Parameters:

this

Pointer to the KeyMap.

key

The character string identifying the value to be retrieved. Trailing spaces are ignored. The supplied string is converted to upper case before use if the KeyCase attribute is currently set to zero.

astMapRename Rename an existing KeyMap entry astMapRename

Description: This function associated a new key with an existing entry in a KeyMap. It returns without action if the oldkey does not exist in the KeyMap.

Synopsis: `void astMapRename( AstKeyMap *this, const char *oldkey, const char *newkey )`

Parameters:

this

Pointer to the KeyMap.

oldkey

The character string identifying the entry to be renamed. Trailing spaces are ignored. The supplied string is converted to upper case before use if the KeyCase attribute is currently set to zero.

newkey

The new character string to associated with the renamed entry. Trailing spaces are ignored. The supplied string is converted to upper case before use if the KeyCase attribute is currently set to zero.

astMapSize Get the number of entries in a KeyMap astMapSize

Description: This function returns the number of entries in a KeyMap.

Synopsis: `int astMapSize( AstKeyMap *this )`

Parameters:

this

Pointer to the KeyMap.

Returned Value:

astMapSize()

The number of entries in the KeyMap.

Notes:

- A function value of zero will be returned if an error has already occurred, or if this function should fail for any reason.

---

astMapSplit Split a Mapping up into parallel component Mappings astMapSplit

Description: This function creates a new Mapping which connects specified inputs within a supplied Mapping to the corresponding outputs of the supplied Mapping. This is only possible if the specified inputs correspond to some subset of the Mapping outputs. That is, there must exist a subset of the Mapping outputs for which each output depends only on the selected Mapping inputs, and not on any of the inputs which have not been selected. Also, any output which is not in this subset must not depend on any of the selected inputs. If these conditions are not met by the supplied Mapping, then a NULL Mapping pointer is returned.

Synopsis: `void astMapSplit( AstMapping *this, int nin, const int *in, int *out, AstMapping **map )`

Parameters:

this

Pointer to the Mapping to be split.

nin

The number of inputs to pick from "this".

in

Pointer to an array holding the indices within the supplied Mapping of the inputs which are to be picked from the Mapping. This array should have "nin" elements. If "Nin" is the number of inputs of the supplied Mapping, then each element should have a value in the range 1 to Nin.

out

Pointer to an array in which to return the indices of the outputs of the supplied Mapping which are fed by the picked inputs. A value of one is used to refer to the first Mapping output. The supplied array should have a length at least equal to the number of outputs in the supplied Mapping. The number of values stored in the array on exit will equal the number of outputs in the returned Mapping. The i'th element in the returned array holds the index within the supplied Mapping which corresponds to the i'th output of the returned Mapping.

map

Address of a location at which to return a pointer to the returned Mapping. This Mapping will have "nin" inputs (the number of outputs may be different to "nin"). NULL is returned if the supplied Mapping has no subset of outputs which depend only on the selected inputs. The returned Mapping is a deep copy of the required parts of the supplied Mapping.

Notes:

- If this function is invoked with the global error status set, or if it should fail for any reason, then a NULL value will be returned for the "map" pointer.

---

astMapType Get the data type of an entry in a KeyMap astMapType

Description: This function returns a value indicating the data type of a named entry in a KeyMap. This is the data type which was used when the entry was added to the KeyMap.

Synopsis: `int astMapType( AstKeyMap *this, const char *key )`

Parameters:

this
Pointer to the KeyMap.

key
The character string identifying the KeyMap entry. Trailing spaces are ignored. The supplied string is converted to upper case before use if the KeyCase attribute is currently set to zero.

Returned Value:

astMapType()
One of AST__INTTYPE (for integer), AST__SINTTYPE (for short int), AST__BYTETYPE (for unsigned bytes

- i.e. unsigned chars) AST__DOUBLETYPE (for double precision floating point), AST__FLOATTYPE (for single precision floating point), AST__STRINGTYPE (for character string), AST__OBJECTTYPE (for AST Object pointer), AST__POINTERTYPE (for arbitrary C pointer) or AST__UNDEFTYPE (for undefined values created by astMapPutU). AST__BADTYPE is returned if the supplied key is not found in the KeyMap.

Notes:

- A function value of AST__BADTYPE will be returned if an error has already occurred, or if this function should fail for any reason.

---

astMark Draw a set of markers for a Plot astMark

Description: This function draws a set of markers (symbols) at positions specified in the physical coordinate system of a Plot. The positions are transformed into graphical coordinates to determine where the markers should appear within the plotting area.

Synopsis: `void astMark( AstPlot *this, int nmark, int ncoord, int indim, const double *in, int type )`

Parameters:

this
Pointer to the Plot.

nmark
The number of markers to draw. This may be zero, in which case nothing will be drawn.

ncoord
The number of coordinates being supplied for each mark (i.e. the number of axes in the current Frame of the Plot, as given by its Naxes attribute).

indim
The number of elements along the second dimension of the "in" array (which contains the marker coordinates). This value is required so that the coordinate values can be correctly located if they do not entirely fill this array. The value given should not be less than "nmark".

in
The address of the first element of a 2-dimensional array of shape "[ncoord][indim]" giving the physical coordinates of the points where markers are to be drawn. These should be stored such that the value of coordinate number "coord" for input mark number "mark" is found in element "in[coord][mark]".

type
A value specifying the type (e.g. shape) of marker to be drawn. The set of values which may be used (and the shapes that will result) is determined by the underlying graphics system.

Notes:

- Markers are not drawn at positions which have any coordinate equal to the value AST__BAD (or where the transformation into graphical coordinates yields coordinates containing the value AST__BAD).
- If any marker position is clipped (see astClip), then the entire marker is not drawn.
- An error results if the base Frame of the Plot is not 2-dimensional.
- An error also results if the transformation between the current and base Frames of the Plot is not defined (i.e. the Plot's TranInverse attribute is zero).

---

astMask<X> Mask a region of a data grid astMask<X>

Description: This is a set of functions for masking out regions within gridded data (e.g. an image). The functions modifies a given data grid by assigning a specified value to all samples which are inside (or outside if "inside" is zero) the specified Region.

You should use a masking function which matches the numerical type of the data you are processing by replacing <X> in the generic function name astMask<X> by an appropriate 1- or 2-character type code. For example, if you are masking data with type "float", you should use the function astMaskF (see the "Data Type Codes" section below for the codes appropriate to other numerical types).

Synopsis: `int astMask<X>( AstRegion *this, AstMapping *map, int inside, int ndim, const int lbnd[], const int ubnd[], <Xtype> in[], <Xtype> val )`

Parameters:

this

Pointer to a Region.

map

Pointer to a Mapping. The forward transformation should map positions in the coordinate system of the supplied Region into pixel coordinates as defined by the "lbnd" and "ubnd" parameters. A NULL pointer can be supplied if the coordinate system of the supplied Region corresponds to pixel coordinates. This is equivalent to supplying a UnitMap.

The number of inputs for this Mapping (as given by its Nin attribute) should match the number of axes in the supplied Region (as given by the Naxes attribute of the Region). The number of outputs for the Mapping (as given by its Nout attribute) should match the number of grid dimensions given by the value of "ndim" below.

inside

A boolean value which indicates which pixel are to be masked. If a non-zero value is supplied, then all grid pixels with centres inside the supplied Region are assigned the value given by "val", and all other pixels are left unchanged. If zero is supplied, then all grid pixels with centres not inside the supplied Region are assigned the value given by "val", and all other pixels are left unchanged. Note, the Negated attribute of the Region is used to determine which pixel are inside the Region and which are outside. So the inside of a Region which has not been negated is the same as the outside of the corresponding negated Region.

For types of Region such as PointList which have zero volume, pixel centres will rarely fall exactly within the Region. For this reason, the inclusion criterion is changed for zero-volume Regions so that pixels are included (or excluded) if any part of the Region passes through the pixel. For a PointList, this means that pixels are included (or excluded) if they contain at least one of the points listed in the PointList.

ndim

The number of dimensions in the input grid. This should be at least one.

lbnd

Pointer to an array of integers, with "ndim" elements, containing the coordinates of the centre of the first pixel in the input grid along each dimension.

ubnd

Pointer to an array of integers, with "ndim" elements, containing the coordinates of the centre of the last pixel in the input grid along each dimension.

Note that "lbnd" and "ubnd" together define the shape and size of the input grid, its extent along a particular (j'th) dimension being ubnd[j]-lbnd[j]+1 (assuming the index "j" to be zero-based). They also define the input grid's coordinate system, each pixel having unit extent along each dimension with integral coordinate values at its centre.

in

Pointer to an array, with one element for each pixel in the input grid, containing the data to be masked. The numerical type of this array should match the 1- or 2-character type code appended to the function name (e.g. if you are using astMaskF, the type of each array element should be "float").

The storage order of data within this array should be such that the index of the first grid dimension varies most rapidly and that of the final dimension least rapidly (i.e. Fortran array indexing is used).

On exit, the samples specified by "inside" are set to the value of "val". All other samples are left unchanged.

val

This argument should have the same type as the elements of the "in" array. It specifies the value used to flag the masked data (see "inside").

Returned Value:

astMask<X>()

The number of pixels to which a value of "badval" has been assigned.

Notes:

- A value of zero will be returned if this function is invoked with the global error status set, or if it should fail for any reason.
- An error will be reported if the overlap of the Region and the array cannot be determined.

Data Type Codes:

To select the appropriate masking function, you should replace <X> in the generic function name astMask<X> with a 1- or 2-character data type code, so as to match the numerical type <Xtype> of the data you are processing, as follows:

- D: double
- F: float
- L: long int
- UL: unsigned long int
- I: int
- UI: unsigned int
- S: short int
- US: unsigned short int
- B: byte (signed char)
- UB: unsigned byte (unsigned char)

For example, astMaskD would be used to process "double" data, while astMaskS would be used to process "short int" data, etc.

---

astMatchAxes Find any corresponding axes in two Frames astMatchAxes

Description: This function looks for corresponding axes within two supplied Frames. An array of integers is returned that contains an element for each axis in the second supplied Frame. An element in this array will be set to zero if the associated axis within the second Frame has no corresponding axis within the first Frame. Otherwise, it will be set to the index (a non-zero positive integer) of the corresponding axis within the first supplied Frame.

Synopsis: `void astMatchAxes( AstFrame *frm1, AstFrame *frm2, int *axes )`

Parameters:

frm1

Pointer to the first Frame.

frm2

Pointer to the second Frame.

axes

Pointer to an integer array in which to return the indices of the axes (within the first Frame) that correspond to each axis within the second Frame. Axis indices start at 1. A value of zero will be stored in the returned array for each axis in the second Frame that has no corresponding axis in the first Frame.

The number of elements in this array must be greater than or equal to the number of axes in the second Frame.

Class Applicability:

Frame

This function applies to all Frames.

Notes:

- Corresponding axes are identified by the fact that a Mapping can be found between them using astFindFrame or astConvert. Thus, "corresponding axes" are not necessarily identical. For instance, SkyFrame axes in two Frames will match even if they describe different celestial coordinate systems

---

astMathMap Create a MathMap astMathMap

Description: This function creates a new MathMap and optionally initialises its attributes.

A MathMap is a Mapping which allows you to specify a set of forward and/or inverse transformation functions using arithmetic operations and mathematical functions similar to those available in C. The MathMap interprets these functions at run-time, whenever its forward or inverse transformation is required. Because the functions are not compiled in the normal sense (unlike an IntraMap), they may be used to describe coordinate transformations in a transportable manner. A MathMap therefore provides a flexible way of defining new types of Mapping whose descriptions may be stored as part of a dataset and interpreted by other programs.

Synopsis: `AstMathMap *astMathMap( int nin, int nout, int nfwd, const char *fwd[], int ninv, const char *inv[], const char *options, ... )`

Parameters:

nin

Number of input variables for the MathMap. This determines the value of its Nin attribute.

nout

Number of output variables for the MathMap. This determines the value of its Nout attribute.

nfwd

The number of forward transformation functions being supplied. This must be at least equal to "nout", but may be increased to accommodate any additional expressions which define intermediate variables for the forward transformation (see the "Calculating Intermediate Values" section below).

fwd

An array (with "nfwd" elements) of pointers to null terminated strings which contain the expressions defining the forward transformation. The syntax of these expressions is described below.

ninv

The number of inverse transformation functions being supplied. This must be at least equal to "nin", but may be increased to accommodate any additional expressions which define intermediate variables for the inverse transformation (see the "Calculating Intermediate Values" section below).

inv

An array (with "ninv" elements) of pointers to null terminated strings which contain the expressions defining the inverse transformation. The syntax of these expressions is described below.

options

Pointer to a null-terminated string containing an optional comma-separated list of attribute assignments to be used for initialising the new MathMap. The syntax used is identical to that for the astSet function and may include "printf" format specifiers identified by "%" symbols in the normal way. If no initialisation is required, a zero-length string may be supplied.

...

If the "options" string contains "%" format specifiers, then an optional list of additional arguments may follow it in order to supply values to be substituted for these specifiers. The rules for supplying these are identical to those for the astSet function (and for the C "printf" function).

Returned Value:

astMathMap()

A pointer to the new MathMap.

Notes:

- The sequence of numbers produced by the random number functions available within a MathMap is normally unpredictable and different for each MathMap. However, this behaviour may be controlled by means of the MathMap's Seed attribute.
- Normally, compound Mappings (CmpMaps) which involve MathMaps will not be subject to simplification (e.g. using astSimplify) because AST cannot know how different MathMaps will interact. However, in the special case where a MathMap occurs in series with its own inverse, then simplification may be possible. Whether simplification does, in fact, occur under these circumstances is controlled by the MathMap's SimpFI and SimpIF attributes.
- A null Object pointer (AST__NULL) will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

Defining Transformation Functions:

A MathMap's transformation functions are supplied as a set of expressions in an array of character strings. Normally you would supply the same number of expressions for the forward transformation, via the "fwd" parameter, as there are output variables (given by the MathMap's Nout attribute). For instance, if Nout is 2 you might use:

- "r = sqrt(x ∗ x + y ∗ y)"
- "theta = atan2(y, x)"

which defines a transformation from Cartesian to polar coordinates. Here, the variables that appear on the left of each expression ("r" and "theta") provide names for the output variables and those that appear on the right ("x" and "y") are references to input variables.

To complement this, you must also supply expressions for the inverse transformation via the "inv" parameter. In this case, the number of expressions given would normally match the number of MathMap input coordinates (given by the Nin attribute). If Nin is 2, you might use:

- "x = r ∗ cos(theta)"
- "y = r ∗ sin(theta)"

which expresses the transformation from polar to Cartesian coordinates. Note that here the input variables ("x" and "y") are named on the left of each expression, and the output variables ("r" and "theta") are referenced on the right.

Normally, you cannot refer to a variable on the right of an expression unless it is named on the left of an expression in the complementary set of functions. Therefore both sets of functions (forward and inverse) must be formulated using the same consistent set of variable names. This means that if you wish to leave one of the transformations undefined, you must supply dummy expressions which simply name each of the output (or input) variables. For example, you might use:

- "x"
- "y"

for the inverse transformation above, which serves to name the input variables but without defining an inverse transformation.

Calculating Intermediate Values:

It is sometimes useful to calculate intermediate values and then to use these in the final expressions for the output (or input) variables. This may be done by supplying additional expressions for the forward (or inverse) transformation functions. For instance, the following array of five expressions describes 2-dimensional pin-cushion distortion:

- "r = sqrt(xin ∗ xin + yin ∗ yin)"
- "rout = r ∗ (1 + 0.1 ∗ r ∗ r)"
- "theta = atan2(yin, xin)"
- "xout = rout ∗ cos(theta)"
- "yout = rout ∗ sin(theta)"

Here, we first calculate three intermediate results ("r", "rout" and "theta") and then use these to calculate the final results ("xout" and "yout"). The MathMap knows that only the final two results constitute values for the output variables because its Nout attribute is set to 2. You may define as many intermediate variables in this way as you choose. Having defined a variable, you may then refer to it on the right of any subsequent expressions.

Note that when defining the inverse transformation you may only refer to the output variables "xout" and "yout". The intermediate variables "r", "rout" and "theta" (above) are private to the forward transformation and may not be referenced by the inverse transformation. The inverse transformation may, however, define its own private intermediate variables.

Expression Syntax:

The expressions given for the forward and inverse transformations closely follow the syntax of the C programming language (with some extensions for compatibility with Fortran). They may contain references to variables and literal constants, together with arithmetic, boolean, relational and bitwise operators, and function invocations. A set of symbolic constants is also available. Each of these is described in detail below. Parentheses may be used to over-ride the normal order of evaluation. There is no built-in limit to the length of expressions and they are insensitive to case or the presence of additional white space.

Variables:

Variable names must begin with an alphabetic character and may contain only alphabetic characters, digits, and the underscore character "_". There is no built-in limit to the length of variable names.

Literal Constants:

Literal constants, such as "0", "1", "0.007" or "2.505e-16" may appear in expressions, with the decimal point and exponent being optional (a "D" may also be used as an exponent character for compatibility with Fortran). A unary minus "-" may be used as a prefix.

Arithmetic Precision:

All arithmetic is floating point, performed in double precision.

Propagation of Missing Data:

Unless indicated otherwise, if any argument of a function or operator has the value AST__BAD (indicating missing data), then the result of that function or operation is also AST__BAD, so that such values are propagated automatically through all operations performed by MathMap transformations. The special value AST__BAD can be represented in expressions by the symbolic constant "<bad>".

A <bad> result (i.e. equal to AST__BAD) is also produced in response to any numerical error (such as division by zero or numerical overflow), or if an invalid argument value is provided to a function or operator.

Arithmetic Operators:

The following arithmetic operators are available:

- x1 + x2: Sum of "x1" and "x2".
- x1 - x2: Difference of "x1" and "x2".
- x1 ∗ x2: Product of "x1" and "x1".
- x1 / x2: Ratio of "x1" and "x2".
- x1 ∗∗ x2: "x1" raised to the power of "x2".
- + x: Unary plus, has no effect on its argument.
- - x: Unary minus, negates its argument.

Boolean Operators:

Boolean values are represented using zero to indicate false and non-zero to indicate true. In addition, the value AST__BAD is taken to mean "unknown". The values returned by boolean operators may therefore be 0, 1 or AST__BAD. Where appropriate, "tri-state" logic is implemented. For example, "a||b" may evaluate to 1 if "a" is non-zero, even if "b" has the value AST__BAD. This is because the result of the operation would not be affected by the value of "b", so long as "a" is non-zero.

The following boolean operators are available:

- x1 && x2: Boolean AND between "x1" and "x2", returning 1 if both "x1" and "x2" are non-zero, and 0 otherwise. This operator implements tri-state logic. (The synonym ".and." is also provided for compatibility with Fortran.)
- x1 || x2: Boolean OR between "x1" and "x2", returning 1 if either "x1" or "x2" are non-zero, and 0 otherwise. This operator implements tri-state logic. (The synonym ".or." is also provided for compatibility with Fortran.)
- x1 ∧∧ x2: Boolean exclusive OR (XOR) between "x1" and "x2", returning 1 if exactly one of "x1" and "x2" is non-zero, and 0 otherwise. Tri-state logic is not used with this operator. (The synonyms ".neqv." and ".xor." are also provided for compatibility with Fortran, although the second of these is not standard.)
- x1 .eqv. x2: This is provided only for compatibility with Fortran and tests whether the boolean states of "x1" and "x2" (i.e. true/false) are equal. It is the negative of the exclusive OR (XOR) function. Tri-state logic is not used with this operator.
- ! x: Boolean unary NOT operation, returning 1 if "x" is zero, and 0 otherwise. (The synonym ".not." is also provided for compatibility with Fortran.)

Relational Operators:

Relational operators return the boolean result (0 or 1) of comparing the values of two floating point values for equality or inequality. The value AST__BAD may also be returned if either argument is <bad>.

The following relational operators are available:

- x1 == x2: Tests whether "x1" equals "x1". (The synonym ".eq." is also provided for compatibility with Fortran.)
- x1 != x2: Tests whether "x1" is unequal to "x2". (The synonym ".ne." is also provided for compatibility with Fortran.)
- x1 > x2: Tests whether "x1" is greater than "x2". (The synonym ".gt." is also provided for compatibility with Fortran.)
- x1 >= x2: Tests whether "x1" is greater than or equal to "x2". (The synonym ".ge." is also provided for compatibility with Fortran.)
- x1 < x2: Tests whether "x1" is less than "x2". (The synonym ".lt." is also provided for compatibility with Fortran.)
- x1 <= x2: Tests whether "x1" is less than or equal to "x2". (The synonym ".le." is also provided for compatibility with Fortran.)

Note that relational operators cannot usefully be used to compare values with the <bad> value (representing missing data), because the result is always <bad>. The isbad() function should be used instead.

Bitwise Operators:

The bitwise operators provided by C are often useful when operating on raw data (e.g. from instruments), so they are also provided for use in MathMap expressions. In this case, however, the values on which they operate are floating point values rather than pure integers. In order to produce results which match the pure integer case, the operands are regarded as fixed point binary numbers (i.e. with the binary equivalent of a decimal point) with negative numbers represented using twos-complement notation. For integer values, the resulting bit pattern corresponds to that of the equivalent signed integer (digits to the right of the point being zero). Operations on the bits representing the fractional part are also possible, however.

The following bitwise operators are available:

- x1 >> x2: Rightward bit shift. The integer value of "x2" is taken (rounding towards zero) and the bits representing "x1" are then shifted this number of places to the right (or to the left if the number of places is negative). This is equivalent to dividing "x1" by the corresponding power of 2.

- x1 << x2: Leftward bit shift. The integer value of "x2" is taken (rounding towards zero), and the bits representing "x1" are then shifted this number of places to the left (or to the right if the number of places is negative). This is equivalent to multiplying "x1" by the corresponding power of 2.
- x1 & x2: Bitwise AND between the bits of "x1" and those of "x2" (equivalent to a boolean AND applied at each bit position in turn).
- x1 | x2: Bitwise OR between the bits of "x1" and those of "x2" (equivalent to a boolean OR applied at each bit position in turn).
- x1 ∧ x2: Bitwise exclusive OR (XOR) between the bits of "x1" and those of "x2" (equivalent to a boolean XOR applied at each bit position in turn).

Note that no bit inversion operator ("∼" in C) is provided. This is because inverting the bits of a twos-complement fixed point binary number is equivalent to simply negating it. This differs from the pure integer case because bits to the right of the binary point are also inverted. To invert only those bits to the left of the binary point, use a bitwise exclusive OR with the value -1 (i.e. "x∧-1").

Functions:

The following functions are available:

- abs(x): Absolute value of "x" (sign removal), same as fabs(x).
- acos(x): Inverse cosine of "x", in radians.
- acosd(x): Inverse cosine of "x", in degrees.
- acosh(x): Inverse hyperbolic cosine of "x".
- acoth(x): Inverse hyperbolic cotangent of "x".
- acsch(x): Inverse hyperbolic cosecant of "x".
- aint(x): Integer part of "x" (round towards zero), same as int(x).
- asech(x): Inverse hyperbolic secant of "x".
- asin(x): Inverse sine of "x", in radians.
- asind(x): Inverse sine of "x", in degrees.
- asinh(x): Inverse hyperbolic sine of "x".
- atan(x): Inverse tangent of "x", in radians.
- atand(x): Inverse tangent of "x", in degrees.
- atanh(x): Inverse hyperbolic tangent of "x".
- atan2(x1, x2): Inverse tangent of "x1/x2", in radians.
- atan2d(x1, x2): Inverse tangent of "x1/x2", in degrees.
- ceil(x): Smallest integer value not less then "x" (round towards plus infinity).
- cos(x): Cosine of "x" in radians.
- cosd(x): Cosine of "x" in degrees.
- cosh(x): Hyperbolic cosine of "x".
- coth(x): Hyperbolic cotangent of "x".
- csch(x): Hyperbolic cosecant of "x".
- dim(x1, x2): Returns "x1-x2" if "x1" is greater than "x2", otherwise 0.
- exp(x): Exponential function of "x".
- fabs(x): Absolute value of "x" (sign removal), same as abs(x).
- floor(x): Largest integer not greater than "x" (round towards minus infinity).
- fmod(x1, x2): Remainder when "x1" is divided by "x2", same as mod(x1, x2).

- gauss(x1, x2): Random sample from a Gaussian distribution with mean "x1" and standard deviation "x2".
- int(x): Integer part of "x" (round towards zero), same as aint(x).
- isbad(x): Returns 1 if "x" has the <bad> value (AST__BAD), otherwise 0.
- log(x): Natural logarithm of "x".
- log10(x): Logarithm of "x" to base 10.
- max(x1, x2, ...): Maximum of two or more values.
- min(x1, x2, ...): Minimum of two or more values.
- mod(x1, x2): Remainder when "x1" is divided by "x2", same as fmod(x1, x2).
- nint(x): Nearest integer to "x" (round to nearest).
- poisson(x): Random integer-valued sample from a Poisson distribution with mean "x".
- pow(x1, x2): "x1" raised to the power of "x2".
- qif(x1, x2, x3): Returns "x2" if "x1" is true, and "x3" otherwise.
- rand(x1, x2): Random sample from a uniform distribution in the range "x1" to "x2" inclusive.
- sech(x): Hyperbolic secant of "x".
- sign(x1, x2): Absolute value of "x1" with the sign of "x2" (transfer of sign).
- sin(x): Sine of "x" in radians.
- sinc(x): Sinc function of "x" [= "sin(x)/x"].
- sind(x): Sine of "x" in degrees.
- sinh(x): Hyperbolic sine of "x".
- sqr(x): Square of "x" (= "x*x").
- sqrt(x): Square root of "x".
- tan(x): Tangent of "x" in radians.
- tand(x): Tangent of "x" in degrees.
- tanh(x): Hyperbolic tangent of "x".

Symbolic Constants:

The following symbolic constants are available (the enclosing "<>" brackets must be included):

- <bad>: The "bad" value (AST__BAD) used to flag missing data. Note that you cannot usefully compare values with this constant because the result is always <bad>. The isbad() function should be used instead.
- <dig>: Number of decimal digits of precision available in a floating point (double) value.
- <e>: Base of natural logarithms.
- <epsilon>: Smallest positive number such that 1.0+<epsilon> is distinguishable from unity.
- <mant_dig>: The number of base <radix> digits stored in the mantissa of a floating point (double) value.
- <max>: Maximum representable floating point (double) value.
- <max_10_exp>: Maximum integer such that 10 raised to that power can be represented as a floating point (double) value.
- <max_exp>: Maximum integer such that <radix> raised to that power minus 1 can be represented as a floating point (double) value.
- <min>: Smallest positive number which can be represented as a normalised floating point (double) value.

- <min_10_exp>: Minimum negative integer such that 10 raised to that power can be represented as a normalised floating point (double) value.
- <min_exp>: Minimum negative integer such that <radix> raised to that power minus 1 can be represented as a normalised floating point (double) value.
- <pi>: Ratio of the circumference of a circle to its diameter.
- <radix>: The radix (number base) used to represent the mantissa of floating point (double) values.
- <rounds>: The mode used for rounding floating point results after addition. Possible values include: -1 (indeterminate), 0 (toward zero), 1 (to nearest), 2 (toward plus infinity) and 3 (toward minus infinity). Other values indicate machine-dependent behaviour.

Evaluation Precedence and Associativity:

Items appearing in expressions are evaluated in the following order (highest precedence first):

- Constants and variables
- Function arguments and parenthesised expressions
- Function invocations
- Unary + - ! .not.
- **
- * /
- + -
- << >>
- < .lt. <= .le. > .gt. >= .ge.
- == .eq. != .ne.
- &
- ^
- |
- && .and.
- ^^
- || .or
- .eqv. .neqv. .xor.

All operators associate from left-to-right, except for unary +, unary -, !, .not. and ** which associate from right-to-left.

---

astMatrixMap Create a MatrixMap astMatrixMap

Description: This function creates a new MatrixMap and optionally initialises its attributes.

A MatrixMap is a form of Mapping which performs a general linear transformation. Each set of input coordinates, regarded as a column-vector, are pre-multiplied by a matrix (whose elements are specified when the MatrixMap is created) to give a new column-vector containing the output coordinates. If appropriate, the inverse transformation may also be performed.

Synopsis: `AstMatrixMap *astMatrixMap( int nin, int nout, int form, const double matrix[], const char *options, ... )`

Parameters:

nin

The number of input coordinates, which determines the number of columns in the matrix.

nout

The number of output coordinates, which determines the number of rows in the matrix.

form

An integer which indicates the form in which the matrix elements will be supplied.

A value of zero indicates that a full "nout" x "nin" matrix of values will be supplied via the "matrix" parameter (below). In this case, the elements should be given in row order (the elements of the first row, followed by the elements of the second row, etc.).

A value of 1 indicates that only the diagonal elements of the matrix will be supplied, and that all others should be zero. In this case, the elements of "matrix" should contain only the diagonal elements, stored consecutively.

A value of 2 indicates that a "unit" matrix is required, whose diagonal elements are set to unity (with all other elements zero). In this case, the "matrix" parameter is ignored and a NULL pointer may be supplied.

matrix

The array of matrix elements to be used, stored according to the value of "form".

options

Pointer to a null-terminated string containing an optional comma-separated list of attribute assignments to be used for initialising the new MatrixMap. The syntax used is identical to that for the astSet function and may include "printf" format specifiers identified by "%" symbols in the normal way.

...

If the "options" string contains "%" format specifiers, then an optional list of additional arguments may follow it in order to supply values to be substituted for these specifiers. The rules for supplying these are identical to those for the astSet function (and for the C "printf" function).

Returned Value:

astMatrixMap()

A pointer to the new MatrixMap.

Notes:

- In general, a MatrixMap's forward transformation will always be available (as indicated by its TranForward attribute), but its inverse transformation (TranInverse attribute) will only be available if the associated matrix is square and non-singular.
- As an exception to this, the inverse transformation is always available if a unit or diagonal matrix is specified. In this case, if the matrix is not square, one or more of the input coordinate values may not be recoverable from a set of output coordinates. Any coordinates affected in this way will simply be set to the value zero.
- A null Object pointer (AST__NULL) will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

Status Handling:

The protected interface to this function includes an extra parameter at the end of the parameter list descirbed above. This parameter is a pointer to the integer inherited status variable: "int ∗status".

astMirrorVariants Make the current Frame mirror the variant Mappings in another Frame astMirrorVariants

Description: This function indicates that all access to the Variant attribute of the current Frame should should be forwarded to some other nominated Frame in the FrameSet. For instance, if a value is set subsequently for the Variant attribute of the current Frame, the current Frame will be left unchanged and the setting is instead applied to the nominated Frame. Likewise, if the value of the Variant attribute is requested, the value returned is the value stored for the nominated Frame rather than the current Frame itself.

This provides a mechanism for propagating the effects of variant Mappings around a FrameSet. If a new Frame is added to a FrameSet by connecting it to an pre-existing Frame that has two or more variant Mappings, then it may be appropriate to set the new Frame so that it mirrors the variants Mappings of the pre-existing Frame. If this is done, then it will be possible to select a specific variant Mapping using either the pre-existing Frame or the new Frame.

Synopsis: `void astMirrorVariants( AstFrameSet *this, int iframe, int *status )`

Parameters:

this

Pointer to the FrameSet.

iframe

The index of the Frame within the FrameSet which is to be mirrored by the current Frame. This value should lie in the range from 1 to the number of Frames in the FrameSet (as given by its Nframe attribute). If AST__NOFRAME is supplied (or the current Frame is specified), then any mirroring established by a previous call to this function is disabled.

Notes:

- Mirrors can be chained. That is, if Frame B is set to be a mirror of Frame A, and Frame C is set to be a mirror of Frame B, then Frame C will act as a mirror of Frame A.
- Variant Mappings cannot be added to the current Frame if it is mirroring another Frame. So calls to the astAddVariant function will cause an error to be reported if the current Frame is mirroring another Frame.
- A value of AST__BASE may be given for the "iframe" parameter to specify the base Frame.
- Any variant Mappings explicitly added to the current Frame using astAddVariant will be ignored if the current Frame is mirroring another Frame.

astNegate Negate the area represented by a Region astNegate

Description: This function negates the area represented by a Region. That is, points which were previously inside the region will then be outside, and points which were outside will be inside. This is acomplished by toggling the state of the Negated attribute for the supplied region.

Synopsis: `void astNegate( AstRegion *this )`

Parameters:

this

Pointer to the Region.

---

astNorm Normalise a set of Frame coordinates astNorm

Description: This function normalises a set of Frame coordinate values which might be unsuitable for display (e.g. may lie outside the expected range) into a set of acceptable values suitable for display.

Synopsis: `void astNorm( AstFrame *this, double value[] )`

Parameters:

this

Pointer to the Frame.

value

An array of double, with one element for each Frame axis (Naxes attribute). Initially, this should contain a set of coordinate values representing a point in the space which the Frame describes. If these values lie outside the expected range for the Frame, they will be replaced with more acceptable (normalised) values. Otherwise, they will be returned unchanged.

Notes:

- For some classes of Frame, whose coordinate values are not constrained, this function will never modify the values supplied. However, for Frames whose axes represent cyclic quantities (such as angles or positions on the sky), coordinates will typically be wrapped into an appropriate standard range, such as zero to 2*pi.
- The NormMap class is a Mapping which can be used to normalise a set of points using the astNorm function of a specified Frame.
- It is intended to be possible to put any set of coordinates into a form suitable for display by using this function to normalise them, followed by appropriate formatting (using astFormat).

---

astNormMap Create a NormMap astNormMap

Description: This function creates a new NormMap and optionally initialises its attributes.

A NormMap is a Mapping which normalises coordinate values using the astNorm function of the supplied Frame. The number of inputs and outputs of a NormMap are both equal to the number of axes in the supplied Frame.

The forward and inverse transformation of a NormMap are both defined but are identical (that is, they do not form a real inverse pair in that the inverse transformation does not undo the normalisation, instead it reapplies it). However, the astSimplify function will replace neighbouring pairs of forward and inverse NormMaps by a single UnitMap.

Synopsis: `AstNormMap *astNormMap( AstFrame *frame, const char *options, ... )`

Parameters:

frame

A pointer to the Frame which is to be used to normalise the supplied axis values.

options

Pointer to a null-terminated string containing an optional comma-separated list of attribute assignments to be used for initialising the new NormMap. The syntax used is identical to that for the astSet function and may include "printf" format specifiers identified by "%" symbols in the normal way.

...

If the "options" string contains "%" format specifiers, then an optional list of additional arguments may follow it in order to supply values to be substituted for these specifiers. The rules for supplying these are identical to those for the astSet function (and for the C "printf" function).

Returned Value:

astNormMap()

A pointer to the new NormMap.

Notes:

- A null Object pointer (AST__NULL) will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

Status Handling:

The protected interface to this function includes an extra parameter at the end of the parameter list descirbed above. This parameter is a pointer to the integer inherited status variable: "int *status".

---

astNullRegion Create a NullRegion astNullRegion

Description: This function creates a new NullRegion and optionally initialises its attributes.

A NullRegion is a Region with no bounds. If the Negated attribute of a NullRegion is false, the NullRegion represents a Region containing no points. If the Negated attribute of a NullRegion is true, the NullRegion represents an infinite Region containing all points within the coordinate system.

Synopsis: `AstNullRegion *astNullRegion( AstFrame *frame, AstRegion *unc, const char *options, ... )`

Parameters:

frame

A pointer to the Frame in which the region is defined. A deep copy is taken of the supplied Frame. This means that any subsequent changes made to the Frame using the supplied pointer will have no effect the Region.

unc

An optional pointer to an existing Region which specifies the uncertainties associated with positions in the supplied Frame. The uncertainty in any point in the Frame is found by shifting the supplied "uncertainty" Region so that it is centred at the point being considered. The area covered by the shifted uncertainty Region then represents the uncertainty in the position. The uncertainty is assumed to be the same for all points.

If supplied, the uncertainty Region must be of a class for which all instances are centro-symetric (e.g. Box, Circle, Ellipse, etc.) or be a Prism containing centro-symetric component Regions. A deep copy of the supplied Region will be taken, so subsequent changes to the uncertainty Region using the supplied pointer will have no effect on the created Box. Alternatively, a NULL Object pointer may be supplied, in which case a default uncertainty of zero is used.

options

Pointer to a null-terminated string containing an optional comma-separated list of attribute assignments to be used for initialising the new NullRegion. The syntax used is identical to that for the astSet function and may include "printf" format specifiers identified by "%" symbols in the normal way.

...

If the "options" string contains "%" format specifiers, then an optional list of additional arguments may follow it in order to supply values to be substituted for these specifiers. The rules for supplying these are identical to those for the astSet function (and for the C "printf" function).

Returned Value:

astNullRegion()
A pointer to the new NullRegion.

Notes:

- A null Object pointer (AST__NULL) will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

---

astOK Test whether AST functions have been successful astOK

Description: This macro returns a boolean value (0 or 1) to indicate if preceding AST functions have completed successfully (i.e. without setting the AST error status). If the error status is set to an error value, a value of zero is returned, otherwise the result is one.

Synopsis: `int astOK`

Returned Value:

astOK
One if the AST error status is OK, otherwise zero.

Notes:

- If the AST error status is set to an error value (after an error), most AST functions will not execute and will simply return without action. To clear the error status and restore normal behaviour, use astClearStatus.

---

astOffset Calculate an offset along a geodesic curve astOffset

Description: This function finds the Frame coordinate values of a point which is offset a specified distance along the geodesic curve between two other points.

For example, in a basic Frame, this offset will be along the straight line joining two points. For a more specialised Frame describing a sky coordinate system, however, it would be along the great circle passing through two sky positions.

Synopsis:

```
void astOffset( AstFrame *this, const double point1[], const double point2[],
double offset, double point3[] )
```

Parameters:

this
Pointer to the Frame.

point1
An array of double, with one element for each Frame axis (Naxes attribute). This should contain the coordinates of the point marking the start of the geodesic curve.

point2
An array of double, with one element for each Frame axis This should contain the coordinates of the point marking the end of the geodesic curve.

offset
The required offset from the first point along the geodesic curve. If this is positive, it will be towards the second point. If it is negative, it will be in the opposite direction. This offset need not imply a position lying between the two points given, as the curve will be extrapolated if necessary.

point3

An array of double, with one element for each Frame axis in which the coordinates of the required point will be returned.

Notes:

- The geodesic curve used by this function is the path of shortest distance between two points, as defined by the astDistance function.
- This function will return "bad" coordinate values (AST__BAD) if any of the input coordinates has this value.
- "Bad" coordinate values will also be returned if the two points supplied are coincident (or otherwise fail to uniquely specify a geodesic curve) but the requested offset is non-zero.

---

astOffset2 Calculate an offset along a geodesic curve in a 2D Frame astOffset2

Description: This function finds the Frame coordinate values of a point which is offset a specified distance along the geodesic curve at a given angle from a specified starting point. It can only be used with 2-dimensional Frames.

For example, in a basic Frame, this offset will be along the straight line joining two points. For a more specialised Frame describing a sky coordinate system, however, it would be along the great circle passing through two sky positions.

Synopsis: `double astOffset2( AstFrame *this, const double point1[2], double angle, double offset, double point2[2] );`

Parameters:

this

Pointer to the Frame.

point1

An array of double, with one element for each Frame axis (Naxes attribute). This should contain the coordinates of the point marking the start of the geodesic curve.

angle

The angle (in radians) from the positive direction of the second axis, to the direction of the required position, as seen from the starting position. Positive rotation is in the sense of rotation from the positive direction of axis 2 to the positive direction of axis 1.

offset

The required offset from the first point along the geodesic curve. If this is positive, it will be in the direction of the given angle. If it is negative, it will be in the opposite direction.

point2

An array of double, with one element for each Frame axis in which the coordinates of the required point will be returned.

Returned Value:

astOffset2

The direction of the geodesic curve at the end point. That is, the angle (in radians) between the positive direction of the second axis and the continuation of the geodesic curve at the requested end point. Positive rotation is in the sense of rotation from the positive direction of axis 2 to the positive direction of axis 1.

Notes:

- The geodesic curve used by this function is the path of shortest distance between two points, as defined by the astDistance function.
- An error will be reported if the Frame is not 2-dimensional.
- This function will return "bad" coordinate values (AST__BAD) if any of the input coordinates has this value.

---

astOutline<X> Create a new Polygon outling values in a 2D data grid astOutline<X>

Description: This is a set of functions that create a Polygon enclosing a single contiguous set of pixels that have a specified value within a gridded 2-dimensional data array (e.g. an image).

A basic 2-dimensional Frame is used to represent the pixel coordinate system in the returned Polygon. The Domain attribute is set to "PIXEL", the Title attribute is set to "Pixel coordinates", and the Unit attribute for each axis is set to "pixel". All other attributes are left unset. The nature of the pixel coordinate system is determined by parameter "starpix".

The "maxerr" and "maxvert" parameters can be used to control how accurately the returned Polygon represents the required region in the data array. The number of vertices in the returned Polygon will be the minimum needed to achieve the required accuracy.

You should use a function which matches the numerical type of the data you are processing by replacing <X> in the generic function name astOutline<X> by an appropriate 1- or 2-character type code. For example, if you are procesing data with type "float", you should use the function astOutlineF (see the "Data Type Codes" section below for the codes appropriate to other numerical types).

Synopsis: `AstPolygon *astOutline<X>( <Xtype> value, int oper, const <Xtype> array[], const int lbnd[2], const int ubnd[2], double maxerr, int maxvert, const int inside[2], int starpix )`

Parameters:

value

A data value that specifies the pixels to be outlined.

oper

Indicates how the "value" parameter is used to select the outlined pixels. It can have any of the following values:

- AST__LT: outline pixels with value less than "value".
- AST__LE: outline pixels with value less than or equal to "value".
- AST__EQ: outline pixels with value equal to "value".
- AST__NE: outline pixels with value not equal to "value".
- AST__GE: outline pixels with value greater than or equal to "value".
- AST__GT: outline pixels with value greater than "value".

array

Pointer to a 2-dimensional array containing the data to be processed. The numerical type of this array should match the 1- or 2-character type code appended to the function name (e.g. if you are using astOutlineF, the type of each array element should be "float").

The storage order of data within this array should be such that the index of the first grid dimension varies most rapidly and that of the second dimension least rapidly (i.e. Fortran array indexing is used).

lbnd

Pointer to an array of two integers containing the coordinates of the centre of the first pixel in the input grid along each dimension.

ubnd

Pointer to an array of two integers containing the coordinates of the centre of the last pixel in the input grid along each dimension.

Note that "lbnd" and "ubnd" together define the shape and size of the input grid, its extent along a particular (j'th) dimension being ubnd[j]-lbnd[j]+1 (assuming the index "j" to be zero-based). They also define the input grid's coordinate system, each pixel having unit extent along each dimension with integral coordinate values at its centre or upper corner, as selected by parameter "starpix".

maxerr

Together with "maxvert", this determines how accurately the returned Polygon represents the required region of the data array. It gives the target discrepancy between the returned Polygon and the accurate outline in the data array, expressed as a number of pixels. Insignificant vertices are removed from the accurate outline, one by one, until the number of vertices remaining in the returned Polygon equals "maxvert", or the largest discrepancy between the accurate outline and the returned Polygon is greater than "maxerr". If "maxerr" is zero or less, its value is ignored and the returned Polygon will have the number of vertices specified by "maxvert".

maxvert

Together with "maxerr", this determines how accurately the returned Polygon represents the required region of the data array. It gives the maximum allowed number of vertices in the returned Polygon. Insignificant vertices are removed from the accurate outline, one by one, until the number of vertices remaining in the returned Polygon equals "maxvert", or the largest discrepancy between the accurate outline and the returned Polygon is greater than "maxerr". If "maxvert" is less than 3, its value is ignored and the number of vertices in the returned Polygon will be the minimum needed to ensure that the discrepancy between the accurate outline and the returned Polygon is less than "maxerr".

inside

Pointer to an array of two integers containing the indices of a pixel known to be inside the required region. This is needed because the supplied data array may contain several disjoint areas of pixels that satisfy the criterion specified by "value" and "oper". In such cases, the area described by the returned Polygon will be the one that contains the pixel specified by "inside". If the specified pixel is outside the bounds given by "lbnd" and "ubnd", or has a value that does not meet the criterion specified by "value" and "oper", then this function will search for a suitable pixel. The search starts at the central pixel and proceeds in a spiral manner until a pixel is found that meets the specified crierion.

starpix

A flag indicating the nature of the pixel coordinate system used to describe the vertex positions in the returned Polygon. If non-zero, the standard Starlink definition of pixel coordinate is used in which a pixel with integer index I spans a range of pixel coordinate from (I-1) to I (i.e. pixel corners have integral pixel coordinates). If zero, the definition of pixel coordinate used by other AST functions such as astResample, astMask, etc., is used. In this definition, a pixel with integer index I spans a range of pixel coordinate from (I-0.5) to (I+0.5) (i.e. pixel centres have integral pixel coordinates).

boxsize

The full width in pixels of a smoothing box to be applied to the polygon vertices before downsizing the polygon to a smaller number of vertices. If an even number is supplied, the next larger odd number is used. Values of one or zero result in no smoothing.

Returned Value:

astOutline<X>()

The number of pixels to which a value of "badval" has been assigned.

Notes:

- This function proceeds by first finding a very accurate polygon, and then removing insignificant vertices from this fine polygon using astDownsize.
- The returned Polygon is the outer boundary of the contiguous set of pixels that includes ths specified "inside" point, and satisfy the specified value requirement. This set of pixels may potentially include "holes" where the pixel values fail to meet the specified value requirement. Such holes will be ignored by this function.
- A value of zero will be returned if this function is invoked with the global error status set, or if it should fail for any reason.

Data Type Codes:

To select the appropriate masking function, you should replace <X> in the generic function name astOutline<X> with a 1- or 2-character data type code, so as to match the numerical type <Xtype> of the data you are processing, as follows:

- D: double
- F: float
- L: long int
- UL: unsigned long int
- I: int
- UI: unsigned int
- S: short int
- US: unsigned short int
- B: byte (signed char)
- UB: unsigned byte (unsigned char)

For example, astOutlineD would be used to process "double" data, while astOutlineS would be used to process "short int" data, etc.

---

astOverlap Test if two regions overlap each other astOverlap

Description: This function returns an integer value indicating if the two supplied Regions overlap. The two Regions are converted to a commnon coordinate system before performing the check. If this conversion is not possible (for instance because the two Regions represent areas in different domains), then the check cannot be performed and a zero value is returned to indicate this.

Synopsis: `int astOverlap( AstRegion *this, AstRegion *that )`

Parameters:

this

Pointer to the first Region.

that

Pointer to the second Region.

Returned Value:

astOverlap()

A value indicating if there is any overlap between the two Regions. Possible values are:

0 - The check could not be performed because the second Region could not be mapped into the coordinate system of the first Region.

1 - There is no overlap between the two Regions.

2 - The first Region is completely inside the second Region.

3 - The second Region is completely inside the first Region.

4 - There is partial overlap between the two Regions.

5 - The Regions are identical to within their uncertainties.

6 - The second Region is the exact negation of the first Region to within their uncertainties.

Notes:

- The returned values 5 and 6 do not check the value of the Closed attribute in the two Regions.
- A value of zero will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

---

astParameterName Get the name of the global parameter at a given index within the Table astParameterName

Description: This function returns a string holding the name of the global parameter with the given index within the Table.

This function is intended primarily as a means of iterating round all the parameters in a Table. For this purpose, the number of parameters in the Table is given by the Nparameter attribute of the Table. This function could then be called in a loop, with the index value going from zero to one less than Nparameter.

Note, the index associated with a parameter decreases monotonically with the age of the parameter: the oldest Parameter in the Table will have index one, and the Parameter added most recently to the Table will have the largest index.

Synopsis: `const char *astParameterName( AstTable *this, int index )`

Parameters:

this
Pointer to the Table.

index
The index into the list of parameters. The first parameter has index one, and the last has index "Nparameter".

Returned Value:

astParameterName()
A pointer to a null-terminated string containing the upper case parameter name.

Notes:

- The returned pointer is guaranteed to remain valid and the string to which it points will not be over-written for a total of 50 successive invocations of this function. After this, the memory containing the string may be re-used, so a copy of the string should be made if it is needed for longer than this.
- A NULL pointer will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

---

astPcdMap Create a PcdMap astPcdMap

Description: This function creates a new PcdMap and optionally initialises its attributes.

A PcdMap is a non-linear Mapping which transforms 2-dimensional positions to correct for the radial distortion introduced by some cameras and telescopes. This can take the form either of pincushion or barrel distortion, and is characterized by a single distortion coefficient.

A PcdMap is specified by giving this distortion coefficient and the coordinates of the centre of the radial distortion. The forward transformation of a PcdMap applies the distortion:

RD = R ∗ (1 + C ∗ R ∗ R)

where R is the undistorted radial distance from the distortion centre (specified by attribute PcdCen), RD is the radial distance from the same centre in the presence of distortion, and C is the distortion coefficient (given by attribute Disco).

The inverse transformation of a PcdMap removes the distortion produced by the forward transformation. The expression used to derive R from RD is an approximate inverse of the expression above, obtained from two iterations of the Newton-Raphson method. The mismatch between the forward and inverse expressions is negligible for astrometric applications (to reach 1 milliarcsec at the edge of the Anglo-Australian Telescope triplet or a Schmidt field would require field diameters of 2.4 and 42 degrees respectively).

If a PcdMap is inverted (e.g. using astInvert) then the roles of the forward and inverse transformations are reversed; the forward transformation will remove the distortion, and the inverse transformation will apply it.

Synopsis: `AstPcdMap *astPcdMap( double disco, const double pcdcen[2], const char *options, ... )`

Parameters:

disco

The distortion coefficient. Negative values give barrel distortion, positive values give pincushion distortion, and zero gives no distortion.

pcdcen

A 2-element array containing the coordinates of the centre of the distortion.

options

Pointer to a null-terminated string containing an optional comma-separated list of attribute assignments to be used for initialising the new PcdMap. The syntax used is identical to that for the astSet function and may include "printf" format specifiers identified by "%" symbols in the normal way.

...

If the "options" string contains "%" format specifiers, then an optional list of additional arguments may follow it in order to supply values to be substituted for these specifiers. The rules for supplying these are identical to those for the astSet function (and for the C "printf" function).

Returned Value:

astPcdMap()

A pointer to the new PcdMap.

Notes:

- A null Object pointer (AST__NULL) will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

Status Handling:

The protected interface to this function includes an extra parameter at the end of the parameter list descirbed above. This parameter is a pointer to the integer inherited status variable: "int *status".

---

astPermAxes Permute the axis order in a Frame astPermAxes

Description: This function permutes the order in which a Frame's axes occur.

Synopsis: `void astPermAxes( AstFrame *this, const int perm[] )`

Parameters:

this
Pointer to the Frame.

perm
An array with one element for each axis of the Frame (Naxes attribute). This should list the axes in their new order, using the original axis numbering (which starts at 1 for the first axis).

Notes:

- Only genuine permutations of the axis order are permitted, so each axis must be referenced exactly once in the "perm" array.
- If successive axis permutations are applied to a Frame, then the effects are cumulative.

---

astPermMap Create a PermMap astPermMap

Description: This function creates a new PermMap and optionally initialises its attributes.

A PermMap is a Mapping which permutes the order of coordinates, and possibly also changes the number of coordinates, between its input and output.

In addition to permuting the coordinate order, a PermMap may also assign constant values to coordinates. This is useful when the number of coordinates is being increased as it allows fixed values to be assigned to any new ones.

Synopsis: `AstPermMap *astPermMap( int nin, const int inperm[], int nout, const int outperm[], double constant[], const char *options, ... )`

Parameters:

nin
The number of input coordinates.

inperm
An optional array with "nin" elements which, for each input coordinate, should contain the number of the output coordinate whose value is to be used (note that this array therefore defines the inverse coordinate transformation). Coordinates are numbered starting from 1.

For details of additional special values that may be used in this array, see the description of the "constant" parameter.

If a NULL pointer is supplied instead of an array, each input coordinate will obtain its value from the corresponding output coordinate (or will be assigned the value AST__BAD if there is no corresponding output coordinate).

nout
The number of output coordinates.

outperm

An optional array with "nout" elements which, for each output coordinate, should contain the number of the input coordinate whose value is to be used (note that this array therefore defines the forward coordinate transformation). Coordinates are numbered starting from 1.

For details of additional special values that may be used in this array, see the description of the "constant" parameter.

If a NULL pointer is supplied instead of an array, each output coordinate will obtain its value from the corresponding input coordinate (or will be assigned the value AST__BAD if there is no corresponding input coordinate).

constant

An optional array containing values which may be assigned to input and/or output coordinates instead of deriving them from other coordinate values. If either of the "inperm" or "outperm" arrays contains a negative value, it is used to address this "constant" array (such that -1 addresses the first element, -2 addresses the second element, etc.) and the value obtained is used as the corresponding coordinate value.

Care should be taken to ensure that locations lying outside the extent of this array are not accidentally addressed. The array is not used if the "inperm" and "outperm" arrays do not contain negative values.

If a NULL pointer is supplied instead of an array, the behaviour is as if the array were of infinite length and filled with the value AST__BAD.

options

Pointer to a null-terminated string containing an optional comma-separated list of attribute assignments to be used for initialising the new PermMap. The syntax used is identical to that for the astSet function and may include "printf" format specifiers identified by "%" symbols in the normal way.

...

If the "options" string contains "%" format specifiers, then an optional list of additional arguments may follow it in order to supply values to be substituted for these specifiers. The rules for supplying these are identical to those for the astSet function (and for the C "printf" function).

Returned Value:

astPermMap()

A pointer to the new PermMap.

Notes:

- If either of the "inperm" or "outperm" arrays contains a zero value (or a positive value which does not identify a valid output/input coordinate, as appropriate), then the value AST__BAD is assigned as the new coordinate value.
- This function does not attempt to ensure that the forward and inverse transformations performed by the PermMap are self-consistent in any way. You are therefore free to supply coordinate permutation arrays that achieve whatever effect is desired.
- A null Object pointer (AST__NULL) will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

---

astPickAxes Create a new Frame by picking axes from an existing one astPickAxes

Description: This function creates a new Frame whose axes are copied from an existing Frame along with other Frame attributes, such as its Title. Any number (zero or more) of the original Frame's axes may be copied, in any order, and additional axes with default attributes may also be included in the new Frame.

Optionally, a Mapping that converts between the coordinate systems described by the two Frames will also be returned.

Synopsis: `AstFrame *astPickAxes( AstFrame *this, int naxes, const int axes[], AstMapping **map )`

Parameters:

this

Pointer to the original Frame.

naxes

The number of axes required in the new Frame.

axes

An array, with "naxes" elements, which lists the axes to be copied. These should be given in the order required in the new Frame, using the axis numbering in the original Frame (which starts at 1 for the first axis). Axes may be selected in any order, but each may only be used once. If additional (default) axes are also to be included, the corresponding elements of this array should be set to zero.

map

Address of a location in which to return a pointer to a new Mapping. This will be a PermMap (or a UnitMap as a special case) that describes the axis permutation that has taken place between the original and new Frames. The Mapping's forward transformation will convert coordinates from the original Frame into the new one, and vice versa.

If this Mapping is not required, a NULL value may be supplied for this parameter.

Class Applicability:

Frame

This function applies to all Frames. The class of Frame returned may differ from that of the original Frame, depending on which axes are selected. For example, if a single axis is picked from a SkyFrame (which must always have two axes) then the resulting Frame cannot be a valid SkyFrame, so will revert to the parent class (Frame) instead.

FrameSet

Using this function on a FrameSet is identical to using it on the current Frame in the FrameSet. The returned Frame will not be a FrameSet.

Region

If this function is used on a Region, an attempt is made to retain the bounds information on the selected axes. If succesful, the returned Frame will be a Region of some class. Otherwise, the returned Frame is obtained by calling this function on the Frame represented by the supplied Region (the returned Frame will then not be a Region). In order to be succesful, the selected axes in the Region must be independent of the others. For instance, a Box can be split in this way but a Circle cannot. Another requirement for success is that no default axes are added (that is, the "axes" array must not contain any zero values.

Returned Value:

astPickAxes()

A pointer to the new Frame.

Notes:

- The new Frame will contain a "deep" copy (c.f. astCopy) of all the data selected from the original Frame. Modifying any aspect of the new Frame will therefore not affect the original one.

- A null Object pointer (AST__NULL) will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

---

astPlot Create a Plot astPlot

Description: This function creates a new Plot and optionally initialises its attributes.

A Plot is a specialised form of FrameSet, in which the base Frame describes a "graphical" coordinate system and is associated with a rectangular plotting area in the underlying graphics system. This plotting area is where graphical output appears. It is defined when the Plot is created.

The current Frame of a Plot describes a "physical" coordinate system, which is the coordinate system in which plotting operations are specified. The results of each plotting operation are automatically transformed into graphical coordinates so as to appear in the plotting area (subject to any clipping which may be in effect).

Because the Mapping between physical and graphical coordinates may often be non-linear, or even discontinuous, most plotting does not result in simple straight lines. The basic plotting element is therefore not a straight line, but a geodesic curve (see astCurve). A Plot also provides facilities for drawing markers or symbols (astMark), text (astText) and grid lines (astGridLine). It is also possible to draw curvilinear axes with optional coordinate grids (astGrid). A range of Plot attributes is available to allow precise control over the appearance of graphical output produced by these functions.

You may select different physical coordinate systems in which to plot (including the native graphical coordinate system itself) by selecting different Frames as the current Frame of a Plot, using its Current attribute. You may also set up clipping (see astClip) to limit the extent of any plotting you perform, and this may be done in any of the coordinate systems associated with the Plot, not necessarily the one you are plotting in.

Like any FrameSet, a Plot may also be used as a Frame. In this case, it behaves like its current Frame, which describes the physical coordinate system.

When used as a Mapping, a Plot describes the inter-relation between graphical coordinates (its base Frame) and physical coordinates (its current Frame). It differs from a normal FrameSet, however, in that an attempt to transform points which lie in clipped areas of the Plot will result in bad coordinate values (AST__BAD).

Synopsis: `AstPlot *astPlot( AstFrame *frame, const float graphbox[ 4 ], const double basebox[ 4 ], const char *options, ... )`

Parameters:

frame

Pointer to a Frame describing the physical coordinate system in which to plot. A pointer to a FrameSet may also be given, in which case its current Frame will be used to define the physical coordinate system and its base Frame will be mapped on to graphical coordinates (see below).

If a null Object pointer (AST__NULL) is given, a default 2-dimensional Frame will be used to describe the physical coordinate system. Labels, etc. may then be attached to this by setting the appropriate Frame attributes (e.g. Label(axis)) for the Plot.

graphbox

An array giving the position and extent of the plotting area (on the plotting surface of the underlying graphics system) in which graphical output is to appear. This must be specified using graphical coordinates appropriate to the underlying graphics system.

The first pair of values should give the coordinates of the bottom left corner of the plotting area and the second pair should give the coordinates of the top right corner. The coordinate on the horizontal axis should be given first in each pair. Note that the order in which these points are given is important because it defines up, down, left and right for subsequent graphical operations.

basebox

An array giving the coordinates of two points in the supplied Frame (or in the base Frame if a FrameSet was supplied) which correspond to the bottom left and top right corners of the plotting area, as specified above. This range of coordinates will be mapped linearly on to the plotting area. The coordinates should be given in the same order as above.

options

Pointer to a null-terminated string containing an optional comma-separated list of attribute assignments to be used for initialising the new Plot. The syntax used is identical to that for the astSet function and may include "printf" format specifiers identified by "%" symbols in the normal way. If no initialisation is required, a zero-length string may be supplied.

...

If the "options" string contains "%" format specifiers, then an optional list of additional arguments may follow it in order to supply values to be substituted for these specifiers. The rules for supplying these are identical to those for the astSet function (and for the C "printf" function).

Returned Value:

astPlot()

A pointer to the new Plot.

Notes:

- The base Frame of the returned Plot will be a new Frame which is created by this function to represent the coordinate system of the underlying graphics system (graphical coordinates). It is given a Frame index of 1 within the Plot. The choice of base Frame (Base attribute) should not, in general, be changed once a Plot has been created (although you could use this as a way of moving the plotting area around on the plotting surface).
- If a Frame is supplied (via the "frame" pointer), then it becomes the current Frame of the new Plot and is given a Frame index of 2.
- If a FrameSet is supplied (via the "frame" pointer), then all the Frames within this FrameSet become part of the new Plot (where their Frame indices are increased by 1), with the FrameSet's current Frame becoming the current Frame of the Plot.
- If a null Object pointer (AST__NULL) is supplied (via the "frame" pointer), then the returned Plot will contain two Frames, both created by this function. The base Frame will describe graphics coordinates (as above) and the current Frame will be a basic Frame with no attributes set (this will therefore give default values for such things as the Plot Title and the Label on each axis). Physical coordinates will be mapped linearly on to graphical coordinates.
- An error will result if the Frame supplied (or the base Frame if a FrameSet was supplied) is not 2-dimensional.
- A null Object pointer (AST__NULL) will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

---

astPlot3D Create a Plot3D astPlot3D

Description: This function creates a new Plot3D and optionally initialises its attributes.

A Plot3D is a specialised form of Plot that provides facilities for producing 3D graphical output.

Synopsis: `AstPlot3D *astPlot3D( AstFrame *frame, const float graphbox[ 6 ], const double basebox[ 6 ], const char *options, ... )`

Parameters:

frame

Pointer to a Frame describing the physical coordinate system in which to plot. A pointer to a FrameSet may also be given, in which case its current Frame will be used to define the physical coordinate system and its base Frame will be mapped on to graphical coordinates (see below).

If a null Object pointer (AST__NULL) is given, a default 3-dimensional Frame will be used to describe the physical coordinate system. Labels, etc. may then be attached to this by setting the appropriate Frame attributes (e.g. Label(axis)) for the Plot.

graphbox

An array giving the position and extent of the plotting volume (within the plotting space of the underlying graphics system) in which graphical output is to appear. This must be specified using graphical coordinates appropriate to the underlying graphics system.

The first triple of values should give the coordinates of the bottom left corner of the plotting volume and the second triple should give the coordinates of the top right corner. The coordinate on the horizontal axis should be given first in each pair. Note that the order in which these points are given is important because it defines up, down, left and right for subsequent graphical operations.

basebox

An array giving the coordinates of two points in the supplied Frame (or in the base Frame if a FrameSet was supplied) which correspond to the bottom left and top right corners of the plotting volume, as specified above. This range of coordinates will be mapped linearly on to the plotting area. The coordinates should be given in the same order as above.

options

Pointer to a null-terminated string containing an optional comma-separated list of attribute assignments to be used for initialising the new Plot3D. The syntax used is identical to that for the astSet function and may include "printf" format specifiers identified by "%" symbols in the normal way. If no initialisation is required, a zero-length string may be supplied.

...

If the "options" string contains "%" format specifiers, then an optional list of additional arguments may follow it in order to supply values to be substituted for these specifiers. The rules for supplying these are identical to those for the astSet function (and for the C "printf" function).

Returned Value:

astPlot3D()

A pointer to the new Plot3D.

Notes:

- The base Frame of the returned Plot3D will be a new Frame which is created by this function to represent the coordinate system of the underlying graphics system (graphical coordinates). It is given a Frame index of 1 within the Plot3D. The choice of base Frame (Base attribute) should not, in general, be changed once a Plot3D has been created (although you could use this as a way of moving the plotting area around on the plotting surface).
- If a Frame is supplied (via the "frame" pointer), then it becomes the current Frame of the new Plot3D and is given a Frame index of 2.
- If a FrameSet is supplied (via the "frame" pointer), then all the Frames within this FrameSet become part of the new Plot3D (where their Frame indices are increased by 1), with the FrameSet's current Frame becoming the current Frame of the Plot3D.
- At least one of the three axes of the current Frame must be independent of the other two current Frame axes.

- If a null Object pointer (AST__NULL) is supplied (via the "frame" pointer), then the returned Plot3D will contain two Frames, both created by this function. The base Frame will describe graphics coordinates (as above) and the current Frame will be a basic Frame with no attributes set (this will therefore give default values for such things as the Plot3D Title and the Label on each axis). Physical coordinates will be mapped linearly on to graphical coordinates.
- An error will result if the Frame supplied (or the base Frame if a FrameSet was supplied) is not 3-dimensional.
- A null Object pointer (AST__NULL) will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

astPointList Create a PointList astPointList

Description: This function creates a new PointList object and optionally initialises its attributes.

A PointList object is a specialised type of Region which represents a collection of points in a coordinate Frame.

Synopsis: `AstPointList *astPointList( AstFrame *frame, int npnt, int ncoord, int dim, const double *points, AstRegion *unc, const char *options, ... )`

Parameters:

frame

A pointer to the Frame in which the region is defined. A deep copy is taken of the supplied Frame. This means that any subsequent changes made to the Frame using the supplied pointer will have no effect the Region.

npnt

The number of points in the Region.

ncoord

The number of coordinates being supplied for each point. This must equal the number of axes in the supplied Frame, given by its Naxes attribute.

dim

The number of elements along the second dimension of the "points" array (which contains the point coordinates). This value is required so that the coordinate values can be correctly located if they do not entirely fill this array. The value given should not be less than "npnt".

points

The address of the first element of a 2-dimensional array of shape "[ncoord][dim]" giving the physical coordinates of the points. These should be stored such that the value of coordinate number "coord" for point number "pnt" is found in element "in[coord][pnt]".

unc

An optional pointer to an existing Region which specifies the uncertainties associated with each point in the PointList being created. The uncertainty at any point in the PointList is found by shifting the supplied "uncertainty" Region so that it is centred at the point being considered. The area covered by the shifted uncertainty Region then represents the uncertainty in the position. The uncertainty is assumed to be the same for all points.

If supplied, the uncertainty Region must be of a class for which all instances are centro-symetric (e.g. Box, Circle, Ellipse, etc.) or be a Prism containing centro-symetric component Regions. A deep copy of the supplied Region will be taken, so subsequent changes to the uncertainty Region using the supplied pointer will have no effect on the created Box. Alternatively, a NULL Object pointer may be supplied, in which case a default uncertainty is used equivalent to a box 1.0E-6 of the size of the bounding box of the PointList being created.

The uncertainty Region has two uses: 1) when the astOverlap function compares two Regions for equality the uncertainty Region is used to determine the tolerance on the comparison, and

2) when a Region is mapped into a different coordinate system and subsequently simplified (using astSimplify), the uncertainties are used to determine if the transformed boundary can be accurately represented by a specific shape of Region.

options

Pointer to a null-terminated string containing an optional comma-separated list of attribute assignments to be used for initialising the new PointList. The syntax used is identical to that for the astSet function and may include "printf" format specifiers identified by "%" symbols in the normal way.

...

If the "options" string contains "%" format specifiers, then an optional list of additional arguments may follow it in order to supply values to be substituted for these specifiers. The rules for supplying these are identical to those for the astSet function (and for the C "printf" function).

Returned Value:

astPointList()

A pointer to the new PointList.

Notes:

- A null Object pointer (AST__NULL) will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

Status Handling:

The protected interface to this function includes an extra parameter at the end of the parameter list descirbed above. This parameter is a pointer to the integer inherited status variable: "int *status".

---

astPolyCurve Draw a series of connected geodesic curves astPolyCurve

Description: This function joins a series of points specified in the physical coordinate system of a Plot by drawing a sequence of geodesic curves. It is equivalent to making repeated use of the astCurve function (q.v.), except that astPolyCurve will generally be more efficient when drawing many geodesic curves end-to-end. A typical application of this might be in drawing contour lines.

As with astCurve, full account is taken of the Mapping between physical and graphical coordinate systems. This includes any discontinuities and clipping established using astClip.

Synopsis: `void astPolyCurve( AstPlot *this, int npoint, int ncoord, int indim, const double *in )`

Parameters:

this

Pointer to the Plot.

npoint

The number of points between which geodesic curves are to be drawn.

ncoord

The number of coordinates being supplied for each point (i.e. the number of axes in the current Frame of the Plot, as given by its Naxes attribute).

indim

The number of elements along the second dimension of the "in" array (which contains the input coordinates). This value is required so that the coordinate values can be correctly located if they do not entirely fill this array. The value given should not be less than "npoint".

in

The address of the first element in a 2-dimensional array of shape "[ncoord][indim]" giving the physical coordinates of the points which are to be joined in sequence by geodesic curves. These should be stored such that the value of coordinate number "coord" for point number "point" is found in element "in[coord][point]".

Notes:

- No curve is drawn on either side of any point which has any coordinate equal to the value AST__BAD.
- An error results if the base Frame of the Plot is not 2-dimensional.
- An error also results if the transformation between the current and base Frames of the Plot is not defined (i.e. the Plot's TranInverse attribute is zero).

---

astPolyMap Create a PolyMap astPolyMap

Description: This function creates a new PolyMap and optionally initialises its attributes.

A PolyMap is a form of Mapping which performs a general polynomial transformation. Each output coordinate is a polynomial function of all the input coordinates. The coefficients are specified separately for each output coordinate. The forward and inverse transformations are defined independantly by separate sets of coefficients. If no inverse transformation is supplied, an iterative method can be used to evaluate the inverse based only on the forward transformation.

Synopsis: `AstPolyMap *astPolyMap( int nin, int nout, int ncoeff_f, const double coeff_f[], int ncoeff_i, const double coeff_i[], const char *options, ... )`

Parameters:

nin

The number of input coordinates.

nout

The number of output coordinates.

ncoeff_f

The number of non-zero coefficients necessary to define the forward transformation of the PolyMap. If zero is supplied, the forward transformation will be undefined.

coeff_f

An array containing "ncoeff_f*(2 + nin)" elements. Each group of "2 + nin" adjacent elements describe a single coefficient of the forward transformation. Within each such group, the first element is the coefficient value; the next element is the integer index of the PolyMap output which uses the coefficient within its defining polynomial (the first output has index 1); the remaining elements of the group give the integer powers to use with each input coordinate value (powers must not be negative, and floating point values are rounded to the nearest integer). If "ncoeff_f" is zero, a NULL pointer may be supplied for "coeff_f".

For instance, if the PolyMap has 3 inputs and 2 outputs, each group consisting of 5 elements, A groups such as "(1.2, 2.0, 1.0, 3.0, 0.0)" describes a coefficient with value 1.2 which is used within the definition of output 2. The output value is incremented by the product of the coefficient value, the value of input coordinate 1 raised to the power 1, and the value of input coordinate 2 raised to the power 3. Input coordinate 3 is not used since its power is specified as zero. As another example, the group "(-1.0, 1.0, 0.0, 0.0, 0.0)" describes adds a constant value -1.0 onto output 1 (it is a constant value since the power for every input axis is given as zero).

Each final output coordinate value is the sum of the "ncoeff_f" terms described by the "ncoeff_f" groups within the supplied array.

ncoeff_i

The number of non-zero coefficients necessary to define the inverse transformation of the PolyMap. If zero is supplied, the inverse transformation will be undefined.

coeff_i

An array containing "ncoeff_i∗(2 + nout)" elements. Each group of "2 + nout" adjacent elements describe a single coefficient of the inverse transformation, using the same schame as "coeff_f", except that "inputs" and "outputs" are transposed. If "ncoeff_i" is zero, a NULL pointer may be supplied for "coeff_i".

options

Pointer to a null-terminated string containing an optional comma-separated list of attribute assignments to be used for initialising the new PolyMap. The syntax used is identical to that for the astSet function and may include "printf" format specifiers identified by "%" symbols in the normal way.

...

If the "options" string contains "%" format specifiers, then an optional list of additional arguments may follow it in order to supply values to be substituted for these specifiers. The rules for supplying these are identical to those for the astSet function (and for the C "printf" function).

Returned Value:

astPolyMap()

A pointer to the new PolyMap.

Notes:

- A null Object pointer (AST__NULL) will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

astPolyTran Fit a PolyMap inverse or forward transformation astPolyTran

Description: This function creates a new PolyMap which is a copy of the supplied PolyMap, in which a specified transformation (forward or inverse) has been replaced by a new polynomial transformation. The coefficients of the new transformation are estimated by sampling the other transformation and performing a least squares polynomial fit in the opposite direction to the sampled positions and values.

This method can only be used on (1-input,1-output) or (2-input,2-output) PolyMaps.

The transformation to create is specified by the "forward" parameter. In what follows "X" refers to the inputs of the PolyMap, and "Y" to the outputs of the PolyMap. The forward transformation transforms input values (X) into output values (Y), and the inverse transformation transforms output values (Y) into input values (X). Within a PolyMap, each transformation is represented by an independent set of polynomials, P_f or P_i: Y=P_f(X) for the forward transformation and X=P_i(Y) for the inverse transformation.

The "forward" parameter specifies the transformation to be replaced. If it is non-zero, a new forward transformation is created by first finding the input values (X) using the inverse transformation (which must be available) at a regular grid of points (Y) covering a rectangular region of the PolyMap's output space. The coefficients of the required forward polynomial, Y=P_f(X), are chosen in order to minimise the sum of the squared residuals between the sampled values of Y and P_f(X).

If "forward" is zero (probably the most likely case), a new inverse transformation is created by first finding the output values (Y) using the forward transformation (which must be available) at a regular grid of points (X) covering a rectangular region of the PolyMap's input space. The coefficients of the required inverse polynomial, X=P_i(Y), are chosen in order to minimise the sum of the squared residuals between the sampled values of X and P_i(Y).

This fitting process is performed repeatedly with increasing polynomial orders (starting with linear) until the target accuracy is achieved, or a specified maximum order is reached. If the target accuracy cannot be achieved even with this maximum-order polynomial, the best fitting maximum-order polynomial is returned so long as its accuracy is better than "maxacc". If it is not, an error is reported.

Synopsis: `AstPolyMap *astPolyTran( AstPolyMap *this, int forward, double acc, double maxacc, int maxorder, const double *lbnd, const double *ubnd )`

Parameters:

this
Pointer to the original Mapping.

forward
If non-zero, the forward PolyMap transformation is replaced. Otherwise the inverse transformation is replaced.

acc
The target accuracy, expressed as a geodesic distance within the PolyMap's input space (if "forward" is zero) or output space (if "forward" is non-zero).

maxacc
The maximum allowed accuracy for an acceptable polynomial, expressed as a geodesic distance within the PolyMap's input space (if "forward" is zero) or output space (if "forward" is non-zero).

maxorder
The maximum allowed polynomial order. This is one more than the maximum power of either input axis. So for instance, a value of 3 refers to a quadratic polynomial. Note, cross terms with total powers greater than or equal to maxorder are not inlcuded in the fit. So the maximum number of terms in each of the fitted polynomials is maxorder∗(maxorder+1)/2.

lbnd
Pointer to an array holding the lower bounds of a rectangular region within the PolyMap's input space (if "forward" is zero) or output space (if "forward" is non-zero). The new polynomial will be evaluated over this rectangle. The length of this array should equal the value of the PolyMap's Nin or Nout attribute, depending on "forward".

ubnd
Pointer to an array holding the upper bounds of a rectangular region within the PolyMap's input space (if "forward" is zero) or output space (if "forward" is non-zero). The new polynomial will be evaluated over this rectangle. The length of this array should equal the value of the PolyMap's Nin or Nout attribute, depending on "forward".

Returned Value:

astPolyTran()
A pointer to the new PolyMap. A NULL pointer will be returned if the fit fails to achieve the accuracy specified by "maxacc", but no error will be reported.

Notes:

- This function can only be used on 1D or 2D PolyMaps which have the same number of inputs and outputs.
- A null Object pointer (AST__NULL) will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

---

astPolygon Create a Polygon astPolygon

Description: This function creates a new Polygon object and optionally initialises its attributes.

The Polygon class implements a polygonal area, defined by a collection of vertices, within a 2-dimensional Frame. The vertices are connected together by geodesic curves within the encapsulated Frame. For instance, if the encapsulated Frame is a simple Frame then the geodesics will be straight lines, but if the Frame is a SkyFrame then the geodesics will be great circles. Note, the vertices must be supplied in an order such that the inside of the polygon is to the left of the boundary as the vertices are traversed. Supplying them in the reverse order will effectively negate the polygon.

Within a SkyFrame, neighbouring vertices are always joined using the shortest path. Thus if an edge of 180 degrees or more in length is required, it should be split into section each of which is less than 180 degrees. The closed path joining all the vertices in order will divide the celestial sphere into two disjoint regions. The inside of the polygon is the region which is circled in an anti-clockwise manner (when viewed from the inside of the celestial sphere) when moving through the list of vertices in the order in which they were supplied when the Polygon was created (i.e. the inside is to the left of the boundary when moving through the vertices in the order supplied).

Synopsis: `AstPolygon *astPolygon( AstFrame *frame, int npnt, int dim, const double *points, AstRegion *unc, const char *options, ... )`

Parameters:

frame

A pointer to the Frame in which the region is defined. It must have exactly 2 axes. A deep copy is taken of the supplied Frame. This means that any subsequent changes made to the Frame using the supplied pointer will have no effect the Region.

npnt

The number of points in the Region.

dim

The number of elements along the second dimension of the "points" array (which contains the point coordinates). This value is required so that the coordinate values can be correctly located if they do not entirely fill this array. The value given should not be less than "npnt".

points

The address of the first element of a 2-dimensional array of shape "[2][dim]" giving the physical coordinates of the vertices. These should be stored such that the value of coordinate number "coord" for point number "pnt" is found in element "in[coord][pnt]".

unc

An optional pointer to an existing Region which specifies the uncertainties associated with the boundary of the Box being created. The uncertainty in any point on the boundary of the Box is found by shifting the supplied "uncertainty" Region so that it is centred at the boundary point being considered. The area covered by the shifted uncertainty Region then represents the uncertainty in the boundary position. The uncertainty is assumed to be the same for all points.

If supplied, the uncertainty Region must be of a class for which all instances are centro-symetric (e.g. Box, Circle, Ellipse, etc.) or be a Prism containing centro-symetric component Regions. A deep copy of the supplied Region will be taken, so subsequent changes to the uncertainty Region using the supplied pointer will have no effect on the created Box. Alternatively, a NULL Object pointer may be supplied, in which case a default uncertainty is used equivalent to a box 1.0E-6 of the size of the Box being created.

The uncertainty Region has two uses: 1) when the astOverlap function compares two Regions for equality the uncertainty Region is used to determine the tolerance on the comparison, and 2) when a Region is mapped into a different coordinate system and subsequently simplified (using astSimplify), the uncertainties are used to determine if the transformed boundary can be accurately represented by a specific shape of Region.

options

Pointer to a null-terminated string containing an optional comma-separated list of attribute assignments to be used for initialising the new Polygon. The syntax used is identical to that for the astSet function and may include "printf" format specifiers identified by "%" symbols in the normal way.

...

If the "options" string contains "%" format specifiers, then an optional list of additional arguments may follow it in order to supply values to be substituted for these specifiers. The rules for supplying these are identical to those for the astSet function (and for the C "printf" function).

Returned Value:

astPolygon()

A pointer to the new Polygon.

Notes:

- A null Object pointer (AST__NULL) will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

Status Handling:

The protected interface to this function includes an extra parameter at the end of the parameter list descirbed above. This parameter is a pointer to the integer inherited status variable: "int ∗status".

---

astPrism Create a Prism astPrism

Description: This function creates a new Prism and optionally initialises its attributes.

A Prism is a Region which represents an extrusion of an existing Region into one or more orthogonal dimensions (specified by another Region). If the Region to be extruded has N axes, and the Region defining the extrusion has M axes, then the resulting Prism will have (M+N) axes. A point is inside the Prism if the first N axis values correspond to a point inside the Region being extruded, and the remaining M axis values correspond to a point inside the Region defining the extrusion.

As an example, a cylinder can be represented by extruding an existing Circle, using an Interval to define the extrusion. Ih this case, the Interval would have a single axis and would specify the upper and lower limits of the cylinder along its length.

Synopsis: `AstPrism *astPrism( AstRegion *region1, AstRegion *region2, const char *options, ... )`

Parameters:

region1

Pointer to the Region to be extruded.

region2

Pointer to the Region defining the extent of the extrusion.

options

Pointer to a null-terminated string containing an optional comma-separated list of attribute assignments to be used for initialising the new Prism. The syntax used is identical to that for the astSet function and may include "printf" format specifiers identified by "%" symbols in the normal way.

...

If the "options" string contains "%" format specifiers, then an optional list of additional arguments may follow it in order to supply values to be substituted for these specifiers. The rules for supplying these are identical to those for the astSet function (and for the C "printf" function).

Returned Value:

astPrism()

A pointer to the new Prism.

Notes:

- Deep copies are taken of the supplied Regions. This means that any subsequent changes made to the component Regions using the supplied pointers will have no effect on the Prism.
- A null Object pointer (AST__NULL) will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

---

astPurgeRows Remove all empty rows from a table astPurgeRows

Description: This function removes all empty rows from the Table, renaming the key associated with each table cell accordingly.

Synopsis: `void astPurgeRows( AstTable *this )`

Parameters:

this

Pointer to the Table.

---

astPurgeWCS Delete all cards in the FitsChan describing WCS information astPurgeWCS

Description: This function deletes all cards in a FitsChan that relate to any of the recognised WCS encodings. On exit, the current card is the first remaining card in the FitsChan.

Synopsis: `void astPurgeWCS( AstFitsChan *this )`

Parameters:

this

Pointer to the FitsChan.

---

astPutCards Store a set of FITS header cards in a FitsChan astPutCards

Description: This function stores a set of FITS header cards in a FitsChan. The cards are supplied concatenated together into a single character string. Any existing cards in the FitsChan are removed before the new cards are added. The FitsChan is "re-wound" on exit by clearing its Card attribute. This means that a subsequent invocation of astRead can be made immediately without the need to re-wind the FitsChan first.

Synopsis: `void astPutCards( AstFitsChan *this, const char *cards )`

Parameters:

this

Pointer to the FitsChan.

cards

Pointer to a null-terminated character string containing the FITS cards to be stored. Each individual card should occupy 80 characters in this string, and there should be no delimiters, new lines, etc, between adjacent cards. The final card may be less than 80 characters long. This is the format produced by the fits_hdr2str function in the CFITSIO library.

Notes:

- An error will result if the supplied string contains any cards which cannot be interpreted.

---

astPutChannelData Store arbitrary data to be passed to a source or sink function astPutChannelData

Description: This function stores a supplied arbitrary pointer in the Channel. When a source or sink function is invoked by the Channel, the invoked function can use the astChannelData macro to retrieve the pointer. This provides a thread-safe alternative to passing file descriptors, etc, via global static variables.

Synopsis: `void astPutChannelData( AstChannel *this, void *data )`

Parameters:

this

Pointer to the Channel.

data

A pointer to be made available to the source and sink functions via the astChannelData macro. May be NULL.

Class Applicability:

Channel

All Channels have this function.

Notes:

- This routine is not available in the Fortran 77 interface to the AST library.

---

astPutColumnData Store new data values for all rows of a column astPutColumnData

Description: This function copies data values from a supplied buffer into a named column. The first element in the buffer becomes the first element in the first row of the column. If the buffer does not completely fill the column, then any trailing rows are filled with null values.

Synopsis: `void astPutColumnData( AstFitsTable *this, const char *column, int clen, size_t size, void *coldata )`

Parameters:

this

Pointer to the FitsTable.

column
The character string holding the name of the column. Trailing spaces are ignored.

clen
If the column holds character strings, then this must be set to the length of each fixed length string in the supplied array. This is often determined by the appropriate TFORMn keyword in the binary table header. The supplied value is ignored if the column does not hold character data.

size
The size of the "coldata" array, in bytes. This should be an integer multiple of the number of bytes needed to hold the full vector value stored in a single cell of the column. An error is reported if this is not the case.

coldata
A pointer to an area of memory holding the data to copy into the column. The values should be stored in row order. If the column holds non-scalar values, the elements of each value should be stored in "Fortran" order. No data type conversion is performed.

---

astPutFits Store a FITS header card in a FitsChan astPutFits

Description: This function stores a FITS header card in a FitsChan. The card is either inserted before the current card (identified by the Card attribute), or over-writes the current card, as required.

Synopsis: `void astPutFits( AstFitsChan *this, const char card[ 80 ], int overwrite )`

Parameters:

this
Pointer to the FitsChan.

card
Pointer to a possibly null-terminated character string containing the FITS card to be stored. No more than 80 characters will be used from this string (or fewer if a null occurs earlier).

overwrite
If this value is zero, the new card is inserted in front of the current card in the FitsChan (as identified by the initial value of the Card attribute). If it is non-zero, the new card replaces the current card. In either case, the Card attribute is then incremented by one so that it subsequently identifies the card following the one stored.

Notes:

- If the Card attribute initially points at the "end-of-file" (i.e. exceeds the number of cards in the FitsChan), then the new card is appended as the last card in the FitsChan.
- An error will result if the supplied string cannot be interpreted as a FITS header card.

---

astPutTable Store a single FitsTable in a FitsChan astPutTable

Description: This function allows a representation of a single FITS binary table to be stored in a FitsChan. For instance, this may provide the coordinate look-up tables needed subequently when reading FITS-WCS headers for axes described using the "-TAB" algorithm. Since, in general, the calling application may not know which tables will be needed - if any - prior to calling astRead, the astTablesSource function provides an alternative mechanism in which a caller-supplied function is invoked to store a named table in the FitsChan.

Synopsis: `void astPutTable( AstFitsChan *this, AstFitsTable *table, const char *extnam )`

Parameters:

this

Pointer to the FitsChan.

table

Pointer to a FitsTable to be added to the FitsChan. If a FitsTable with the associated extension name already exists in the FitsChan, it is replaced with the new one. A deep copy of the FitsTable is stored in the FitsChan, so any subsequent changes made to the FitsTable will have no effect on the behaviour of the FitsChan.

extnam

The name of the FITS extension associated with the table.

Notes:

- Tables stored in the FitsChan may be retrieved using astGetTables.
- The astPutTables method can add multiple FitsTables in a single call.

---

astPutTableHeader Store new FITS headers in a FitsTable astPutTableHeader

Description: This function stores new FITS headers in the supplied FitsTable. Any existing headers are first deleted.

Synopsis: `void astPutTableHeader( AstFitsTable *this, AstFitsChan *header )`

Parameters:

this

Pointer to the FitsTable.

header

Pointer to a FitsChan holding the headers for the FitsTable. A deep copy of the supplied FitsChan is stored in the FitsTable, replacing the current FitsChan in the Fitstable. Keywords that are fixed either by the properties of the Table, or by the FITS standard, are removed from the copy (see "Notes:" below).

Notes:

- The attributes of the supplied FitsChan, together with any source and sink functions associated with the FitsChan, are copied to the FitsTable.
- Values for the following keywords are generated automatically by the FitsTable (any values for these keywords in the supplied FitsChan will be ignored): "XTENSION", "BITPIX", "NAXIS", "NAXIS1", "NAXIS2", "PCOUNT", "GCOUNT", "TFIELDS", "TFORM%d", "TTYPE%d", "TNULL%d", "THEAP", "TDIM%d".

---

astPutTables Store one or more FitsTables in a FitsChan astPutTables

Description: This function allows representations of one or more FITS binary tables to be stored in a FitsChan. For instance, these may provide the coordinate look-up tables needed subequently when reading FITS-WCS headers for axes described using the "-TAB" algorithm. Since, in general, the calling application may not know which tables will be needed - if any - prior to calling astRead, the astTablesSource function provides an alternative mechanism in which a caller-supplied function is invoked to store a named table in the FitsChan.

Synopsis: `void astPutTables( AstFitsChan *this, AstKeyMap *tables )`

Parameters:

this
Pointer to the FitsChan.

tables
Pointer to a KeyMap holding the tables that are to be added to the FitsChan. Each entry should hold a scalar value which is a pointer to a FitsTable to be added to the FitsChan. Any unusable entries are ignored. The key associated with each entry should be the name of the FITS binary extension from which the table was read. If a FitsTable with the associated key already exists in the FitsChan, it is replaced with the new one. A deep copy of each usable FitsTable is stored in the FitsChan, so any subsequent changes made to the FitsTables will have no effect on the behaviour of the FitsChan.

Notes:

- Tables stored in the FitsChan may be retrieved using astGetTables.
- The tables in the supplied KeyMap are added to any tables already in the FitsChan.
- The astPutTable method provides a simpler means of adding a single table to a FitsChan.

---

astQuadApprox Obtain a quadratic approximation to a 2D Mapping astQuadApprox

Description: This function returns the co-efficients of a quadratic fit to the supplied Mapping over the input area specified by "lbnd" and "ubnd". The Mapping must have 2 inputs, but may have any number of outputs. The i'th Mapping output is modelled as a quadratic function of the 2 inputs (x,y):

output_i = a_i_0 + a_i_1*x + a_i_2*y + a_i_3*x*y + a_i_4*x*x + a_i_5*y*y

The "fit" array is returned holding the values of the co-efficients a_0_0, a_0_1, etc.

Synopsis: `int QuadApprox( AstMapping *this, const double lbnd[2], const double ubnd[2], int nx, int ny, double *fit, double *rms )`

Parameters:

this
Pointer to the Mapping.

lbnd
Pointer to an array of doubles containing the lower bounds of a box defined within the input coordinate system of the Mapping. The number of elements in this array should equal the value of the Mapping's Nin attribute. This box should specify the region over which the fit is to be performed.

ubnd
Pointer to an array of doubles containing the upper bounds of the box specifying the region over which the fit is to be performed.

nx
The number of points to place along the first Mapping input. The first point is at "lbnd[0]" and the last is at "ubnd[0]". If a value less than three is supplied a value of three will be used.

ny

The number of points to place along the second Mapping input. The first point is at "lbnd[1]" and the last is at "ubnd[1]". If a value less than three is supplied a value of three will be used.

fit

Pointer to an array of doubles in which to return the co-efficients of the quadratic approximation to the specified transformation. This array should have at least "6*Nout", elements. The first 6 elements hold the fit to the first Mapping output. The next 6 elements hold the fit to the second Mapping output, etc. So if the Mapping has 2 inputs and 2 outputs the quadratic approximation to the forward transformation is:

$$X\_out = fit[0] + fit[1]*X\_in + fit[2]*Y\_in + fit[3]*X\_in*Y\_in + fit[4]*X\_in*X\_in + fit[5]*Y\_in*Y\_in$$
$$Y\_out = fit[6] + fit[7]*X\_in + fit[8]*Y\_in + fit[9]*X\_in*Y\_in + fit[10]*X\_in*X\_in + fit[11]*Y\_in*Y\_in$$

rms

Pointer to a double in which to return the RMS residual between the fit and the Mapping, summed over all Mapping outputs.

Returned Value:

astQuadApprox()

If a quadratic approximation was created, a non-zero value is returned. Otherwise zero is returned and the fit co-efficients are set to AST__BAD.

Notes:

- This function fits the Mapping's forward transformation. To fit the inverse transformation, the Mapping should be inverted using astInvert before invoking this function.
- A value of zero will be returned if this function is invoked with the global error status set, or if it should fail for any reason.

---

astRate Calculate the rate of change of a Mapping output astRate

Description: This function evaluates the rate of change of a specified output of the supplied Mapping with respect to a specified input, at a specified input position.

The result is estimated by interpolating the function using a fourth order polynomial in the neighbourhood of the specified position. The size of the neighbourhood used is chosen to minimise the RMS residual per unit length between the interpolating polynomial and the supplied Mapping function. This method produces good accuracy but can involve evaluating the Mapping 100 or more times.

Synopsis: `double astRate( AstMapping *this, double *at, int ax1, int ax2 )`

Parameters:

this

Pointer to the Mapping to be applied.

at

The address of an array holding the axis values at the position at which the rate of change is to be evaluated. The number of elements in this array should equal the number of inputs to the Mapping.

ax1

The index of the Mapping output for which the rate of change is to be found (output numbering starts at 1 for the first output).

ax2
The index of the Mapping input which is to be varied in order to find the rate of change (input numbering starts at 1 for the first input).

Returned Value:

astRate()
The rate of change of Mapping output "ax1" with respect to input "ax2", evaluated at "at", or AST__BAD if the value cannot be calculated.

Notes:

- A value of AST__BAD will be returned if this function is invoked with the global error status set, or if it should fail for any reason.

---

astRateMap Create a RateMap astRateMap

Description: This function creates a new RateMap and optionally initialises its attributes.

A RateMap is a Mapping which represents a single element of the Jacobian matrix of another Mapping. The Mapping for which the Jacobian is required is specified when the new RateMap is created, and is referred to as the "encapsulated Mapping" below.

The number of inputs to a RateMap is the same as the number of inputs to its encapsulated Mapping. The number of outputs from a RateMap is always one. This one output equals the rate of change of a specified output of the encapsulated Mapping with respect to a specified input of the encapsulated Mapping (the input and output to use are specified when the RateMap is created).

A RateMap which has not been inverted does not define an inverse transformation. If a RateMap has been inverted then it will define an inverse transformation but not a forward transformation.

Synopsis: `AstRateMap *astRateMap( AstMapping *map, int ax1, int ax2, const char *options, ... )`

Parameters:

map
Pointer to the encapsulated Mapping.

ax1
Index of the output from the encapsulated Mapping for which the rate of change is required. This corresponds to the delta quantity forming the numerator of the required element of the Jacobian matrix. The first axis has index 1.

ax2
Index of the input to the encapsulated Mapping which is to be varied. This corresponds to the delta quantity forming the denominator of the required element of the Jacobian matrix. The first axis has index 1.

options
Pointer to a null-terminated string containing an optional comma-separated list of attribute assignments to be used for initialising the new RateMap. The syntax used is identical to that for the astSet function and may include "printf" format specifiers identified by "%" symbols in the normal way.

...
If the "options" string contains "%" format specifiers, then an optional list of additional arguments may follow it in order to supply values to be substituted for these specifiers. The rules for supplying these are identical to those for the astSet function (and for the C "printf" function).

Returned Value:

astRateMap()
A pointer to the new RateMap.

Notes:

- The forward transformation of the encapsulated Mapping must be defined.
- Note that the component Mappings supplied are not copied by astRateMap (the new RateMap simply retains a reference to them). They may continue to be used for other purposes, but should not be deleted. If a RateMap containing a copy of its component Mappings is required, then a copy of the RateMap should be made using astCopy.
- A null Object pointer (AST__NULL) will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

---

astRead Read an Object from a Channel astRead

Description: This function reads the next Object from a Channel and returns a pointer to the new Object.

Synopsis: `AstObject *astRead( AstChannel *this )`

Parameters:

this
Pointer to the Channel.

Class Applicability:

FitsChan
All successful use of astRead on a FitsChan is destructive, so that FITS header cards are consumed in the process of reading an Object, and are removed from the FitsChan (this deletion can be prevented for specific cards by calling the FitsChan astRetainFits function). An unsuccessful call of astRead (for instance, caused by the FitsChan not containing the necessary FITS headers cards needed to create an Object) results in the contents of the FitsChan being left unchanged.

StcsChan
The AST Object returned by a successful use of astRead on an StcsChan, will be either a Region or a KeyMap, depending on the values of the StcsArea, StcsCoords and StcsProps attributes. See the documentation for these attributes for further information.

Returned Value:

astRead()
A pointer to the new Object. The class to which this will belong is determined by the input data, so is not known in advance.

Notes:

- A null Object pointer (AST__NULL) will be returned, without error, if the Channel contains no further Objects to be read.
- A null Object pointer will also be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

astReadFits Read cards into a FitsChan from the source function astReadFits

Description: This function reads cards from the source function that was specified when the FitsChan was created, and stores them in the FitsChan. This normally happens once-only, when the FitsChan is accessed for the first time. This function provides a means of forcing a re-read of the external source, and may be useful if (say) new cards have been deposited into the external source. Any newcards read from the source are appended to the end of the current contents of the FitsChan.

Synopsis: `void astReadFits( AstFitsChan *this )`

Parameters:

this

Pointer to the FitsChan.

Notes:

- This function returns without action if no source function was specified when the FitsChan was created.
- The SourceFile attribute is ignored by this function. New cards are read from the source file whenever a new value is assigned to the SourceFile attribute.

astRebin<X> Rebin a region of a data grid astRebin<X>

Description: This is a set of functions for rebinning gridded data (e.g. an image) under the control of a geometrical transformation, which is specified by a Mapping. The functions operate on a pair of data grids (input and output), each of which may have any number of dimensions. Rebinning may be restricted to a specified region of the input grid. An associated grid of error estimates associated with the input data may also be supplied (in the form of variance values), so as to produce error estimates for the rebined output data. Propagation of missing data (bad pixels) is supported.

Note, if you will be rebining a sequence of input arrays and then co-adding them into a single array, the alternative astRebinSeq<X> functions will in general be more efficient.

You should use a rebinning function which matches the numerical type of the data you are processing by replacing <X> in the generic function name astRebin<X> by an appropriate 1- or 2-character type code. For example, if you are rebinning data with type "float", you should use the function astRebinF (see the "Data Type Codes" section below for the codes appropriate to other numerical types).

Rebinning of the grid of input data is performed by transforming the coordinates of the centre of each input grid element (or pixel) into the coordinate system of the output grid. The input pixel value is then divided up and assigned to the output pixels in the neighbourhood of the central output coordinates. A choice of schemes are provided for determining how each input pixel value is divided up between the output pixels. In general, each output pixel may be assigned values from more than one input pixel. All contributions to a given output pixel are summed to produce the final output pixel value. Output pixels can be set to the supplied bad value if they receive contributions from an insufficient number of input pixels. This is controlled by the "wlim" parameter.

Input pixel coordinates are transformed into the coordinate system of the output grid using the forward transformation of the Mapping which is supplied. This means that geometrical features in the input data are subjected to the Mapping's forward transformation as they are transferred from the input to the output grid.

In practice, transforming the coordinates of every pixel of a large data grid can be time-consuming, especially if the Mapping involves complicated functions, such as sky projections. To improve performance, it is therefore possible to approximate non-linear Mappings by a set of linear transformations which are applied piece-wise to separate sub-regions of the data. This approximation process is applied automatically by an adaptive algorithm, under control of an accuracy criterion which expresses the maximum tolerable geometrical distortion which may be introduced, as a fraction of a pixel.

This algorithm first attempts to approximate the Mapping with a linear transformation applied over the whole region of the input grid which is being used. If this proves to be insufficiently accurate, the input region is sub-divided into two along its largest dimension and the process is repeated within each of the resulting sub-regions. This process of sub-division continues until a sufficiently good linear approximation is found, or the region to which it is being applied becomes too small (in which case the original Mapping is used directly).

Synopsis: `void astRebin<X>(AstMapping *this, double wlim, int ndim_in, const int lbnd_in[], const int ubnd_in[], const <Xtype> in[], const <Xtype> in_var[], int spread, const double params[], int flags, double tol, int maxpix, <Xtype> badval, int ndim_out, const int lbnd_out[], const int ubnd_out[], const int lbnd[], const int ubnd[], <Xtype> out[], <Xtype> out_var[]);`

Parameters:

this

Pointer to a Mapping, whose forward transformation will be used to transform the coordinates of pixels in the input grid into the coordinate system of the output grid.

The number of input coordinates used by this Mapping (as given by its Nin attribute) should match the number of input grid dimensions given by the value of "ndim_in" below. Similarly, the number of output coordinates (Nout attribute) should match the number of output grid dimensions given by "ndim_out".

wlim

Gives the required number of input pixel values which must contribute to an output pixel in order for the output pixel value to be considered valid. If the sum of the input pixel weights contributing to an output pixel is less than the supplied "wlim" value, then the output pixel value is returned set to the supplied bad value.

ndim_in

The number of dimensions in the input grid. This should be at least one.

lbnd_in

Pointer to an array of integers, with "ndim_in" elements, containing the coordinates of the centre of the first pixel in the input grid along each dimension.

ubnd_in

Pointer to an array of integers, with "ndim_in" elements, containing the coordinates of the centre of the last pixel in the input grid along each dimension.

Note that "lbnd_in" and "ubnd_in" together define the shape and size of the input grid, its extent along a particular (j'th) dimension being ubnd_in[j]-lbnd_in[j]+1 (assuming the index "j" to be zero-based). They also define the input grid's coordinate system, each pixel having unit extent along each dimension with integral coordinate values at its centre.

in

Pointer to an array, with one element for each pixel in the input grid, containing the input data to be rebined. The numerical type of this array should match the 1- or 2-character type code appended to the function name (e.g. if you are using astRebinF, the type of each array element should be "float").

The storage order of data within this array should be such that the index of the first grid dimension varies most rapidly and that of the final dimension least rapidly (i.e. Fortran array indexing is used).

in_var

An optional pointer to a second array with the same size and type as the "in" array. If given, this should contain a set of non-negative values which represent estimates of the statistical variance associated with each element of the "in" array. If this array is supplied (together with the corresponding "out_var" array), then estimates of the variance of the rebined output data will be calculated.

If no input variance estimates are being provided, a NULL pointer should be given.

spread

This parameter specifies the scheme to be used for dividing each input data value up amongst the corresponding output pixels. It may be used to select from a set of pre-defined schemes by supplying one of the values described in the "Pixel Spreading Schemes" section below. If a value of zero is supplied, then the default linear spreading scheme is used (equivalent to supplying the value AST__LINEAR).

params

An optional pointer to an array of double which should contain any additional parameter values required by the pixel spreading scheme. If such parameters are required, this will be noted in the "Pixel Spreading Schemes" section below.

If no additional parameters are required, this array is not used and a NULL pointer may be given.

flags

The bitwise OR of a set of flag values which may be used to provide additional control over the rebinning operation. See the "Control Flags" section below for a description of the options available. If no flag values are to be set, a value of zero should be given.

tol

The maximum tolerable geometrical distortion which may be introduced as a result of approximating non-linear Mappings by a set of piece-wise linear transformations. This should be expressed as a displacement in pixels in the output grid's coordinate system.

If piece-wise linear approximation is not required, a value of zero may be given. This will ensure that the Mapping is used without any approximation, but may increase execution time.

If the value is too high, discontinuities between the linear approximations used in adjacent panel will be higher, and may cause the edges of the panel to be visible when viewing the output image at high contrast. If this is a problem, reduce the tolerance value used.

maxpix

A value which specifies an initial scale size (in pixels) for the adaptive algorithm which approximates non-linear Mappings with piece-wise linear transformations. Normally, this should be a large value (larger than any dimension of the region of the input grid being used). In this case, a first attempt to approximate the Mapping by a linear transformation will be made over the entire input region.

If a smaller value is used, the input region will first be divided into sub-regions whose size does not exceed "maxpix" pixels in any dimension. Only at this point will attempts at approximation commence.

This value may occasionally be useful in preventing false convergence of the adaptive algorithm in cases where the Mapping appears approximately linear on large scales, but has irregularities (e.g. holes) on smaller scales. A value of, say, 50 to 100 pixels can also be employed as a safeguard in general-purpose software, since the effect on performance is minimal.

If too small a value is given, it will have the effect of inhibiting linear approximation altogether (equivalent to setting "tol" to zero). Although this may degrade performance, accurate results will still be obtained.

badval

This argument should have the same type as the elements of the "in" array. It specifies the value used to flag missing data (bad pixels) in the input and output arrays.

If the AST__USEBAD flag is set via the "flags" parameter, then this value is used to test for bad pixels in the "in" (and "in_var") array(s).

In all cases, this value is also used to flag any output elements in the "out" (and "out_var") array(s) for which rebined values could not be obtained (see the "Propagation of Missing Data" section below for details of the circumstances under which this may occur).

ndim_out

The number of dimensions in the output grid. This should be at least one. It need not necessarily be equal to the number of dimensions in the input grid.

lbnd_out

Pointer to an array of integers, with "ndim_out" elements, containing the coordinates of the centre of the first pixel in the output grid along each dimension.

ubnd_out

Pointer to an array of integers, with "ndim_out" elements, containing the coordinates of the centre of the last pixel in the output grid along each dimension.

Note that "lbnd_out" and "ubnd_out" together define the shape, size and coordinate system of the output grid in the same way as "lbnd_in" and "ubnd_in" define the shape, size and coordinate system of the input grid.

lbnd

Pointer to an array of integers, with "ndim_in" elements, containing the coordinates of the first pixel in the region of the input grid which is to be included in the rebined output array.

ubnd

Pointer to an array of integers, with "ndim_in" elements, containing the coordinates of the last pixel in the region of the input grid which is to be included in the rebined output array.

Note that "lbnd" and "ubnd" together define the shape and position of a (hyper-)rectangular region of the input grid which is to be included in the rebined output array. This region should lie wholly within the extent of the input grid (as defined by the "lbnd_in" and "ubnd_in" arrays). Regions of the input grid lying outside this region will not be used.

out

Pointer to an array, with one element for each pixel in the output grid, in which the rebined data values will be returned. The numerical type of this array should match that of the "in" array, and the data storage order should be such that the index of the first grid dimension varies most rapidly and that of the final dimension least rapidly (i.e. Fortran array indexing is used).

out_var

An optional pointer to an array with the same type and size as the "out" array. If given, this array will be used to return variance estimates for the rebined data values. This array will only be used if the "in_var" array has also been supplied.

The output variance values will be calculated on the assumption that errors on the input data values are statistically independent and that their variance estimates may simply be summed (with appropriate weighting factors) when several input pixels contribute to an output data value. If this assumption is not valid, then the output error estimates may be biased. In addition, note that the statistical errors on neighbouring output data values (as well as the estimates of those errors) may often be correlated, even if the above assumption about the input data is correct, because of the pixel spreading schemes employed.

If no output variance estimates are required, a NULL pointer should be given.

Data Type Codes:

To select the appropriate rebinning function, you should replace <X> in the generic function name astRebin<X> with a 1- or 2-character data type code, so as to match the numerical type <Xtype> of the data you are processing, as follows:

- D: double
- F: float
- I: int

For example, astRebinD would be used to process "double" data, while astRebinI would be used to process "int" data, etc.

Note that, unlike astResample<X>, the astRebin<X> set of functions does not yet support unsigned integer data types or integers of different sizes.

Pixel Spreading Schemes:

The pixel spreading scheme specifies the Point Spread Function (PSF) applied to each input pixel value as it is copied into the output array. It can be thought of as the inverse of the sub-pixel interpolation schemes used by the astResample<X> group of functions. That is, in a sub-pixel interpolation scheme the kernel specifies the weight to assign to each input pixel when forming the weighted mean of the input pixels, whereas the kernel in a pixel spreading scheme specifies the fraction of the input data value which is to be assigned to each output pixel. As for interpolation, the choice of suitable pixel spreading scheme involves stricking a balance between schemes which tend to degrade sharp features in the data by smoothing them, and those which attempt to preserve sharp features but which often tend to introduce unwanted artifacts. See the astResample<X> documentation for further discussion.

The binning algorithm used has the ability to introduce artifacts not seen when using a resampling algorithm. Particularly, when viewing the output image at high contrast, systems of curves lines covering the entire image may be visible. These are caused by a beating effect between the input pixel positions and the output pixels position, and their nature and strength depend critically upon the nature of the Mapping and the spreading function being used. In general, the nearest neighbour spreading function demonstrates this effect more clearly than the other functions, and for this reason should be used with caution.

The following values (defined in the "ast.h" header file) may be assigned to the "spread" parameter. See the astResample<X> documentation for details of these schemes including the use of the "fspread" and "params" parameters:

- AST__NEAREST
- AST__LINEAR
- AST__SINC
- AST__SINCSINC
- AST__SINCCOS
- AST__SINCGAUSS
- AST__SOMBCOS

In addition, the following schemes can be used with astRebin<X> but not with astResample<X>:

- AST__GAUSS: This scheme uses a kernel of the form exp(-k*x*x), with k a positive constant determined by the full-width at half-maximum (FWHM). The FWHM should be supplied in units of output pixels by means of the "params[1]" value and should be at least 0.1. The "params[0]" value should be used to specify at what point the Gaussian is truncated to zero. This should be given as a number of output pixels on either side of the central output point in each dimension (the nearest integer value is used).

Control Flags:

The following flags are defined in the "ast.h" header file and may be used to provide additional control over the rebinning process. Having selected a set of flags, you should supply the bitwise OR of their values via the "flags" parameter:

- AST__USEBAD: Indicates that there may be bad pixels in the input array(s) which must be recognised by comparing with the value given for "badval" and propagated to the output array(s). If this flag is not set, all input values are treated literally and the "badval" value is only used for flagging output array values.

Propagation of Missing Data:

Instances of missing data (bad pixels) in the output grid are identified by occurrences of the "badval" value in the "out" array. These are produced if the sum of the weights of the contributing input pixels is less than "wlim".

An input pixel is considered bad (and is consequently ignored) if its data value is equal to "badval" and the AST__USEBAD flag is set via the "flags" parameter.

In addition, associated output variance estimates (if calculated) may be declared bad and flagged with the "badval" value in the "out_var" array for similar reasons.

---

astRebinSeq<X> Rebin a region of a sequence of data grids astRebinSeq<X>

Description: This set of functions is identical to astRebin<X> except that the rebinned input data is added into the supplied output arrays, rather than simply over-writing the contents of the output arrays. Thus, by calling this function repeatedly, a sequence of input arrays can be rebinned and accumulated into a single output array, effectively forming a mosaic of the input data arrays.

In addition, the weights associated with each output pixel are returned. The weight of an output pixel indicates the number of input pixels which have been accumulated in that output pixel. If the entire value of an input pixel is assigned to a single output pixel, then the weight of that output pixel is incremented by one. If some fraction of the value of an input pixel is assigned to an output pixel, then the weight of that output pixel is incremented by the fraction used.

The start of a new sequence is indicated by specifying the AST__REBININIT flag via the "flags" parameter. This causes the supplied arrays to be filled with zeros before the rebinned input data is added into them. Subsequenct invocations within the same sequence should omit the AST__REBININIT flag.

The last call in a sequence is indicated by specifying the AST__REBINEND flag. Depending on which flags are supplied, this may cause the output data and variance arrays to be normalised before being returned. This normalisation consists of dividing the data array by the weights array, and can eliminate artifacts which may be introduced into the rebinned data as a consequence of aliasing between the input and output grids. This results in each output pixel value being the weighted mean of the input pixel values that fall in the neighbourhood of the output pixel (rather like astResample<X>). Optionally, these normalised values can then be multiplied by a scaling factor to ensure that the total data sum in any small area is unchanged. This scaling factor is equivalent to the number of input pixel values that fall into each output pixel. In addition to normalisation of the output data values, any output variances are also appropriately normalised, and any output data values with weight less than "wlim" are set to "badval".

Output variances can be generated in two ways; by rebinning the supplied input variances with appropriate weights, or by finding the spread of input data values contributing to each output pixel (see the AST__GENVAR and AST__USEVAR flags).

Synopsis:

```
void astRebinSeq<X>( AstMapping *this, double wlim, int ndim_in, const int lbnd_in[],
const int ubnd_in[], const <Xtype> in[], const <Xtype> in_var[], int spread, const
double params[], int flags, double tol, int maxpix, <Xtype> badval, int ndim_out,
const int lbnd_out[], const int ubnd_out[], const int lbnd[], const int ubnd[], <Xtype>
out[], <Xtype> out_var[], double weights[], int64_t *nused );
```

Parameters:

this

Pointer to a Mapping, whose forward transformation will be used to transform the coordinates of pixels in the input grid into the coordinate system of the output grid.

The number of input coordinates used by this Mapping (as given by its Nin attribute) should match the number of input grid dimensions given by the value of "ndim_in" below. Similarly, the number of output coordinates (Nout attribute) should match the number of output grid dimensions given by "ndim_out". If "in" is NULL, the Mapping will not be used, but a valid Mapping must still be supplied.

wlim

This value is only used if the AST__REBINEND flag is specified via the "flags" parameter. It gives the required number of input pixel values which must contribute to an output pixel (i.e. the output pixel weight) in order for the output pixel value to be considered valid. If the sum of the input pixel weights contributing to an output pixel is less than the supplied "wlim" value, then the output pixel value is returned set to the supplied bad value. If the supplied value is less than 1.0E-10 then 1.0E-10 is used instead.

ndim_in

The number of dimensions in the input grid. This should be at least one. Not used if "in" is NULL.

lbnd_in

Pointer to an array of integers, with "ndim_in" elements, containing the coordinates of the centre of the first pixel in the input grid along each dimension. Not used if "in" is NULL.

ubnd_in

Pointer to an array of integers, with "ndim_in" elements, containing the coordinates of the centre of the last pixel in the input grid along each dimension.

Note that "lbnd_in" and "ubnd_in" together define the shape and size of the input grid, its extent along a particular (j'th) dimension being ubnd_in[j]-lbnd_in[j]+1 (assuming the index "j" to be zero-based). They also define the input grid's coordinate system, each pixel having unit extent along each dimension with integral coordinate values at its centre. Not used if "in" is NULL.

in

Pointer to an array, with one element for each pixel in the input grid, containing the input data to be rebined. The numerical type of this array should match the 1- or 2-character type code appended to the function name (e.g. if you are using astRebinSeqF, the type of each array element should be "float").

The storage order of data within this array should be such that the index of the first grid dimension varies most rapidly and that of the final dimension least rapidly (i.e. Fortran array indexing is used). If a NULL pointer is supplied for "in", then no data is added to the output arrays, but any initialisation or normalisation requested by "flags" is still performed.

in_var

An optional pointer to a second array with the same size and type as the "in" array. If given, this should contain a set of non-negative values which represent estimates of the statistical variance associated with each element of the "in" array. If neither the AST__USEVAR nor the AST__VARWGT flag is set, no input variance estimates are required and this pointer will not be used. A NULL pointer may then be supplied.

spread

This parameter specifies the scheme to be used for dividing each input data value up amongst the corresponding output pixels. It may be used to select from a set of pre-defined schemes by supplying one of the values described in the "Pixel Spreading Schemes" section in the description of the astRebin<X> functions. If a value of zero is supplied, then the default linear spreading scheme is used (equivalent to supplying the value AST__LINEAR). Not used if "in" is NULL.

params

An optional pointer to an array of double which should contain any additional parameter values required by the pixel spreading scheme. If such parameters are required, this will be noted in the "Pixel Spreading Schemes" section in the description of the astRebin<X> functions.

If no additional parameters are required, this array is not used and a NULL pointer may be given. Not used if "in" is NULL.

flags

The bitwise OR of a set of flag values which may be used to provide additional control over the rebinning operation. See the "Control Flags" section below for a description of the options available. If no flag values are to be set, a value of zero should be given.

tol

The maximum tolerable geometrical distortion which may be introduced as a result of approximating non-linear Mappings by a set of piece-wise linear transformations. This should be expressed as a displacement in pixels in the output grid's coordinate system.

If piece-wise linear approximation is not required, a value of zero may be given. This will ensure that the Mapping is used without any approximation, but may increase execution time.

If the value is too high, discontinuities between the linear approximations used in adjacent panel will be higher, and may cause the edges of the panel to be visible when viewing the output image at high contrast. If this is a problem, reduce the tolerance value used. Not used if "in" is NULL.

maxpix

A value which specifies an initial scale size (in pixels) for the adaptive algorithm which approximates non-linear Mappings with piece-wise linear transformations. Normally, this should be a large value (larger than any dimension of the region of the input grid being used). In this case, a first attempt to approximate the Mapping by a linear transformation will be made over the entire input region.

If a smaller value is used, the input region will first be divided into sub-regions whose size does not exceed "maxpix" pixels in any dimension. Only at this point will attempts at approximation commence.

This value may occasionally be useful in preventing false convergence of the adaptive algorithm in cases where the Mapping appears approximately linear on large scales, but has irregularities (e.g. holes) on smaller scales. A value of, say, 50 to 100 pixels can also be employed as a safeguard in general-purpose software, since the effect on performance is minimal.

If too small a value is given, it will have the effect of inhibiting linear approximation altogether (equivalent to setting "tol" to zero). Although this may degrade performance, accurate results will still be obtained. Not used if "in" is NULL.

badval

This argument should have the same type as the elements of the "in" array. It specifies the value used to flag missing data (bad pixels) in the input and output arrays.

If the AST__USEBAD flag is set via the "flags" parameter, then this value is used to test for bad pixels in the "in" (and "in_var") array(s).

In all cases, this value is also used to flag any output elements in the "out" (and "out_var") array(s) for which rebined values could not be obtained (see the "Propagation of Missing Data" section below for details of the circumstances under which this may occur).

ndim_out

The number of dimensions in the output grid. This should be at least one. It need not necessarily be equal to the number of dimensions in the input grid.

lbnd_out

Pointer to an array of integers, with "ndim_out" elements, containing the coordinates of the centre of the first pixel in the output grid along each dimension.

ubnd_out

Pointer to an array of integers, with "ndim_out" elements, containing the coordinates of the centre of the last pixel in the output grid along each dimension.

Note that "lbnd_out" and "ubnd_out" together define the shape, size and coordinate system of the output grid in the same way as "lbnd_in" and "ubnd_in" define the shape, size and coordinate system of the input grid.

lbnd

Pointer to an array of integers, with "ndim_in" elements, containing the coordinates of the first pixel in the region of the input grid which is to be included in the rebined output array. Not used if "in" is NULL.

ubnd

Pointer to an array of integers, with "ndim_in" elements, containing the coordinates of the last pixel in the region of the input grid which is to be included in the rebined output array.

Note that "lbnd" and "ubnd" together define the shape and position of a (hyper-)rectangular region of the input grid which is to be included in the rebined output array. This region should lie wholly within the extent of the input grid (as defined by the "lbnd_in" and "ubnd_in" arrays). Regions of the input grid lying outside this region will not be used. Not used if "in" is NULL.

out

Pointer to an array, with one element for each pixel in the output grid. The rebined data values will be added into the original contents of this array. The numerical type of this array should match that of the "in" array, and the data storage order should be such that the index of the first grid dimension varies most rapidly and that of the final dimension least rapidly (i.e. Fortran array indexing is used).

out_var

A pointer to an array with the same type and size as the "out" array. This pointer will only be used if the AST__USEVAR or AST__GENVAR flag is set in which case variance estimates for the rebined data values will be added into the array. If neither the AST__USEVAR flag nor the AST__GENVAR flag is set, no output variance estimates will be calculated and this pointer will not be used. A NULL pointer may then be supplied.

weights

Pointer to an array of double, with one or two elements for each pixel in the output grid, depending on whether or not the AST__GENVAR flag has been supplied via the "flags" parameter. If AST__GENVAR has not been specified then the array should have one element for each output pixel, and it will be used to accumulate the weight associated with each output pixel. If AST__GENVAR has been specified then the array should have two elements for each output pixel. The first half of the array is again used to accumulate the weight associated with each output pixel, and the second half is used to accumulate the square of the weights. In each half, the data storage order should be such that the index of the first grid dimension varies most rapidly and that of the final dimension least rapidly (i.e. Fortran array indexing is used).

nused

A pointer to an int64_t containing the number of input data values that have been added into the output array so far. The supplied value is incremented on exit by the number of input values used. The value is initially set to zero if the AST__REBININIT flag is set in "flags".

Data Type Codes:

To select the appropriate rebinning function, you should replace <X> in the generic function name astRebinSeq<X> with a 1- or 2-character data type code, so as to match the numerical type <Xtype> of the data you are processing, as follows:

- D: double
- F: float

- I: int

For example, astRebinSeqD would be used to process "double" data, while astRebinSeqI would be used to process "int" data, etc.

Note that, unlike astResample<X>, the astRebinSeq<X> set of functions does not yet support unsigned integer data types or integers of different sizes.

Control Flags:

The following flags are defined in the "ast.h" header file and may be used to provide additional control over the rebinning process. Having selected a set of flags, you should supply the bitwise OR of their values via the "flags" parameter:

- AST__REBININIT: Used to mark the first call in a sequence. It indicates that the supplied "out", "out_var" and "weights" arrays should be filled with zeros (thus over-writing any supplied values) before adding the rebinned input data into them. This flag should be used when rebinning the first input array in a sequence.
- AST__REBINEND: Used to mark the last call in a sequence. It causes each value in the "out" and "out_var" arrays to be divided by a normalisation factor before being returned. The normalisation factor for each output data value is just the corresponding value from the weights array. The normalisation factor for each output variance value is the square of the data value normalisation factor (see also AST__CONSERVEFLUX). It also causes output data values to be set bad if the corresponding weight is less than the value supplied for parameter "wlim". It also causes any temporary values stored in the output variance array (see flag AST__GENVAR below) to be converted into usable variance values. Note, this flag is ignored if the AST__NONORM flag is set.
- AST__USEBAD: Indicates that there may be bad pixels in the input array(s) which must be recognised by comparing with the value given for "badval" and propagated to the output array(s). If this flag is not set, all input values are treated literally and the "badval" value is only used for flagging output array values.
- AST__USEVAR: Indicates that output variance estimates should be created by rebinning the supplied input variance estimates. An error will be reported if both this flag and the AST__GENVAR flag are supplied.
- AST__GENVAR: Indicates that output variance estimates should be created based on the spread of input data values contributing to each output pixel. An error will be reported if both this flag and the AST__USEVAR flag are supplied. If the AST__GENVAR flag is specified, the supplied output variance array is first used as a work array to accumulate the temporary values needed to generate the output variances. When the sequence ends (as indicated by the AST__REBINEND flag), the contents of the output variance array are converted into the required variance estimates. If the generation of such output variances is required, this flag should be used on every invocation of this function within a sequence, and any supplied input variances will have no effect on the output variances (although input variances will still be used to weight the input data if the AST__VARWGT flag is also supplied). The statistical meaning of these output varianes is determined by the presence or absence of the AST__DISVAR flag (see below).
- AST__DISVAR: This flag is ignored unless the AST__GENVAR flag has also been specified. It determines the statistical meaning of the generated output variances. If AST__DISVAR is not specified, generated variances represent variances on the output mean values. If AST__DISVAR is specified, the generated variances represent the variance of the distribution from which the input values were taken. Each output variance created with AST__DISVAR will be larger than that created without AST__DISVAR by a factor equal to the number of input samples that contribute to the output sample.

- AST__VARWGT: Indicates that the input data should be weighted by the reciprocal of the input variances. Otherwise, all input data are given equal weight. If this flag is specified, the calculation of the output variances (if any) is modified to take account of the varying weights assigned to the input data values.
- AST__NONORM: If the simple unnormalised sum of all input data falling in each output pixel is required, then this flag should be set on each call in the sequence and the AST__REBINEND should not be used on the last call. In this case NULL pointers can be supplied for "weights" and "nused". This flag cannot be used with the AST__CONSERVEFLUX, AST__GENVAR or AST__VARWGT flag.
- AST__CONSERVEFLUX: Indicates that the normalized output pixel values generated by the AST__REBINEND flag should be scaled in such a way as to preserve the total data value in a feature on the sky. Without this flag, each normalised output pixel value represents a weighted mean of the input data values around the corresponding input position. is appropriate if the input data represents the spatial density of some quantity (e.g. surface brightness in Janskys per square arc-second) because the output pixel values will have the same normalisation and units as the input pixel values. However, if the input data values represent flux (or some other physical quantity) per pixel, then the AST__CONSERVEFLUX flag could be of use. It causes each output pixel value to be scaled by the ratio of the output pixel size to the input pixel size.

This flag can only be used if the Mapping is successfully approximated by one or more linear transformations. Thus an error will be reported if it used when the "tol" parameter is set to zero (which stops the use of linear approximations), or if the Mapping is too non-linear to be approximated by a piece-wise linear transformation. The ratio of output to input pixel size is evaluated once for each panel of the piece-wise linear approximation to the Mapping, and is assumed to be constant for all output pixels in the panel. The scaling factors for adjacent panels will in general differ slightly, and so the joints between panels may be visible when viewing the output image at high contrast. If this is a problem, reduce the value of the "tol" parameter until the difference between adjacent panels is sufficiently small to be insignificant.

This flag should normally be supplied on each invocation of astRebinSeq<X> within a given sequence.

Note, this flag cannot be used in conjunction with the AST__NOSCALE flag (an error will be reported if both flags are specified).

Propagation of Missing Data:

Instances of missing data (bad pixels) in the output grid are identified by occurrences of the "badval" value in the "out" array. These are only produced if the AST__REBINEND flag is specified and a pixel has zero weight.

An input pixel is considered bad (and is consequently ignored) if its data value is equal to "badval" and the AST__USEBAD flag is set via the "flags" parameter.

In addition, associated output variance estimates (if calculated) may be declared bad and flagged with the "badval" value in the "out_var" array for similar reasons.

---

astRemapFrame Modify a Frame's relationship to other Frames in a FrameSet astRemapFrame

Description: This function modifies the relationship (i.e. Mapping) between a specified Frame in a FrameSet and the other Frames in that FrameSet.

Typically, this might be required if the FrameSet has been used to calibrate (say) an image, and that image is re-binned. The Frame describing the image will then have undergone a coordinate transformation, and this should be communicated to the associated FrameSet using this function.

Synopsis: `void astRemapFrame( AstFrameSet *this, int iframe, AstMapping *map )`

Parameters:

this

Pointer to the FrameSet.

iframe

The index within the FrameSet of the Frame to be modified. This value should lie in the range from 1 to the number of Frames in the FrameSet (as given by its Nframe attribute).

map

Pointer to a Mapping whose forward transformation converts coordinate values from the original coordinate system described by the Frame to the new one, and whose inverse transformation converts in the opposite direction.

Notes:

- A value of AST__BASE or AST__CURRENT may be given for the "iframe" parameter to specify the base Frame or the current Frame respectively.
- The relationship between the selected Frame and any other Frame within the FrameSet will be modified by this function, but the relationship between all other Frames in the FrameSet remains unchanged.
- The number of input coordinate values accepted by the Mapping (its Nin attribute) and the number of output coordinate values generated (its Nout attribute) must be equal and must match the number of axes in the Frame being modified.
- If a simple change of axis order is required, then the astPermAxes function may provide a more straightforward method of making the required changes to the FrameSet.
- This function cannot be used to change the number of Frame axes. To achieve this, a new Frame must be added to the FrameSet (astAddFrame) and the original one removed if necessary (astRemoveFrame).
- Any variant Mappings associated with the remapped Frame (except for the current variant) will be lost as a consequence of calling this method (see attribute "Variant").

---

astRemoveColumn Remove a column from a table astRemoveColumn

Description: This function removes a specified column from the supplied table. The function returns without action if the named column does not exist in the Table (no error is reported).

Synopsis: `void astRemoveColumn( AstTable *this, const char *name )`

Parameters:

this

Pointer to the Table.

name

The column name. Trailing spaces are ignored (all other spaces are significant). Case is significant.

---

astRemoveFrame Remove a Frame from a FrameSet astRemoveFrame

Description: This function removes a Frame from a FrameSet. All other Frames in the FrameSet have their indices re-numbered from one (if necessary), but are otherwise unchanged.

Synopsis: `void astRemoveFrame( AstFrameSet *this, int iframe )`

Parameters:

this
Pointer to the FrameSet.

iframe
The index within the FrameSet of the Frame to be removed. This value should lie in the range from 1 to the number of Frames in the FrameSet (as given by its Nframe attribute).

Notes:

- Removing a Frame from a FrameSet does not affect the relationship between other Frames in the FrameSet, even if they originally depended on the Frame being removed.
- The number of Frames in a FrameSet cannot be reduced to zero. An error will result if an attempt is made to remove the only remaining Frame.
- A value of AST__BASE or AST__CURRENT may be given for the "iframe" parameter to specify the base Frame or the current Frame respectively.
- If a FrameSet's base or current Frame is removed, the Base or Current attribute (respectively) of the FrameSet will have its value cleared, so that another Frame will then assume its role by default.
- If any other Frame is removed, the base and current Frames will remain the same. To ensure this, the Base and/or Current attributes of the FrameSet will be changed, if necessary, to reflect any change in the indices of these Frames.

astRemoveParameter Remove a global parameter from a table astRemoveParameter

Description: This function removes a specified global parameter from the supplied table. The function returns without action if the named parameter does not exist in the Table (no error is reported).

Synopsis: `void astRemoveParameter( AstTable *this, const char *name )`

Parameters:

this
Pointer to the Table.

name
The parameter name. Trailing spaces are ignored (all other spaces are significant). Case is significant.

astRemoveRegions Remove any Regions from a Mapping astRemoveRegions

Description: This function searches the suppliedMapping (which may be a compound Mapping such as a CmpMap) for any component Mappings that are instances of the AST Region class. It then creates a new Mapping from which all Regions have been removed. If a Region cannot simply be removed (for instance, if it is a component of a parallel CmpMap), then it is replaced with an equivalent UnitMap in the returned Mapping.

Synopsis: `AstMapping *astRemoveRegions( AstMapping *this )`

Parameters:

this

Pointer to the original Mapping.

Class Applicability:

CmpFrame

If the supplied Mapping is a CmpFrame, any component Frames that are instances of the Region class are replaced by the equivalent Frame.

FrameSet

If the supplied Mapping is a FrameSet, the returned Mapping will be a copy of the supplied FrameSet in which Regions have been removed from all the inter-Frame Mappings, and any Frames which are instances of the Region class are repalced by the equivalent Frame.

Mapping

This function applies to all Mappings.

Region

If the supplied Mapping is a Region, the returned Mapping will be the equivalent Frame.

Returned Value:

astRemoveRegions()

A new pointer to the (possibly modified) Mapping.

Notes:

- This function can safely be applied even to Mappings which contain no Regions. If no Regions are found, it behaves exactly like astClone and returns a pointer to the original Mapping.
- The Mapping returned by this function may not be independent of the original (even if some Regions were removed), and modifying it may therefore result in indirect modification of the original. If a completely independent result is required, a copy should be made using astCopy.
- A null Object pointer (AST__NULL) will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

astRemoveRow Remove a row from a table astRemoveRow

Description: This function removes a specified row from the supplied table. The function returns without action if the row does not exist in the Table (no error is reported).

Synopsis: `void astRemoveRow( AstTable *this, int index )`

Parameters:

this

Pointer to the Table.

index

The index of the row to be removed. The first row has index 1.

astRemoveTables Remove one or more tables from a FitsChan astRemoveTables

Description: This function removes the named tables from the FitsChan, it they exist (no error is reported if any the tables do not exist).

Synopsis: `void astRemoveTables( AstFitsChan *this, const char *key )`

Parameters:

this
Pointer to the FitsChan.

key
The key indicating which tables to exist. A single key or a comma-separated list of keys can be supplied. If a blank string is supplied, all tables are removed.

---

astResample<X> Resample a region of a data grid astResample<X>

Description: This is a set of functions for resampling gridded data (e.g. an image) under the control of a geometrical transformation, which is specified by a Mapping. The functions operate on a pair of data grids (input and output), each of which may have any number of dimensions. Resampling may be restricted to a specified region of the output grid. An associated grid of error estimates associated with the input data may also be supplied (in the form of variance values), so as to produce error estimates for the resampled output data. Propagation of missing data (bad pixels) is supported.

You should use a resampling function which matches the numerical type of the data you are processing by replacing <X> in the generic function name astResample<X> by an appropriate 1- or 2-character type code. For example, if you are resampling data with type "float", you should use the function astResampleF (see the "Data Type Codes" section below for the codes appropriate to other numerical types).

Resampling of the grid of input data is performed by transforming the coordinates of the centre of each output grid element (or pixel) into the coordinate system of the input grid. Since the resulting coordinates will not, in general, coincide with the centre of an input pixel, sub-pixel interpolation is performed between the neighbouring input pixels. This produces a resampled value which is then assigned to the output pixel. A choice of sub-pixel interpolation schemes is provided, but you may also implement your own.

This algorithm samples the input data value, it does not integrate it. Thus total data value in the input image will not, in general, be conserved. However, an option is provided (see the "Control Flags" section below) which can produce approximate flux conservation by scaling the output values using the ratio of the output pixel size to the input pixel size. However, if accurate flux conservation is important to you, consder using the astRebin<X> or astRebinSeq<X> family of functions instead.

Output pixel coordinates are transformed into the coordinate system of the input grid using the inverse transformation of the Mapping which is supplied. This means that geometrical features in the input data are subjected to the Mapping's forward transformation as they are transferred from the input to the output grid (although the Mapping's forward transformation is not explicitly used).

In practice, transforming the coordinates of every pixel of a large data grid can be time-consuming, especially if the Mapping involves complicated functions, such as sky projections. To improve performance, it is therefore possible to approximate non-linear Mappings by a set of linear transformations which are applied piece-wise to separate sub-regions of the data. This approximation process is applied automatically by an adaptive algorithm, under control of an accuracy criterion which expresses the maximum tolerable geometrical distortion which may be introduced, as a fraction of a pixel.

This algorithm first attempts to approximate the Mapping with a linear transformation applied over the whole region of the output grid which is being used. If this proves to be insufficiently accurate, the output region is sub-divided into two along its largest dimension and the process is repeated within each of the resulting sub-regions. This process of sub-division continues until a sufficiently good linear approximation is found, or the region to which it is being applied becomes too small (in which case the original Mapping is used directly).

Synopsis: `int astResample<X>(AstMapping *this, int ndim_in, const int lbnd_in[], const int ubnd_in[], const <Xtype> in[], const <Xtype> in_var[], int interp, void (* finterp)(void), const double params[], int flags, double tol, int maxpix, <Xtype> badval, int ndim_out, const int lbnd_out[], const int ubnd_out[], const int lbnd[], const int ubnd[], <Xtype> out[], <Xtype> out_var[]);`

Parameters:

this

Pointer to a Mapping, whose inverse transformation will be used to transform the coordinates of pixels in the output grid into the coordinate system of the input grid. This yields the positions which are used to obtain resampled values by sub-pixel interpolation within the input grid.

The number of input coordinates used by this Mapping (as given by its Nin attribute) should match the number of input grid dimensions given by the value of "ndim_in" below. Similarly, the number of output coordinates (Nout attribute) should match the number of output grid dimensions given by "ndim_out".

ndim_in

The number of dimensions in the input grid. This should be at least one.

lbnd_in

Pointer to an array of integers, with "ndim_in" elements, containing the coordinates of the centre of the first pixel in the input grid along each dimension.

ubnd_in

Pointer to an array of integers, with "ndim_in" elements, containing the coordinates of the centre of the last pixel in the input grid along each dimension.

Note that "lbnd_in" and "ubnd_in" together define the shape and size of the input grid, its extent along a particular (j'th) dimension being ubnd_in[j]-lbnd_in[j]+1 (assuming the index "j" to be zero-based). They also define the input grid's coordinate system, each pixel having unit extent along each dimension with integral coordinate values at its centre.

in

Pointer to an array, with one element for each pixel in the input grid, containing the input data to be resampled. The numerical type of this array should match the 1- or 2-character type code appended to the function name (e.g. if you are using astResampleF, the type of each array element should be "float").

The storage order of data within this array should be such that the index of the first grid dimension varies most rapidly and that of the final dimension least rapidly (i.e. Fortran array indexing is used).

in_var

An optional pointer to a second array with the same size and type as the "in" array. If given, this should contain a set of non-negative values which represent estimates of the statistical variance associated with each element of the "in" array. If this array is supplied (together with the corresponding "out_var" array), then estimates of the variance of the resampled output data will be calculated.

If no input variance estimates are being provided, a NULL pointer should be given.

interp

This parameter specifies the scheme to be used for sub-pixel interpolation within the input grid. It may be used to select from a set of pre-defined schemes by supplying one of the values described in the "Sub-Pixel Interpolation Schemes" section below. If a value of zero is supplied, then the default linear interpolation scheme is used (equivalent to supplying the value AST__LINEAR).

Alternatively, you may supply a value which indicates that you will provide your own function to perform sub-pixel interpolation by means of the "finterp " parameter. Again, see the "Sub-Pixel Interpolation Schemes" section below for details.

finterp

If the value given for the "interp" parameter indicates that you will provide your own function for sub-pixel interpolation, then a pointer to that function should be given here. For details of the interface which the function should have (several are possible, depending on the value of "interp"), see the "Sub-Pixel Interpolation Schemes" section below.

If the "interp" parameter has any other value, corresponding to one of the pre-defined interpolation schemes, then this function will not be used and you may supply a NULL pointer.

params

An optional pointer to an array of double which should contain any additional parameter values required by the sub-pixel interpolation scheme. If such parameters are required, this will be noted in the "Sub-Pixel Interpolation Schemes" section below (you may also use this array to pass values to your own interpolation function).

If no additional parameters are required, this array is not used and a NULL pointer may be given.

flags

The bitwise OR of a set of flag values which may be used to provide additional control over the resampling operation. See the "Control Flags" section below for a description of the options available. If no flag values are to be set, a value of zero should be given.

tol

The maximum tolerable geometrical distortion which may be introduced as a result of approximating non-linear Mappings by a set of piece-wise linear transformations. This should be expressed as a displacement in pixels in the input grid's coordinate system.

If piece-wise linear approximation is not required, a value of zero may be given. This will ensure that the Mapping is used without any approximation, but may increase execution time.

maxpix

A value which specifies an initial scale size (in pixels) for the adaptive algorithm which approximates non-linear Mappings with piece-wise linear transformations. Normally, this should be a large value (larger than any dimension of the region of the output grid being used). In this case, a first attempt to approximate the Mapping by a linear transformation will be made over the entire output region.

If a smaller value is used, the output region will first be divided into sub-regions whose size does not exceed "maxpix" pixels in any dimension. Only at this point will attempts at approximation commence.

This value may occasionally be useful in preventing false convergence of the adaptive algorithm in cases where the Mapping appears approximately linear on large scales, but has irregularities (e.g. holes) on smaller scales. A value of, say, 50 to 100 pixels can also be employed as a safeguard in general-purpose software, since the effect on performance is minimal.

If too small a value is given, it will have the effect of inhibiting linear approximation altogether (equivalent to setting "tol" to zero). Although this may degrade performance, accurate results will still be obtained.

badval

This argument should have the same type as the elements of the "in" array. It specifies the value used to flag missing data (bad pixels) in the input and output arrays.

If the AST__USEBAD flag is set via the "flags" parameter, then this value is used to test for bad pixels in the "in" (and "in_var") array(s).

Unless the AST__NOBAD flag is set via the "flags" parameter, this value is also used to flag any output elements in the "out" (and "out_var") array(s) for which resampled values could not be obtained (see the "Propagation of Missing Data" section below for details of the circumstances under which this may occur). The astResample<X> function return value indicates whether any such values have been produced. If the AST__NOBAD flag is set.

then output array elements for which no resampled value could be obtained are left set to the value they had on entry to this function.

ndim_out

The number of dimensions in the output grid. This should be at least one. It need not necessarily be equal to the number of dimensions in the input grid.

lbnd_out

Pointer to an array of integers, with "ndim_out" elements, containing the coordinates of the centre of the first pixel in the output grid along each dimension.

ubnd_out

Pointer to an array of integers, with "ndim_out" elements, containing the coordinates of the centre of the last pixel in the output grid along each dimension.

Note that "lbnd_out" and "ubnd_out" together define the shape, size and coordinate system of the output grid in the same way as "lbnd_in" and "ubnd_in" define the shape, size and coordinate system of the input grid.

lbnd

Pointer to an array of integers, with "ndim_out" elements, containing the coordinates of the first pixel in the region of the output grid for which a resampled value is to be calculated.

ubnd

Pointer to an array of integers, with "ndim_out" elements, containing the coordinates of the last pixel in the region of the output grid for which a resampled value is to be calculated.

Note that "lbnd" and "ubnd" together define the shape and position of a (hyper-)rectangular region of the output grid for which resampled values should be produced. This region should lie wholly within the extent of the output grid (as defined by the "lbnd_out" and "ubnd_out" arrays). Regions of the output grid lying outside this region will not be modified.

out

Pointer to an array, with one element for each pixel in the output grid, into which the resampled data values will be returned. The numerical type of this array should match that of the "in" array, and the data storage order should be such that the index of the first grid dimension varies most rapidly and that of the final dimension least rapidly (i.e. Fortran array indexing is used).

out_var

An optional pointer to an array with the same type and size as the "out" array. If given, this array will be used to return variance estimates for the resampled data values. This array will only be used if the "in_var" array has also been supplied.

The output variance values will be calculated on the assumption that errors on the input data values are statistically independent and that their variance estimates may simply be summed (with appropriate weighting factors) when several input pixels contribute to an output data value. If this assumption is not valid, then the output error estimates may be biased. In addition, note that the statistical errors on neighbouring output data values (as well as the estimates of those errors) may often be correlated, even if the above assumption about the input data is correct, because of the sub-pixel interpolation schemes employed.

If no output variance estimates are required, a NULL pointer should be given.

Returned Value:

astResample<X>()

The number of output pixels for which no valid resampled value could be obtained. Thus, in the absence of any error, a returned value of zero indicates that all the required output pixels received valid resampled data values (and variances). See the "badval" and "flags" parameters.

Notes:

- A value of zero will be returned if this function is invoked with the global error status set, or if it should fail for any reason.

Data Type Codes:

To select the appropriate resampling function, you should replace <X> in the generic function name astResample<X> with a 1- or 2-character data type code, so as to match the numerical type <Xtype> of the data you are processing, as follows:

- D: double
- F: float
- L: long int (may be 32 or 64 bit)
- K: 64 bit int
- UL: unsigned long int (may be 32 or 64 bit)
- UK: unsigned 64 bit int
- I: int
- UI: unsigned int
- S: short int
- US: unsigned short int
- B: byte (signed char)
- UB: unsigned byte (unsigned char)

For example, astResampleD would be used to process "double" data, while astResampleS would be used to process "short int" data, etc.

Sub-Pixel Interpolation Schemes:

There is no such thing as a perfect sub-pixel interpolation scheme and, in practice, all resampling will result in some degradation of gridded data. A range of schemes is therefore provided, from which you can choose the one which best suits your needs.

In general, a balance must be struck between schemes which tend to degrade sharp features in the data by smoothing them, and those which attempt to preserve sharp features. The latter will often tend to introduce unwanted oscillations, typically visible as "ringing" around sharp features and edges, especially if the data are under-sampled (i.e. if the sharpest features are less than about two pixels across). In practice, a good interpolation scheme is likely to be a compromise and may exhibit some aspects of both these features.

For under-sampled data, some interpolation schemes may appear to preserve data resolution because they transform single input pixels into single output pixels, rather than spreading their data between several output pixels. While this may look better cosmetically, it can result in a geometrical shift of sharp features in the data. You should beware of this if you plan to use such features (e.g.) for image alignment.

The following are two easy-to-use sub-pixel interpolation schemes which are generally applicable. They are selected by supplying the appropriate value (defined in the "ast.h" header file) via the "interp" parameter. In these cases, the "finterp" and "params" parameters are not used:

- AST__NEAREST: This is the simplest possible scheme, in which the value of the input pixel with the nearest centre to the interpolation point is used. This is very quick to execute and will preserve single-pixel features in the data, but may displace them by up to half their width along each dimension. It often gives a good cosmetic result, so is useful for quick-look processing, but is unsuitable if accurate geometrical transformation is required.

- AST__LINEAR: This is the default scheme, which uses linear interpolation between the nearest neighbouring pixels in the input grid (there are two neighbours in one dimension, four neighbours in two dimensions, eight in three dimensions, etc.). It is superior to the nearest-pixel scheme (above) in not displacing features in the data, yet it still executes fairly rapidly. It is generally a safe choice if you do not have any particular reason to favour another scheme, since it cannot introduce oscillations. However, it does introduce some spatial smoothing which varies according to the distance of the interpolation point from the neighbouring pixels. This can degrade the shape of sharp features in the data in a position-dependent way. It may also show in the output variance grid (if used) as a pattern of stripes or fringes.

An alternative set of interpolation schemes is based on forming the interpolated value from the weighted sum of a set of surrounding pixel values (not necessarily just the nearest neighbours). This approach has its origins in the theory of digital filtering, in which interpolated values are obtained by conceptually passing the sampled data (represented by a grid of delta functions) through a linear filter which implements a convolution. Because the convolution kernel is continuous, the convolution yields a continuous function which may then be evaluated at fractional pixel positions. The (possibly multi-dimensional) kernel is usually regarded as "separable" and formed from the product of a set of identical 1-dimensional kernel functions, evaluated along each dimension. Different interpolation schemes are then distinguished by the choice of this 1-dimensional interpolation kernel. The number of surrounding pixels which contribute to the result may also be varied.

From a practical standpoint, it is useful to divide the weighted sum of pixel values by the sum of the weights when determining the interpolated value. Strictly, this means that a true convolution is no longer being performed. However, the distinction is rarely important in practice because (for slightly subtle reasons) the sum of weights is always approximately constant for good interpolation kernels. The advantage of this technique, which is used here, is that it can easily accommodate missing data and tends to minimise unwanted oscillations at the edges of the data grid.

In the following schemes, which are based on a 1-dimensional interpolation kernel, the first element of the "params" array should be used to specify how many pixels are to contribute to the interpolated result on either side of the interpolation point in each dimension (the nearest integer value is used). Execution time increases rapidly with this number. Typically, a value of 2 is appropriate and the minimum value used will be 1 (i.e. two pixels altogether, one on either side of the interpolation point). A value of zero or less may be given for "params[0]" to indicate that a suitable number of pixels should be calculated automatically.

In each of these cases, the "finterp" parameter is not used:

- AST__GAUSS: This scheme uses a kernel of the form exp(-k*x*x), with k a positive constant. The full-width at half-maximum (FWHM) is given by "params[1]" to zero will select the number of contributing pixels so as to utilise the width of the kernel out to where the envelope declines to 1% of its maximum value). This kernel suppresses noise at the expense of smoothing the output array.
- AST__SINC: This scheme uses a sinc(pi*x) kernel, where x is the pixel offset from the interpolation point and sinc(z)=sin(z)/z. This sometimes features as an "optimal" interpolation kernel in books on image processing. Its supposed optimality depends on the assumption that the data are band-limited (i.e. have no spatial frequencies above a certain value) and are adequately sampled. In practice, astronomical data rarely meet these requirements. In addition, high spatial frequencies are often present due (e.g.) to image defects and cosmic ray events. Consequently, substantial ringing can be experienced with this kernel. The kernel also decays slowly with distance, so that many surrounding pixels are required, leading to poor performance. Abruptly truncating it, by using only a few neighbouring pixels, improves performance and may reduce ringing (if "params[0]" is set to zero, then only two pixels will be used on either side). However, a more gradual truncation, as implemented by other kernels, is generally to be preferred. This kernel is provided mainly so that you can convince yourself not to use it!

- AST__SINCSINC: This scheme uses an improved kernel, of the form sinc(pi∗x).sinc(k∗pi∗x), with k a constant, out to the point where sinc(k∗pi∗x) goes to zero, and zero beyond. The second sinc() factor provides an "envelope" which gradually rolls off the normal sinc(pi∗x) kernel at large offsets. The width of this envelope is specified by giving the number of pixels offset at which it goes to zero by means of the "params[1]" value, which should be at least 1.0 (in addition, setting "params[0]" to zero will select the number of contributing pixels so as to utilise the full width of the kernel, out to where it reaches zero). The case given by "params[0]=2, params[1]=2" is typically a good choice and is sometimes known as the Lanczos kernel. This is a valuable general-purpose interpolation scheme, intermediate in its visual effect on images between the AST__NEAREST and AST__LINEAR schemes. Although the kernel is slightly oscillatory, ringing is adequately suppressed if the data are well sampled.
- AST__SINCCOS: This scheme uses a kernel of the form sinc(pi∗x).cos(k∗pi∗x), with k a constant, out to the point where cos(k∗pi∗x) goes to zero, and zero beyond. As above, the cos() factor provides an envelope which gradually rolls off the sinc() kernel at large offsets. The width of this envelope is specified by giving the number of pixels offset at which it goes to zero by means of the "params[1]" value, which should be at least 1.0 (in addition, setting "params[0]" to zero will select the number of contributing pixels so as to utilise the full width of the kernel, out to where it reaches zero). This scheme gives similar results to the AST__SINCSINC scheme, which it resembles.
- AST__SINCGAUSS: This scheme uses a kernel of the form sinc(pi∗x).exp(-k∗x∗x), with k a positive constant. Here, the sinc() kernel is rolled off using a Gaussian envelope which is specified by giving its full-width at half-maximum (FWHM) by means of the "params[1]" value, which should be at least 0.1 (in addition, setting "params[0]" to zero will select the number of contributing pixels so as to utilise the width of the kernel out to where the envelope declines to 1% of its maximum value). On astronomical images and spectra, good results are often obtained by approximately matching the FWHM of the envelope function, given by "params[1]", to the point spread function of the input data. However, there does not seem to be any theoretical reason for this.
- AST__SOMB: This scheme uses a somb(pi∗x) kernel (a "sombrero" function), where x is the pixel offset from the interpolation point and somb(z)=2∗J1(z)/z (J1 is a Bessel function of the first kind of order 1). It is similar to the AST__SINC kernel, and has the same parameter usage.
- AST__SOMBCOS: This scheme uses a kernel of the form somb(pi∗x).cos(k∗pi∗x), with k a constant, out to the point where cos(k∗pi∗x) goes to zero, and zero beyond. It is similar to the AST__SINCCOS kernel, and has the same parameter usage.

In addition, the following schemes are provided which are not based on a 1-dimensional kernel:

- AST__BLOCKAVE: This scheme simply takes an average of all the pixels on the input grid in a cube centred on the interpolation point. The number of pixels in the cube is determined by the value of the first element of the "params" array, which gives the number of pixels in each dimension on either side of the central point. Hence a block of (2 ∗ params[0])∧ndim_in pixels in the input grid will be examined to determine the value of the output pixel. If the variance is not being used (var_in or var_out = NULL) then all valid pixels in this cube will be averaged in to the result with equal weight. If variances are being used, then each input pixel will be weighted proportionally to the reciprocal of its variance; any pixel without a valid variance will be discarded. This scheme is suitable where the output grid is much coarser than the input grid; if the ratio of pixel sizes is R then a suitable value of params[0] may be R/2.

Finally, supplying the following values for "interp" allows you to implement your own sub-pixel interpolation scheme by means of your own function. You should supply a pointer to this function via the "finterp" parameter:

- AST__UKERN1: In this scheme, you supply a function to evaluate your own 1-dimensional interpolation kernel, which is then used to perform sub-pixel interpolation (as described above). The function you supply should have the same interface as the fictitious astUkern1 function (q.v.). In addition, a value should be given via "params[0]" to specify the number of neighbouring pixels which are to contribute to each interpolated value (in the same way as for the pre-defined interpolation schemes described above). Other elements of the "params" array are available to pass values to your interpolation function.
- AST__UINTERP: This is a completely general scheme, in which your interpolation function has access to all of the input data. This allows you to implement any interpolation algorithm you choose, which could (for example) be non-linear, or adaptive. In this case, the astResample<X> functions play no role in the sub-pixel interpolation process and simply handle the geometrical transformation of coordinates and other housekeeping. The function you supply should have the same interface as the fictitious astUinterp function (q.v.). In this case, the "params" parameter is not used by astResample<X>, but is available to pass values to your interpolation function.

Control Flags:

The following flags are defined in the "ast.h" header file and may be used to provide additional control over the resampling process. Having selected a set of flags, you should supply the bitwise OR of their values via the "flags" parameter:

- AST__NOBAD: Indicates that any output array elements for which no resampled value could be obtained should be left set to the value they had on entry to this function. If this flag is not supplied, such output array elements are set to the value supplied for parameter "badval". Note, this flag cannot be used in conjunction with the AST__CONSERVEFLUX flag (an error will be reported if both flags are specified).
- AST__URESAMP1, 2, 3 & 4: A set of four flags which are reserved for your own use. They may be used to pass private information to any sub-pixel interpolation function which you implement yourself. They are ignored by all the pre-defined interpolation schemes.
- AST__USEBAD: Indicates that there may be bad pixels in the input array(s) which must be recognised by comparing with the value given for "badval" and propagated to the output array(s). If this flag is not set, all input values are treated literally and the "badval" value is only used for flagging output array values.
- AST__CONSERVEFLUX: Indicates that the output pixel values should be scaled in such a way as to preserve (approximately) the total data value in a feature on the sky. Without this flag, each output pixel value represents an instantaneous sample of the input data values at the corresponding input position. This is appropriate if the input data represents the spatial density of some quantity (e.g. surface brightness in Janskys per square arc-second) because the output pixel values will have the same normalisation and units as the input pixel values. However, if the input data values represent flux (or some other physical quantity) per pixel, then the AST__CONSERVEFLUX flag could be used. This causes each output pixel value to be scaled by the ratio of the output pixel size to the input pixel size.

This flag can only be used if the Mapping is successfully approximated by one or more linear transformations. Thus an error will be reported if it used when the "tol" parameter is set to zero (which stops the use of linear approximations), or if the Mapping is too non-linear to be approximated by a piece-wise linear transformation. The ratio of output to input pixel size is evaluated once for each panel of the piece-wise linear approximation to the Mapping, and is assumed to be constant for all output pixels in the panel. The scaling factors for adjacent panels will in general differ slightly, and so the joints between panels may be visible when viewing the output image at high contrast. If this is a problem, reduce the value of the "tol" parameter until the difference between adjacent panels is sufficiently small to be insignificant.

Note, this flag cannot be used in conjunction with the AST__NOBAD flag (an error will be reported if both flags are specified).

Propagation of Missing Data:

Unless the AST__NOBAD flag is specified, instances of missing data (bad pixels) in the output grid are identified by occurrences of the "badval" value in the "out" array. These may be produced if any of the following happen:

- The input position (the transformed position of the output pixel's centre) lies outside the boundary of the grid of input pixels.
- The input position lies inside the boundary of a bad input pixel. In this context, an input pixel is considered bad if its data value is equal to "badval" and the AST__USEBAD flag is set via the "flags" parameter. (Positions which have half-integral coordinate values, and therefore lie on a pixel boundary, are regarded as lying within the pixel with the larger, i.e. more positive, index.)
- The set of neighbouring input pixels (excluding those which are bad) is unsuitable for calculating an interpolated value. Whether this is true may depend on the sub-pixel interpolation scheme in use.
- The interpolated value lies outside the range which can be represented using the data type of the "out" array.

In addition, associated output variance estimates (if calculated) may be declared bad and flagged with the "badval" value in the "out_var" array under any of the following circumstances:

- The associated resampled data value (in the "out" array) is bad.
- The set of neighbouring input pixels which contributed to the output data value do not all have valid variance estimates associated with them. In this context, an input variance estimate may be regarded as bad either because it has the value "badval" (and the AST__USEBAD flag is set), or because it is negative.
- The set of neighbouring input pixels for which valid variance values are available is unsuitable for calculating an overall variance value. Whether this is true may depend on the sub-pixel interpolation scheme in use.
- The variance value lies outside the range which can be represented using the data type of the "out_var" array.

If the AST__NOBAD flag is specified via parameter "flags", then output array elements that would otherwise be set to "badval" are instead left holding the value they had on entry to this function. The number of such array elements is returned as the function value.

---

astResolve Resolve a vector into two orthogonal components astResolve

Description: This function resolves a vector into two perpendicular components. The vector from point 1 to point 2 is used as the basis vector. The vector from point 1 to point 3 is resolved into components parallel and perpendicular to this basis vector. The lengths of the two components are returned, together with the position of closest aproach of the basis vector to point 3.

Synopsis:

```
void astResolve( AstFrame *this, const double point1[], const double point2[],
const double point3[], double point4[], double *d1, double *d2 );
```

Parameters:

this
Pointer to the Frame.

point1
An array of double, with one element for each Frame axis (Naxes attribute). This marks the start of the basis vector, and of the vector to be resolved.

point2
An array of double, with one element for each Frame axis (Naxes attribute). This marks the end of the basis vector.

point3
An array of double, with one element for each Frame axis (Naxes attribute). This marks the end of the vector to be resolved.

point4
An array of double, with one element for each Frame axis in which the coordinates of the point of closest approach of the basis vector to point 3 will be returned.

d1
The address of a location at which to return the distance from point 1 to point 4 (that is, the length of the component parallel to the basis vector). Positive values are in the same sense as movement from point 1 to point 2.

d2
The address of a location at which to return the distance from point 4 to point 3 (that is, the length of the component perpendicular to the basis vector). The value is always positive.

Notes:

- Each vector used in this function is the path of shortest distance between two points, as defined by the astDistance function.
- This function will return "bad" coordinate values (AST__BAD) if any of the input coordinates has this value, or if the required output values are undefined.

---

astRetainFits Indicate that the current card in a FitsChan should be retained astRetainFits

Description: This function stores a flag with the current card in the FitsChan indicating that the card should not be removed from the FitsChan when an Object is read from the FitsChan using astRead.

Cards that have not been flagged in this way are removed when a read operation completes successfully, but only if the card was used in the process of creating the returned AST Object. Any cards that are irrelevant to the creation of the AST Object are retained whether or not they are flagged.

Synopsis: `void astRetainFits( AstFitsChan *this )`

Parameters:

this
Pointer to the FitsChan.

Notes:

- This function returns without action if the FitsChan is initially positioned at the "end-of-file" (i.e. if the Card attribute exceeds the number of cards in the FitsChan).
- The current card is not changed by this function.

---

astSame Test if two AST pointers refer to the same Object astSame

Description: This function returns a boolean result (0 or 1) to indicate whether two pointers refer to the same Object.

Synopsis: `int astSame( AstObject *this, AstObject *that )`

Parameters:

this
Pointer to the first Object.

that
Pointer to the second Object.

Class Applicability:

Object
This function applies to all Objects.

Returned Value:

astSame()
One if the two pointers refer to the same Object, otherwise zero.

Notes:

- Two independent Objects that happen to be identical are not considered to be the same Object by this function.
- A value of zero will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

---

astSelectorMap Create a SelectorMap astSelectorMap

Description: This function creates a new SelectorMap and optionally initialises its attributes.

A SelectorMap is a Mapping that identifies which Region contains a given input position.

A SelectorMap encapsulates a number of Regions that all have the same number of axes and represent the same coordinate Frame. The number of inputs (Nin attribute) of the SelectorMap equals the number of axes spanned by one of the encapsulated Region. All SelectorMaps have only a single output. SelectorMaps do not define an inverse transformation.

For each input position, the forward transformation of a SelectorMap searches through the encapsulated Regions (in the order supplied when the SelectorMap was created) until a Region is found which contains the input position. The index associated with this Region is returned as the SelectorMap output value (the index value is the position of the Region within the list of Regions supplied when the SelectorMap was created, starting at 1 for the first Region). If an input position is not contained within any Region, a value of zero is returned by the forward transformation.

If a compound Mapping contains a SelectorMap in series with its own inverse, the combination of the two adjacent SelectorMaps will be replaced by a UnitMap when the compound Mapping is simplified using astSimplify.

In practice, SelectorMaps are often used in conjunction with SwitchMaps.

Synopsis: `AstSelectorMap *astSelectorMap( int nreg, AstRegion *regs[], double badval, const char *options, ... )`

Parameters:

nreg
The number of supplied Regions.

regs
An array of pointers to the Regions. All the supplied Regions must relate to the same coordinate Frame. The number of axes in this coordinate Frame defines the number of inputs for the SelectorMap.

badval
The value to be returned by the forward transformation of the SelectorMap for any input positions that have a bad (AST__BAD) value on any axis.

options
Pointer to a null-terminated string containing an optional comma-separated list of attribute assignments to be used for initialising the new SelectorMap. The syntax used is identical to that for the astSet function and may include "printf" format specifiers identified by "%" symbols in the normal way.

...
If the "options" string contains "%" format specifiers, then an optional list of additional arguments may follow it in order to supply values to be substituted for these specifiers. The rules for supplying these are identical to those for the astSet function (and for the C "printf" function).

Returned Value:

astSelectorMap()
A pointer to the new SelectorMap.

Notes:

- Deep copies are taken of the supplied Regions. This means that any subsequent changes made to the component Regions using the supplied pointers will have no effect on the SelectorMap.
- A null Object pointer (AST__NULL) will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

---

astSet Set attribute values for an Object astSet

Description: This function assigns a set of attribute values to an Object, over-riding any previous values. The attributes and their new values are specified via a character string, which should contain a comma-separated list of the form:

"attribute_1 = value_1, attribute_2 = value_2, ... "

where "attribute_n" specifies an attribute name, and the value to the right of each "=" sign should be a suitable textual representation of the value to be assigned. This value will be interpreted according to the attribute's data type.

The string supplied may also contain "printf"-style format specifiers, identified by "%" signs in the usual way. If present, these will be substituted by values supplied as additional optional arguments (using the normal "printf" rules) before the string is used.

Synopsis: `void astSet( AstObject *this, const char *settings, ... )`

Parameters:

this
Pointer to the Object.

settings
Pointer to a null-terminated character string containing a comma-separated list of attribute settings in the form described above.

...

Optional additional arguments which supply values to be substituted for any "printf"-style format specifiers that appear in the "settings" string.

Class Applicability:

Object

This function applies to all Objects.

Examples:

`astSet( map, "Report = 1, Zoom = 25.0" );`

Sets the Report attribute for Object "map" to the value 1 and the Zoom attribute to 25.0.

`astSet( frame, "Label( %d ) =Offset along axis %d", axis, axis );`

Sets the Label(axis) attribute for Object "frame" to a suitable string, where the axis number is obtained from "axis", a variable of type int.

`astSet( frame, "Title =%s", mystring );`

Sets the Title attribute for Object "frame" to the contents of the string "mystring".

Notes:

- Attribute names are not case sensitive and may be surrounded by white space.
- White space may also surround attribute values, where it will generally be ignored (except for string-valued attributes where it is significant and forms part of the value to be assigned).
- To include a literal comma in the value assigned to an attribute, the whole attribute value should be enclosed in quotation markes. Alternatively, you can use "%s" format and supply the value as a separate additional argument to astSet (or use the astSetC function instead).
- The same procedure may be adopted if "%" signs are to be included and are not to be interpreted as format specifiers (alternatively, the "printf" convention of writing "%%" may be used).
- An error will result if an attempt is made to set a value for a read-only attribute.

astSet<X> Set an attribute value for an Object astSet<X>

Description: This is a family of functions which set a specified attribute value for an Object using one of several different data types. The type is selected by replacing <X> in the function name by C, D, F, I or L, to supply a value in const char∗ (i.e. string), double, float, int, or long format, respectively.

If possible, the value you supply is converted to the type of the attribute. If conversion is not possible, an error will result.

Synopsis: `void astSet<X>( AstObject *this, const char *attrib, <X>type value )`

Parameters:

this

Pointer to the Object.

attrib

Pointer to a null-terminated character string containing the name of the attribute whose value is to be set.

value

The value to be set for the attribute, in the data type corresponding to <X> (or, in the case of astSetC, a pointer to a null-terminated character string containing this value).

Class Applicability:

Object
These functions apply to all Objects.

Examples:

`astSetI( frame, "Preserve", 1 );`
Sets the Preserve attribute value for Object "frame" to 1.

`astSetC( plot, "Format(1)", "%.2g" );`
Sets the Format(1) attribute value for Object "plot" to the character string "%.2g".

Notes:

- Attribute names are not case sensitive and may be surrounded by white space.
- An error will result if an attempt is made to set a value for a read-only attribute.

---

astSetActiveUnit Specify how the Unit attribute should be used astSetActiveUnit

Description: This function sets the current value of the ActiveUnit flag for a Frame, which controls how the Frame behaves when it is used (by astFindFrame or astConvert) to match another Frame. If the ActiveUnit flag is set in both template and target Frames then the returned Mapping takes into account any differences in axis units. The default value for simple Frames is zero, which preserves the behaviour of versions of AST prior to version 2.0.

If the ActiveUnit flag of either Frame is zero, then the Mapping will ignore any difference in the Unit attributes of corresponding template and target axes. In this mode, the Unit attributes are purely descriptive commentary for the benefit of human readers and do not influence the Mappings between Frames. This is the behaviour which all Frames had in older version of AST, prior to the introduction of this attribute.

If the ActiveUnit flag of both Frames is non-zero, then the Mapping from template to target will take account of any difference in the axis Unit attributes, where-ever possible. For instance, if corresponding target and template axes have Unit strings of "km" and "m", then the FrameSet class will use a ZoomMap to connect them which introduces a scaling of 1000. If no Mapping can be found between the corresponding units string, then an error is reported. In this mode, it is assumed that values of the Unit attribute conform to the syntax for units strings described in the FITS WCS Paper I "Representations of world coordinates in FITS" (Greisen & Calabretta). Particularly, any of the named unit symbols, functions, operators or standard multiplier prefixes listed within that paper can be used within a units string. A units string may contain symbols for unit which are not listed in the FITS paper, but transformation to any other units will then not be possible (except to units which depend only on the same unknown units - thus "flops" can be transformed to "Mflops" even though "flops" is not a standard FITS unit symbol).

A range of common non-standard variations of unit names and multiplier prefixes are also allowed, such as adding an "s" to the end of Angstrom, using a lower case "a" at the start of "angstrom", "micron" instead of "um", "sec" instead of "s", etc.

If the ActiveUnit flag is non-zero, setting a new Unit value for an axis may also change its Label and Symbol attributes. For instance, if an axis has Unit "Hz" and Label "frequency", then changing its Unit to "log(Hz)" will change its Label to "log(frequency)". In addition, the Axis Format attribute will be cleared when-ever a new value is assigned to the Unit attribute.

Note, if a non-zero value is set for the ActiveUnit flag, then changing a Unit value for the current Frame within a FrameSet will result in the Frame being re-mapped (that is, the Mappings which define the relationships between Frames within the FrameSet will be modified to take into account the change in Units).

Synopsis: `void astSetActiveUnit( AstFrame *this, int value )`

Parameters:

this
Pointer to the Frame.

value
The new value to use.

Class Applicability:

SkyFrame
The ActiveUnit flag for a SkyFrame is always 0 (any value supplied using this function is ignored).

SpecFrame
The ActiveUnit flag for a SpecFrame is always 1 (any value supplied using this function is ignored).

FluxFrame
The ActiveUnit flag for a FluxFrame is always 1 (any value supplied using this function is ignored).

CmpFrame
The default ActiveUnit flag for a CmpFrame is 1 if both of the component Frames are using active units, and zero otherwise. When a new value is set for the ActiveUnit flag, the flag value is propagated to the component Frames. This change will be reflected through all references to the component Frames, not just those encapsulated within the CmpFrame.

Region:
Regions always use active units if possible.

Notes:

- The ActiveUnit flag resembles a Frame attribute, except that it cannot be tested or cleared, and it cannot be accessed using the generic astGet<X> and astSet<X> functions.
- The astGetActiveUnit function can be used to retrieve the current value of the ActiveUnit flag.

---

astSetFits<X> Store a keyword value in a FitsChan astSetFits<X>

Description: This is a family of functions which store values for named keywords within a FitsChan at the current card position. The supplied keyword value can either over-write an existing keyword value, or can be inserted as a new header card into the FitsChan.

The keyword data type is selected by replacing <X> in the function name by one of the following strings representing the recognised FITS data

types:

- CF - Complex floating point values.
- CI - Complex integer values.
- F - Floating point values.
- I - Integer values.
- L - Logical (i.e. boolean) values.
- S - String values.

- CN - A "CONTINUE" value, these are treated like string values, but are encoded without an equals sign.

The data type of the "value" parameter depends on <X> as follows:

- CF - "double ∗" (a pointer to a 2 element array holding the real and imaginary parts of the complex value).
- CI - "int ∗" (a pointer to a 2 element array holding the real and imaginary parts of the complex value).
- F - "double".
- I - "int".
- L - "int".
- S - "const char ∗".
- CN - "const char ∗".

Synopsis: `void astSetFits<X>( AstFitsChan *this, const char *name, <X>type value, const char *comment, int overwrite )`

Parameters:

this

Pointer to the FitsChan.

name

Pointer to a null-terminated character string containing the FITS keyword name. This may be a complete FITS header card, in which case the keyword to use is extracted from it. No more than 80 characters are read from this string.

value

The keyword value to store with the named keyword. The data type of this parameter depends on <X> as described above.

comment

A pointer to a null terminated string holding a comment to associated with the keyword. If a NULL pointer or a blank string is supplied, then any comment included in the string supplied for the "name" parameter is used instead. If "name" contains no comment, then any existing comment in the card being over-written is retained. Otherwise, no comment is stored with the card.

overwrite

If non-zero, the new card formed from the supplied keyword name, value and comment string over-writes the current card, and the current card is incremented to refer to the next card (see the "Card" attribute). If zero, the new card is inserted in front of the current card and the current card is left unchanged. In either case, if the current card on entry points to the "end-of-file", the new card is appended to the end of the list.

Notes:

- The function astSetFitsU can be used to indicate that no value is associated with a keyword.
- The function astSetFitsCM can be used to store a pure comment card (i.e. a card with a blank keyword).
- To assign a new value for an existing keyword within a FitsChan, first find the card describing the keyword using astFindFits, and then use one of the astSetFits<X> family to over-write the old value.
- If, on exit, there are no cards following the card written by this function, then the current card is left pointing at the "end-of-file".
- An error will be reported if the keyword name does not conform to FITS requirements.

---

astSetFitsCM Store a comment card in a FitsChan astSetFitsCM

Description: This function stores a comment card (i.e. a card with no keyword name or equals sign) within a FitsChan at the current card position. The new card can either over-write an existing card, or can be inserted as a new card into the FitsChan.

Synopsis: `void astSetFitsCM( AstFitsChan *this, const char *comment, int overwrite )`

Parameters:

this

Pointer to the FitsChan.

comment

A pointer to a null terminated string holding the text of the comment card. If a NULL pointer or a blank string is supplied, then a totally blank card is produced.

overwrite

If non-zero, the new card over-writes the current card, and the current card is incremented to refer to the next card (see the "Card" attribute). If zero, the new card is inserted in front of the current card and the current card is left unchanged. In either case, if the current card on entry points to the "end-of-file", the new card is appended to the end of the list.

Notes:

- If, on exit, there are no cards following the card written by this function, then the current card is left pointing at the "end-of-file".

---

astSetFitsU Store an undefined keyword value in a FitsChan astSetFitsU

Description: This function stores an undefined value for a named keyword within a FitsChan at the current card position. The new undefined value can either over-write an existing keyword value, or can be inserted as a new header card into the FitsChan.

Synopsis: `void astSetFitsU( AstFitsChan *this, const char *name, const char *comment, int overwrite )`

Parameters:

this

Pointer to the FitsChan.

name

Pointer to a null-terminated character string containing the FITS keyword name. This may be a complete FITS header card, in which case the keyword to use is extracted from it. No more than 80 characters are read from this string.

comment

A pointer to a null terminated string holding a comment to associated with the keyword. If a NULL pointer or a blank string is supplied, then any comment included in the string supplied for the "name" parameter is used instead. If "name" contains no comment, then any existing comment in the card being over-written is retained. Otherwise, no comment is stored with the card.

overwrite

If non-zero, the new card formed from the supplied keyword name and comment string over-writes the current card, and the current card is incremented to refer to the next card (see the "Card" attribute). If zero, the new card is inserted in front of the current card and the current card is left unchanged. In either case, if the current card on entry points to the "end-of-file", the new card is appended to the end of the list.

Notes:

- If, on exit, there are no cards following the card written by this function, then the current card is left pointing at the "end-of-file".
- An error will be reported if the keyword name does not conform to FITS requirements.

---

astSetRefPos Set the reference position in a specified celestial coordinate system astSetRefPos

Description: This function sets the reference position (see attributes RefRA and RefDec) using axis values (in radians) supplied within the celestial coordinate system represented by a supplied SkyFrame.

Synopsis: `void astSetRefPos( AstSpecFrame *this, AstSkyFrame *frm, double lon, double lat )`

Parameters:

this
Pointer to the SpecFrame.

frm
Pointer to the SkyFrame which defines the celestial coordinate system in which the longitude and latitude values are supplied. If NULL is supplied, then the supplied longitude and latitude values are assumed to be FK5 J2000 RA and Dec values.

lon
The longitude of the reference point, in the coordinate system represented by the supplied SkyFrame (radians).

lat
The latitude of the reference point, in the coordinate system represented by the supplied SkyFrame (radians).

---

astSetStatus Set the AST error status to an explicit value astSetStatus

Description: This function sets the AST error status to the value supplied. It does not cause any error message to be produced and should not be used as part of normal error reporting. Its purpose is simply to communicate to AST that an error has occurred in some other item of software.

For example, a source or sink function supplied as an argument to astChannel or astFitsChan might use this to signal that an input/output error has occurred. AST could then respond by terminating the current read or write operation.

Synopsis: `void astSetStatus( int status_value )`

Parameters:

status_value
The new error status value to be set.

Notes:

- If the AST error status is set to an error value, most AST functions will not execute and will simply return without action. To clear the error status and restore normal behaviour, use astClearStatus.

astSetUnc Store uncertainty information in a Region astSetUnc

Description: Each Region (of any class) can have an "uncertainty" which specifies the uncertainties associated with the boundary of the Region. This information is supplied in the form of a second Region. The uncertainty in any point on the boundary of a Region is found by shifting the associated "uncertainty" Region so that it is centred at the boundary point being considered. The area covered by the shifted uncertainty Region then represents the uncertainty in the boundary position. The uncertainty is assumed to be the same for all points.

The uncertainty is usually specified when the Region is created, but this function allows it to be changed at any time.

Synopsis: `void astSetUnc( AstRegion *this, AstRegion *unc )`

Parameters:

this

Pointer to the Region which is to be assigned a new uncertainty.

unc

Pointer to the new uncertainty Region. This must be of a class for which all instances are centro-symetric (e.g. Box, Circle, Ellipse, etc.) or be a Prism containing centro-symetric component Regions. A deep copy of the supplied Region will be taken, so subsequent changes to the uncertainty Region using the supplied pointer will have no effect on the Region "this".

astShiftMap Create a ShiftMap astShiftMap

Description: This function creates a new ShiftMap and optionally initialises its attributes.

A ShiftMap is a linear Mapping which shifts each axis by a specified constant value.

Synopsis: `AstShiftMap *astShiftMap( int ncoord, const double shift[], const char *options, ... )`

Parameters:

ncoord

The number of coordinate values for each point to be transformed (i.e. the number of dimensions of the space in which the points will reside). The same number is applicable to both input and output points.

shift

An array containing the values to be added on to the input coordinates in order to create the output coordinates. A separate value should be supplied for each coordinate.

options

Pointer to a null-terminated string containing an optional comma-separated list of attribute assignments to be used for initialising the new ShiftMap. The syntax used is identical to that for the astSet function and may include "printf" format specifiers identified by "%" symbols in the normal way.

...

If the "options" string contains "%" format specifiers, then an optional list of additional arguments may follow it in order to supply values to be substituted for these specifiers. The rules for supplying these are identical to those for the astSet function (and for the C "printf" function).

Returned Value:

astShiftMap()

A pointer to the new ShiftMap.

Notes:

- A null Object pointer (AST__NULL) will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

Status Handling:

The protected interface to this function includes an extra parameter at the end of the parameter list descirbed above. This parameter is a pointer to the integer inherited status variable: "int ∗status".

---

astShow Display a textual representation of an Object on standard output astShow

Description: This function displays a textual description of any AST Object on standard output. It is provided primarily as an aid to debugging.

Synopsis: `void astShow( AstObject *this )`

Parameters:

this
Pointer to the Object to be displayed.

Class Applicability:

Object
This function applies to all Objects.

---

astShowMesh Display a mesh of points covering the surface of a Region astShowMesh

Description: This function writes a table to standard output containing the axis values at a mesh of points covering the surface of the supplied Region. Each row of output contains a tab-separated list of axis values, one for each axis in the Frame encapsulated by the Region. The number of points in the mesh is determined by the MeshSize attribute.

The table is preceded by a given title string, and followed by a single line containing the word "ENDMESH".

Synopsis: `void astShowMesh( AstRegion *this, int format, const char *ttl )`

Parameters:

this
Pointer to the Region.

format
A boolean value indicating if the displayed axis values should be formatted according to the Format attribute associated with the Frame's axis. Otherwise, they are displayed as simple floating point values.

ttl
A title to display before displaying the first position.

---

astSimplify Simplify a Mapping astSimplify

Description: This function simplifies a Mapping (which may be a compound Mapping such as a CmpMap) to eliminate redundant computational steps, or to merge separate steps which can be performed more efficiently in a single operation.

As a simple example, a Mapping which multiplied coordinates by 5, and then multiplied the result by 10, could be simplified to a single step which multiplied by 50. Similarly, a Mapping which multiplied by 5, and then divided by 5, could be reduced to a simple copying operation.

This function should typically be applied to Mappings which have undergone substantial processing or have been formed by merging other Mappings. It is of potential benefit, for example, in reducing execution time if applied before using a Mapping to transform a large number of coordinates.

Synopsis: `AstMapping *astSimplify( AstMapping *this )`

Parameters:

this

Pointer to the original Mapping.

Class Applicability:

Mapping

This function applies to all Mappings.

FrameSet

If the supplied Mapping is a FrameSet, the returned Mapping will be a copy of the supplied FrameSet in which all the inter-Frame Mappings have been simplified.

Returned Value:

astSimplify()

A new pointer to the (possibly simplified) Mapping.

Notes:

- Mappings that have a set value for their Ident attribute are left unchanged after simplification. This is so that their individual identity is preserved. This restriction does not apply to the simplification of Frames.
- This function can safely be applied even to Mappings which cannot be simplified. If no simplification is possible, it behaves exactly like astClone and returns a pointer to the original Mapping.
- The Mapping returned by this function may not be independent of the original (even if simplification was possible), and modifying it may therefore result in indirect modification of the original. If a completely independent result is required, a copy should be made using astCopy.
- A null Object pointer (AST__NULL) will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

---

astSkyFrame Create a SkyFrame astSkyFrame

Description: This function creates a new SkyFrame and optionally initialises its attributes.

A SkyFrame is a specialised form of Frame which describes celestial longitude/latitude coordinate systems. The particular celestial coordinate system to be represented is specified by setting the SkyFrame's System attribute (currently, the default is ICRS) qualified, as necessary, by a mean Equinox value and/or an Epoch.

For each of the supported celestial coordinate systems, a SkyFrame can apply an optional shift of origin to create a coordinate system representing offsets within the celestial coordinate system from some specified point. This offset coordinate system can also be rotated to define new longitude and latitude axes. See attributes SkyRef, SkyRefIs and SkyRefP

All the coordinate values used by a SkyFrame are in radians. These may be formatted in more conventional ways for display by using astFormat.

Synopsis: `AstSkyFrame *astSkyFrame( const char *options, ... )`

Parameters:

options
Pointer to a null-terminated string containing an optional comma-separated list of attribute assignments to be used for initialising the new SkyFrame. The syntax used is identical to that for the astSet function and may include "printf" format specifiers identified by "%" symbols in the normal way. If no initialisation is required, a zero-length string may be supplied.

...
If the "options" string contains "%" format specifiers, then an optional list of additional arguments may follow it in order to supply values to be substituted for these specifiers. The rules for supplying these are identical to those for the astSet function (and for the C "printf" function).

Returned Value:

astSkyFrame()
A pointer to the new SkyFrame.

Examples:

`frame = astSkyFrame( "" );`
Creates a SkyFrame to describe the default ICRS celestial coordinate system.

`frame = astSkyFrame( "System = FK5, Equinox = J2005, Digits = 10" );`
Creates a SkyFrame to describe the FK5 celestial coordinate system, with a mean Equinox of J2005.0. Because especially accurate coordinates will be used, additional precision (10 digits) has been requested. This will be used when coordinate values are formatted for display.

`frame = astSkyFrame( "System = FK4, Equinox = 1955-sep-2" );`
Creates a SkyFrame to describe the old FK4 celestial coordinate system. A default Epoch value (B1950.0) is used, but the mean Equinox value is given explicitly as "1955-sep-2".

`frame = astSkyFrame( "System = GAPPT, Epoch = %s", date );`
Creates a SkyFrame to describe the Geocentric Apparent celestial coordinate system. The Epoch value, which specifies the date of observation, is obtained from a date/time string supplied via the string pointer "date".

Notes:

- Currently, the default celestial coordinate system is ICRS. However, this default may change in future as new astrometric standards evolve. The intention is to track the most modern appropriate standard. For this reason, you should use the default only if this is what you intend (and can tolerate any associated slight change in behaviour with future versions of this function). If you intend to use the ICRS system indefinitely, then you should specify it explicitly using an "options" value of "System=ICRS".
- Whichever celestial coordinate system is represented, it will have two axes. The first of these will be the longitude axis and the second will be the latitude axis. This order can be changed using astPermAxes if required.

- When conversion between two SkyFrames is requested (as when supplying SkyFrames to astConvert), account will be taken of the nature of the celestial coordinate systems they represent, together with any qualifying mean Equinox or Epoch values, etc. The AlignSystem attribute will also be taken into account. The results will therefore fully reflect the relationship between positions on the sky measured in the two systems.
- A null Object pointer (AST__NULL) will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

---

astSkyOffsetMap Returns a Mapping which goes from absolute coordinates to offset coordinates astSkyOffsetMap

Description: This function returns a Mapping in which the forward transformation transforms a position in the coordinate system given by the System attribute of the supplied SkyFrame, into the offset coordinate system specified by the SkyRef, SkyRefP and SkyRefIs attributes of the supplied SkyFrame.

A UnitMap is returned if the SkyFrame does not define an offset coordinate system.

Synopsis: `AstMapping *astSkyOffsetMap( AstSkyFrame *this )`

Parameters:

this
Pointer to the SkyFrame.

Returned Value:

astSkyOffsetMap()
Pointer to the returned Mapping.

Notes:

- A null Object pointer (AST__NULL) will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

---

astSlaAdd Add a celestial coordinate conversion to an SlaMap astSlaAdd

Description: This function adds one of the standard celestial coordinate system conversions provided by the SLALIB Positional Astronomy Library (Starlink User Note SUN/67) to an existing SlaMap.

When an SlaMap is first created (using astSlaMap), it simply performs a unit (null) Mapping. By using astSlaAdd (repeatedly if necessary), one or more coordinate conversion steps may then be added, which the SlaMap will perform in sequence. This allows multi-step conversions between a variety of celestial coordinate systems to be assembled out of the building blocks provided by SLALIB.

Normally, if an SlaMap's Invert attribute is zero (the default), then its forward transformation is performed by carrying out each of the individual coordinate conversions specified by astSlaAdd in the order given (i.e. with the most recently added conversion applied last).

This order is reversed if the SlaMap's Invert attribute is non-zero (or if the inverse transformation is requested by any other means) and each individual coordinate conversion is also replaced by its own inverse. This process inverts the overall effect of the SlaMap. In this case, the first conversion to be applied would be the inverse of the one most recently added.

Synopsis: `void astSlaAdd( AstSlaMap *this, const char *cvt, const double args[] )`

Parameters:

this
Pointer to the SlaMap.

cvt
Pointer to a null-terminated string which identifies the celestial coordinate conversion to be added to the SlaMap. See the "SLALIB Conversions" section for details of those available.

args
An array containing argument values for the celestial coordinate conversion. The number of arguments required, and hence the number of array elements used, depends on the conversion specified (see the "SLALIB Conversions" section). This array is ignored and a NULL pointer may be supplied if no arguments are needed.

Notes:

- All coordinate values processed by an SlaMap are in radians. The first coordinate is the celestial longitude and the second coordinate is the celestial latitude.
- When assembling a multi-stage conversion, it can sometimes be difficult to determine the most economical conversion path. For example, converting to the standard FK5 coordinate system as an intermediate stage is often sensible in formulating the problem, but may introduce unnecessary extra conversion steps. A solution to this is to include all the steps which are (logically) necessary, but then to use astSimplify to simplify the resulting SlaMap. The simplification process will eliminate any steps which turn out not to be needed.
- This function does not check to ensure that the sequence of coordinate conversions added to an SlaMap is physically meaningful.

SLALIB Conversions:

The following strings (which are case-insensitive) may be supplied via the "cvt" parameter to indicate which celestial coordinate conversion is to be added to the SlaMap. Each string is derived from the name of the SLALIB routine that performs the conversion and the relevant documentation (SUN/67) should be consulted for details. Where arguments are needed by the conversion, they are listed in parentheses. Values for these arguments should be given, via the "args" array, in the order indicated. The argument names match the corresponding SLALIB routine arguments and their values should be given using exactly the same units, time scale, calendar, etc. as described in SUN/67:

- "ADDET" (EQ): Add E-terms of aberration.
- "SUBET" (EQ): Subtract E-terms of aberration.
- "PREBN" (BEP0,BEP1): Apply Bessel-Newcomb pre-IAU 1976 (FK4) precession model.
- "PREC" (EP0,EP1): Apply IAU 1975 (FK5) precession model.
- "FK45Z" (BEPOCH): Convert FK4 to FK5 (no proper motion or parallax).
- "FK54Z" (BEPOCH): Convert FK5 to FK4 (no proper motion or parallax).
- "AMP" (DATE,EQ): Convert geocentric apparent to mean place.
- "MAP" (EQ,DATE): Convert mean place to geocentric apparent.
- "ECLEQ" (DATE): Convert ecliptic coordinates to FK5 J2000.0 equatorial.
- "EQECL" (DATE): Convert equatorial FK5 J2000.0 to ecliptic coordinates.
- "GALEQ": Convert galactic coordinates to FK5 J2000.0 equatorial.
- "EQGAL": Convert FK5 J2000.0 equatorial to galactic coordinates.

- `"HFK5Z"` (JEPOCH): Convert ICRS coordinates to FK5 J2000.0 equatorial.
- `"FK5HZ"` (JEPOCH): Convert FK5 J2000.0 equatorial coordinates to ICRS.
- `"GALSUP"`: Convert galactic to supergalactic coordinates.
- `"SUPGAL"`: Convert supergalactic coordinates to galactic.
- `"J2000H"`: Convert dynamical J2000.0 to ICRS.
- `"HJ2000"`: Convert ICRS to dynamical J2000.0.
- `"R2H"` (LAST): Convert RA to Hour Angle.
- `"H2R"` (LAST): Convert Hour Angle to RA.

For example, to use the `"ADDET"` conversion, which takes a single argument EQ, you should consult the documentation for the SLALIB routine SLA_ADDET. This describes the conversion in detail and shows that EQ is the Besselian epoch of the mean equator and equinox. This value should then be supplied to astSlaAdd in args[0].

In addition the following strings may be supplied for more complex conversions which do not correspond to any one single SLALIB routine (DIURAB is the magnitude of the diurnal aberration vector in units of `"day/(2.PI)"`, DATE is the Modified Julian Date of the observation, and (OBSX,OBSY,OBZ) are the Heliocentric-Aries-Ecliptic cartesian coordinates, in metres, of the observer):

- `"HPCEQ"` (DATE,OBSX,OBSY,OBSZ): Convert Helioprojective-Cartesian coordinates to J2000.0 equatorial.
- `"EQHPC"` (DATE,OBSX,OBSY,OBSZ): Convert J2000.0 equatorial coordinates to Helioprojective-Cartesian.
- `"HPREQ"` (DATE,OBSX,OBSY,OBSZ): Convert Helioprojective-Radial coordinates to J2000.0 equatorial.
- `"EQHPR"` (DATE,OBSX,OBSY,OBSZ): Convert J2000.0 equatorial coordinates to Helioprojective-Radial.
- `"HEEQ"` (DATE): Convert helio-ecliptic coordinates to J2000.0 equatorial.
- `"EQHE"` (DATE): Convert J2000.0 equatorial coordinates to helio-ecliptic.
- `"H2E"` (LAT,DIRUAB): Convert horizon coordinates to equatorial.
- `"E2H"` (LAT,DIURAB): Convert equatorial coordinates to horizon.

Note, the `"H2E"` and `"E2H"` conversions convert between topocentric horizon coordinates (azimuth,elevation), and apparent local equatorial coordinates (hour angle,declination). Thus, the effects of diurnal aberration are taken into account in the conversions but the effects of atmospheric refraction are not.

---

astSlaMap Create an SlaMap astSlaMap

Description: This function creates a new SlaMap and optionally initialises its attributes.

An SlaMap is a specialised form of Mapping which can be used to represent a sequence of conversions between standard celestial (longitude, latitude) coordinate systems.

When an SlaMap is first created, it simply performs a unit (null) Mapping on a pair of coordinates. Using the astSlaAdd function, a series of coordinate conversion steps may then be added, selected from those provided by the SLALIB Positional Astronomy Library (Starlink User Note SUN/67). This allows multi-step conversions between a variety of celestial coordinate systems to be assembled out of the building blocks provided by SLALIB.

For details of the individual coordinate conversions available, see the description of the astSlaAdd function.

Synopsis: `AstSlaMap *astSlaMap( int flags, const char *options, ... )`

Parameters:

flags

This parameter is reserved for future use and should currently always be set to zero.

options

Pointer to a null-terminated string containing an optional comma-separated list of attribute assignments to be used for initialising the new SlaMap. The syntax used is identical to that for the astSet function and may include "printf" format specifiers identified by "%" symbols in the normal way. If no initialisation is required, a zero-length string may be supplied.

...

If the "options" string contains "%" format specifiers, then an optional list of additional arguments may follow it in order to supply values to be substituted for these specifiers. The rules for supplying these are identical to those for the astSet function (and for the C "printf" function).

Returned Value:

astSlaMap()

A pointer to the new SlaMap.

Notes:

- The Nin and Nout attributes (number of input and output coordinates) for an SlaMap are both equal to 2. The first coordinate is the celestial longitude and the second coordinate is the celestial latitude. All coordinate values are in radians.
- A null Object pointer (AST__NULL) will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

---

astSpecAdd Add a spectral coordinate conversion to a SpecMap astSpecAdd

Description: This function adds one of the standard spectral coordinate system conversions listed below to an existing SpecMap.

When a SpecMap is first created (using astSpecMap), it simply performs a unit (null) Mapping. By using astSpecAdd (repeatedly if necessary), one or more coordinate conversion steps may then be added, which the SpecMap will perform in sequence. This allows multi-step conversions between a variety of spectral coordinate systems to be assembled out of the building blocks provided by this class.

Normally, if a SpecMap's Invert attribute is zero (the default), then its forward transformation is performed by carrying out each of the individual coordinate conversions specified by astSpecAdd in the order given (i.e. with the most recently added conversion applied last).

This order is reversed if the SpecMap's Invert attribute is non-zero (or if the inverse transformation is requested by any other means) and each individual coordinate conversion is also replaced by its own inverse. This process inverts the overall effect of the SpecMap. In this case, the first conversion to be applied would be the inverse of the one most recently added.

Synopsis: `void astSpecAdd( AstSpecMap *this, const char *cvt, const double args[] )`

Parameters:

this

Pointer to the SpecMap.

cvt
Pointer to a null-terminated string which identifies the spectral coordinate conversion to be added to the SpecMap. See the "Available Conversions" section for details of those available.

args
An array containing argument values for the spectral coordinate conversion. The number of arguments required, and hence the number of array elements used, depends on the conversion specified (see the "Available Conversions" section). This array is ignored and a NULL pointer may be supplied if no arguments are needed.

Notes:

- When assembling a multi-stage conversion, it can sometimes be difficult to determine the most economical conversion path. For example, when converting between reference frames, converting first to the heliographic reference frame as an intermediate stage is often sensible in formulating the problem, but may introduce unnecessary extra conversion steps. A solution to this is to include all the steps which are (logically) necessary, but then to use astSimplify to simplify the resulting SpecMap. The simplification process will eliminate any steps which turn out not to be needed.
- This function does not check to ensure that the sequence of coordinate conversions added to a SpecMap is physically meaningful.

Available Conversions:

The following strings (which are case-insensitive) may be supplied via the "cvt" parameter to indicate which spectral coordinate conversion is to be added to the SpecMap. Where arguments are needed by the conversion, they are listed in parentheses. Values for these arguments should be given, via the "args" array, in the order indicated. Units and argument names are described at the end of the list of conversions.

- "FRTOVL" (RF): Convert frequency to relativistic velocity.
- "VLTOFR" (RF): Convert relativistic velocity to Frequency.
- "ENTOFR": Convert energy to frequency.
- "FRTOEN": Convert frequency to energy.
- "WNTOFR": Convert wave number to frequency.
- "FRTOWN": Convert frequency to wave number.
- "WVTOFR": Convert wavelength (vacuum) to frequency.
- "FRTOWV": Convert frequency to wavelength (vacuum).
- "AWTOFR": Convert wavelength (air) to frequency.
- "FRTOAW": Convert frequency to wavelength (air).
- "VRTOVL": Convert radio to relativistic velocity.
- "VLTOVR": Convert relativistic to radio velocity.
- "VOTOVL": Convert optical to relativistic velocity.
- "VLTOVO": Convert relativistic to optical velocity.
- "ZOTOVL": Convert redshift to relativistic velocity.
- "VLTOZO": Convert relativistic velocity to redshift.
- "BTTOVL": Convert beta factor to relativistic velocity.
- "VLTOBT": Convert relativistic velocity to beta factor.

- "USF2HL" (VOFF,RA,DEC): Convert frequency from a user-defined reference frame to heliocentric.
- "HLF2US" (VOFF,RA,DEC): Convert frequency from heliocentric reference frame to user-defined.
- "TPF2HL" (OBSLON,OBSLAT,OBSALT,EPOCH,RA,DEC): Convert frequency from topocentric reference frame to heliocentric.
- "HLF2TP" (OBSLON,OBSLAT,OBSALT,EPOCH,RA,DEC): Convert frequency from heliocentric reference frame to topocentric.
- "GEF2HL" (EPOCH,RA,DEC): Convert frequency from geocentric reference frame to heliocentric.
- "HLF2GE" (EPOCH,RA,DEC): Convert frequency from heliocentric reference frame to geocentric.
- "BYF2HL" (EPOCH,RA,DEC): Convert frequency from barycentric reference frame to heliocentric.
- "HLF2BY" (EPOCH,RA,DEC): Convert frequency from heliocentric reference frame to barycentric.
- "LKF2HL" (RA,DEC): Convert frequency from kinematic LSR reference frame to heliocentric.
- "HLF2LK" (RA,DEC): Convert frequency from heliocentric reference frame to kinematic LSR.
- "LDF2HL" (RA,DEC): Convert frequency from dynamical LSR reference frame to heliocentric.
- "HLF2LD" (RA,DEC): Convert frequency from heliocentric reference frame to dynamical LSR.
- "LGF2HL" (RA,DEC): Convert frequency from local group reference frame to heliocentric.
- "HLF2LG" (RA,DEC): Convert frequency from heliocentric reference frame to local group.
- "GLF2HL" (RA,DEC): Convert frequency from galactic reference frame to heliocentric.
- "HLF2GL" (RA,DEC): Convert frequency from heliocentric reference frame to galactic.

The units for the values processed by the above conversions are as follows:

- all velocities: metres per second (positive if the source receeds from the observer).
- frequency: Hertz.
- all wavelengths: metres.
- energy: Joules.
- wave number: cycles per metre.

The arguments used in the above conversions are as follows:

- RF: Rest frequency (Hz).
- OBSALT: Geodetic altitude of observer (IAU 1975, metres).
- OBSLAT: Geodetic latitude of observer (IAU 1975, radians).
- OBSLON: Longitude of observer (radians - positive eastwards).
- EPOCH: Epoch of observation (UT1 expressed as a Modified Julian Date).
- RA: Right Ascension of source (radians, FK5 J2000).
- DEC: Declination of source (radians, FK5 J2000).

- VOFF: Velocity of the user-defined reference frame, towards the position given by RA and DEC, measured in the heliocentric reference frame.

If the SpecMap is 3-dimensional, source positions are provided by the values supplied to inputs 2 and 3 of the SpecMap (which are simply copied to outputs 2 and 3). Note, usable values are still required for the RA and DEC arguments in order to define the "user-defined" reference frame used by USF2HL and HLF2US. However, AST__BAD can be supplied for RA and DEC if the user-defined reference frame is not required.

---

astSpecFluxFrame Create a SpecFluxFrame astSpecFluxFrame

Description: This function creates a new SpecFluxFrame and optionally initialises its attributes.

A SpecFluxFrame combines a SpecFrame and a FluxFrame into a single 2-dimensional compound Frame. Such a Frame can for instance be used to describe a Plot of a spectrum in which the first axis represents spectral position and the second axis represents flux.

Synopsis: `AstSpecFluxFrame *astSpecFluxFrame( AstSpecFrame *frame1, AstFluxFrame *frame2, const char *options, ... )`

Parameters:

frame1

Pointer to the SpecFrame. This will form the first axis in the new SpecFluxFrame.

frame2

Pointer to the FluxFrame. This will form the second axis in the new SpecFluxFrame. The "SpecVal" attribute of this FluxFrame is not used by the SpecFluxFrame class and so may be set to AST__BAD when the FluxFrame is created.

options

Pointer to a null-terminated string containing an optional comma-separated list of attribute assignments to be used for initialising the new SpecFluxFrame. The syntax used is identical to that for the astSet function and may include "printf" format specifiers identified by "%" symbols in the normal way.

...

If the "options" string contains "%" format specifiers, then an optional list of additional arguments may follow it in order to supply values to be substituted for these specifiers. The rules for supplying these are identical to those for the astSet function (and for the C "printf" function).

Returned Value:

astSpecFluxFrame()

A pointer to the new SpecFluxFrame.

Notes:

- The supplied Frame pointers are stored directly, rather than being used to create deep copies of the supplied Frames. This means that any subsequent changes made to the Frames via the supplied pointers will result in equivalent changes being visible in the SpecFluxFrame.
- A null Object pointer (AST__NULL) will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

Status Handling:

The protected interface to this function includes an extra parameter at the end of the parameter list described above. This parameter is a pointer to the integer inherited status variable: "int *status".

astSpecFrame Create a SpecFrame astSpecFrame

Description: This function creates a new SpecFrame and optionally initialises its attributes.

A SpecFrame is a specialised form of one-dimensional Frame which represents various coordinate systems used to describe positions within an electro-magnetic spectrum. The particular coordinate system to be used is specified by setting the SpecFrame's System attribute (the default is wavelength) qualified, as necessary, by other attributes such as the rest frequency, the standard of rest, the epoch of observation, etc (see the description of the System attribute for details).

By setting a value for thr SpecOrigin attribute, a SpecFrame can be made to represent offsets from a given spectral position, rather than absolute

Synopsis: `AstSpecFrame *astSpecFrame( const char *options, ... )`

Parameters:

options

Pointer to a null-terminated string containing an optional comma-separated list of attribute assignments to be used for initialising the new SpecFrame. The syntax used is identical to that for the astSet function and may include "printf" format specifiers identified by "%" symbols in the normal way. If no initialisation is required, a zero-length string may be supplied.

...

If the "options" string contains "%" format specifiers, then an optional list of additional arguments may follow it in order to supply values to be substituted for these specifiers. The rules for supplying these are identical to those for the astSet function (and for the C "printf" function).

Returned Value:

astSpecFrame()

A pointer to the new SpecFrame.

Examples:

```
frame = astSpecFrame( "" );
```

Creates a SpecFrame to describe the default wavelength spectral coordinate system. The RestFreq attribute (rest frequency) is unspecified, so it will not be possible to align this SpecFrame with another SpecFrame on the basis of a velocity-based system. The standard of rest is also unspecified. This means that alignment will be possible with other SpecFrames, but no correction will be made for Doppler shift caused by change of rest frame during the alignment.

```
frame = astSpecFrame( "System=VELO, RestFreq=1.0E15, StdOfRest=LSRK"
);
```

Creates a SpecFrame describing a apparent radial velocity ("VELO") axis with rest frequency 1.0E15 Hz (about 3000 Angstroms), measured in the kinematic Local Standard of Rest ("LSRK"). Since the source position has not been specified (using attributes RefRA and RefDec), it will only be possible to align this SpecFrame with other SpecFrames which are also measured in the LSRK standard of rest.

Notes:

- When conversion between two SpecFrames is requested (as when supplying SpecFrames to astConvert), account will be taken of the nature of the spectral coordinate systems they represent, together with any qualifying rest frequency, standard of rest, epoch values, etc. The AlignSystem and AlignStdOfRest attributes will also be taken into account. The results will therefore fully reflect the relationship between positions measured in the two systems. In addition, any difference in the Unit attributes of the two systems will also be taken into account.

- A null Object pointer (AST__NULL) will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

---

astSpecMap Create a SpecMap astSpecMap

Description: This function creates a new SpecMap and optionally initialises its attributes.

An SpecMap is a specialised form of Mapping which can be used to represent a sequence of conversions between standard spectral coordinate systems. This includes conversions between frequency, wavelength, and various forms of velocity, as well as conversions between different standards of rest.

When a SpecMap is first created, it simply performs a unit (null) Mapping. Using the astSpecAdd function, a series of coordinate conversion steps may then be added, selected from the list of supported conversions. This allows multi-step conversions between a variety of spectral coordinate systems to be assembled out of the building blocks provided by this class.

For details of the individual coordinate conversions available, see the description of the astSpecAdd function.

Conversions are available to transform between standards of rest. Such conversions need to know the source position as an RA and DEC. This information can be supplied in the form of parameters for the relevant conversions, in which case the SpecMap is 1-dimensional, simply transforming the spectral axis values. This means that the same source position will always be used by the SpecMap. However, this may not be appropriate for an accurate description of a 3-D spectral cube, where changes of spatial position can produce significant changes in the Doppler shift introduced when transforming between standards of rest. For this situation, a 3-dimensional SpecMap can be created in which axes 2 and 3 correspond to the source RA and DEC The SpecMap simply copies values for axes 2 and 3 from input to output).

Synopsis: `AstSpecMap *astSpecMap( int nin, int flags, const char *options, ... )`

Parameters:

nin

The number of inputs to the Mapping (this will also equal the number of outputs). This value must be either 1 or 3. In either case, the first input and output correspoindis the spectral axis. For a 3-axis SpecMap, the second and third axes give the RA and DEC (J2000 FK5) of the source. This positional information is used by conversions which transform between standards of rest, and replaces the "RA" and "DEC" arguments for the individual conversions listed in description of the "SpecAdd" function.

flags

This parameter is reserved for future use and should currently always be set to zero.

options

Pointer to a null-terminated string containing an optional comma-separated list of attribute assignments to be used for initialising the new SpecMap. The syntax used is identical to that for the astSet function and may include "printf" format specifiers identified by "%" symbols in the normal way. If no initialisation is required, a zero-length string may be supplied.

...

If the "options" string contains "%" format specifiers, then an optional list of additional arguments may follow it in order to supply values to be substituted for these specifiers. The rules for supplying these are identical to those for the astSet function (and for the C "printf" function).

Returned Value:

astSpecMap()

A pointer to the new SpecMap.

Notes:

- The nature and units of the coordinate values supplied for the first input (i.e. the spectral input) of a SpecMap must be appropriate to the first conversion step applied by the SpecMap. For instance, if the first conversion step is "FRTOVL" (frequency to relativistic velocity), then the coordinate values for the first input should be frequency in units of Hz. Similarly, the nature and units of the coordinate values returned by a SpecMap will be determined by the last conversion step applied by the SpecMap. For instance, if the last conversion step is "VLTOVO" (relativistic velocity to optical velocity), then the coordinate values for the first output will be optical velocity in units of metres per second. See the description of the astSpecAdd function for the units expected and returned by each conversion.
- A null Object pointer (AST__NULL) will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

---

astSphMap Create a SphMap **astSphMap**

Description: This function creates a new SphMap and optionally initialises its attributes.

A SphMap is a Mapping which transforms points from a 3-dimensional Cartesian coordinate system into a 2-dimensional spherical coordinate system (longitude and latitude on a unit sphere centred at the origin). It works by regarding the input coordinates as position vectors and finding their intersection with the sphere surface. The inverse transformation always produces points which are a unit distance from the origin (i.e. unit vectors).

Synopsis: `AstSphMap *astSphMap( const char *options, ... )`

Parameters:

options

Pointer to a null-terminated string containing an optional comma-separated list of attribute assignments to be used for initialising the new SphMap. The syntax used is identical to that for the astSet function and may include "printf" format specifiers identified by "%" symbols in the normal way.

...

If the "options" string contains "%" format specifiers, then an optional list of additional arguments may follow it in order to supply values to be substituted for these specifiers. The rules for supplying these are identical to those for the astSet function (and for the C "printf" function).

Returned Value:

astSphMap()

A pointer to the new SphMap.

Notes:

- The spherical coordinates are longitude (positive anti-clockwise looking from the positive latitude pole) and latitude. The Cartesian coordinates are right-handed, with the x axis (axis 1) at zero longitude and latitude, and the z axis (axis 3) at the positive latitude pole.
- At either pole, the longitude is set to the value of the PolarLong attribute.
- If the Cartesian coordinates are all zero, then the longitude and latitude are set to the value AST__BAD.
- A null Object pointer (AST__NULL) will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

Status Handling:

The protected interface to this function includes an extra parameter at the end of the parameter list descirbed above. This parameter is a pointer to the integer inherited status variable: "int ∗status".

Status Handling:

The protected interface to this function includes an extra parameter at the end of the parameter list descirbed above. This parameter is a pointer to the integer inherited status variable: "int ∗status".

Status Handling:

The protected interface to this function includes an extra parameter at the end of the parameter list descirbed above. This parameter is a pointer to the integer inherited status variable: "int ∗status".

---

astStatus Obtain the current AST error status value astStatus

Description: This function returns the current value of the AST error status.

Synopsis: `int astStatus`

Returned Value:

astStatus
The AST error status value.

Notes:

- If the AST error status is set to an error value (after an error), most AST functions will not execute and will simply return without action. To clear the error status and restore normal behaviour, use astClearStatus.

---

astStcCatalogEntryLocation Create a StcCatalogEntryLocation astStcCatalogEntryLocation

Description: This function creates a new StcCatalogEntryLocation and optionally initialises its attributes.

The StcCatalogEntryLocation class is a sub-class of Stc used to describe the coverage of the datasets contained in some VO resource.

See http://hea-www.harvard.edu/∼arots/nvometa/STC.html

Synopsis: `AstStcCatalogEntryLocation ∗astStcCatalogEntryLocation( AstRegion ∗region, int ncoords, AstKeyMap ∗coords[], const char ∗options, ... )`

Parameters:

region

Pointer to the encapsulated Region.

ncoords

The length of the `"coords"` array. Supply zero if `"coords"` is NULL.

coords

Pointer to an array holding `"ncoords"` AstKeyMap pointers (if `"ncoords"` is zero, the supplied value is ignored). Each supplied KeyMap describes the contents of a single STC <AstroCoords> element, and should have elements with keys given by constants AST__STCNAME, AST__STCVALUE, AST__STCERROR, AST__STCRES, AST__STCSIZE, AST__STCPIXSZ. Any of these elements may be omitted, but no other elements should be included. If supplied, the AST__STCNAME element should be a vector of character string pointers holding the `"Name"` item for each axis in the coordinate system represented by `"region"`. Any other supplied elements should be scalar elements, each holding a pointer to a Region describing the associated item of ancillary information (error, resolution, size, pixel size or value). These Regions should describe a volume within the coordinate system represented by `"region"`.

options

Pointer to a null-terminated string containing an optional comma-separated list of attribute assignments to be used for initialising the new StcCatalogEntryLocation. The syntax used is identical to that for the astSet function and may include `"printf"` format specifiers identified by `"%"` symbols in the normal way.

...

If the `"options"` string contains `"%"` format specifiers, then an optional list of additional arguments may follow it in order to supply values to be substituted for these specifiers. The rules for supplying these are identical to those for the astSet function (and for the C `"printf"` function).

Returned Value:

astStcCatalogEntryLocation()

A pointer to the new StcCatalogEntryLocation.

Notes:

- A deep copy is taken of the supplied Region. This means that any subsequent changes made to the encapsulated Region using the supplied pointer will have no effect on the Stc.
- A null Object pointer (AST__NULL) will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

---

astStcObsDataLocation Create a StcObsDataLocation astStcObsDataLocation

Description: This function creates a new StcObsDataLocation and optionally initialises its attributes.

The StcObsDataLocation class is a sub-class of Stc used to describe the coverage of the datasets contained in some VO resource.

See http://hea-www.harvard.edu/∼arots/nvometa/STC.html

Synopsis: `AstStcObsDataLocation *astStcObsDataLocation( AstRegion *region, int ncoords, AstKeyMap *coords[], const char *options, ... )`

Parameters:

region

Pointer to the encapsulated Region.

ncoords

The length of the "coords" array. Supply zero if "coords" is NULL.

coords

Pointer to an array holding "ncoords" AstKeyMap pointers (if "ncoords" is zero, the supplied value is ignored). Each supplied KeyMap describes the contents of a single STC <AstroCoords> element, and should have elements with keys given by constants AST__STCNAME, AST__STCVALUE, AST__STCERROR, AST__STCRES, AST__STCSIZE, AST__STCPIXSZ. Any of these elements may be omitted, but no other elements should be included. If supplied, the AST__STCNAME element should be a vector of character string pointers holding the "Name" item for each axis in the coordinate system represented by "region". Any other supplied elements should be scalar elements, each holding a pointer to a Region describing the associated item of ancillary information (error, resolution, size, pixel size or value). These Regions should describe a volume within the coordinate system represented by "region".

options

Pointer to a null-terminated string containing an optional comma-separated list of attribute assignments to be used for initialising the new StcObsDataLocation. The syntax used is identical to that for the astSet function and may include "printf" format specifiers identified by "%" symbols in the normal way.

...

If the "options" string contains "%" format specifiers, then an optional list of additional arguments may follow it in order to supply values to be substituted for these specifiers. The rules for supplying these are identical to those for the astSet function (and for the C "printf" function).

Returned Value:

astStcObsDataLocation()

A pointer to the new StcObsDataLocation.

Notes:

- A deep copy is taken of the supplied Region. This means that any subsequent changes made to the encapsulated Region using the supplied pointer will have no effect on the Stc.
- A null Object pointer (AST__NULL) will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

---

astStcResourceProfile Create a StcResourceProfile astStcResourceProfile

Description: This function creates a new StcResourceProfile and optionally initialises its attributes.

The StcResourceProfile class is a sub-class of Stc used to describe the coverage of the datasets contained in some VO resource.

See http://hea-www.harvard.edu/∼arots/nvometa/STC.html

Synopsis: `AstStcResourceProfile *astStcResourceProfile( AstRegion *region, int ncoords, AstKeyMap *coords[], const char *options, ... )`

Parameters:

region

Pointer to the encapsulated Region.

ncoords

The length of the "coords" array. Supply zero if "coords" is NULL.

coords

Pointer to an array holding "ncoords" AstKeyMap pointers (if "ncoords" is zero, the supplied value is ignored). Each supplied KeyMap describes the contents of a single STC <AstroCoords> element, and should have elements with keys given by constants AST__STCNAME, AST__STCVALUE, AST__STCERROR, AST__STCRES, AST__STCSIZE, AST__STCPIXSZ. Any of these elements may be omitted, but no other elements should be included. If supplied, the AST__STCNAME element should be a vector of character string pointers holding the "Name" item for each axis in the coordinate system represented by "region". Any other supplied elements should be scalar elements, each holding a pointer to a Region describing the associated item of ancillary information (error, resolution, size, pixel size or value). These Regions should describe a volume within the coordinate system represented by "region".

options

Pointer to a null-terminated string containing an optional comma-separated list of attribute assignments to be used for initialising the new StcResourceProfile. The syntax used is identical to that for the astSet function and may include "printf" format specifiers identified by "%" symbols in the normal way.

...

If the "options" string contains "%" format specifiers, then an optional list of additional arguments may follow it in order to supply values to be substituted for these specifiers. The rules for supplying these are identical to those for the astSet function (and for the C "printf" function).

Returned Value:

astStcResourceProfile()

A pointer to the new StcResourceProfile.

Notes:

- A deep copy is taken of the supplied Region. This means that any subsequent changes made to the encapsulated Region using the supplied pointer will have no effect on the Stc.
- A null Object pointer (AST__NULL) will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

Status Handling:

The protected interface to this function includes an extra parameter at the end of the parameter list descirbed above. This parameter is a pointer to the integer inherited status variable: "int *status".

---

astStcSearchLocation Create a StcSearchLocation astStcSearchLocation

Description: This function creates a new StcSearchLocation and optionally initialises its attributes.

The StcSearchLocation class is a sub-class of Stc used to describe the coverage of a VO query.

See http://hea-www.harvard.edu/∼arots/nvometa/STC.html

Synopsis: `AstStcResourceProfile *astStcSearchLocation( AstRegion *region, int ncoords, AstKeyMap *coords[], const char *options, ... )`

Parameters:

region

Pointer to the encapsulated Region.

ncoords

The length of the "coords" array. Supply zero if "coords" is NULL.

coords

Pointer to an array holding "ncoords" AstKeyMap pointers (if "ncoords" is zero, the supplied value is ignored). Each supplied KeyMap describes the contents of a single STC <AstroCoords> element, and should have elements with keys given by constants AST__STCNAME, AST__STCVALUE, AST__STCERROR, AST__STCRES, AST__STCSIZE, AST__STCPIXSZ. Any of these elements may be omitted, but no other elements should be included. If supplied, the AST__STCNAME element should be a vector of character string pointers holding the "Name" item for each axis in the coordinate system represented by "region". Any other supplied elements should be scalar elements, each holding a pointer to a Region describing the associated item of ancillary information (error, resolution, size, pixel size or value). These Regions should describe a volume within the coordinate system represented by "region".

options

Pointer to a null-terminated string containing an optional comma-separated list of attribute assignments to be used for initialising the new StcSearchLocation. The syntax used is identical to that for the astSet function and may include "printf" format specifiers identified by "%" symbols in the normal way.

...

If the "options" string contains "%" format specifiers, then an optional list of additional arguments may follow it in order to supply values to be substituted for these specifiers. The rules for supplying these are identical to those for the astSet function (and for the C "printf" function).

Returned Value:

astStcSearchLocation()

A pointer to the new StcSearchLocation.

Notes:

- A deep copy is taken of the supplied Region. This means that any subsequent changes made to the encapsulated Region using the supplied pointer will have no effect on the Stc.
- A null Object pointer (AST__NULL) will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

Status Handling:

The protected interface to this function includes an extra parameter at the end of the parameter list descirbed above. This parameter is a pointer to the integer inherited status variable: "int ∗status".

---

astStcsChan Create an StcsChan astStcsChan

Description: This function creates a new StcsChan and optionally initialises its attributes.

A StcsChan is a specialised form of Channel which supports STC-S I/O operations. Writing an Object to an StcsChan (using astWrite) will, if the Object is suitable, generate an STC-S description of that Object, and reading from an StcsChan will create a new Object from its STC-S description.

Normally, when you use an StcsChan, you should provide "source" and "sink" functions which connect it to an external data store by reading and writing the resulting text. These functions should perform any conversions needed between external character encodings and the internal ASCII encoding. If no such functions are supplied, a Channel will read from standard input and write to standard output.

Alternatively, an XmlChan can be told to read or write from specific text files using the SinkFile and SourceFile attributes, in which case no sink or source function need be supplied.

Synopsis: `AstStcsChan *astStcsChan( const char *(* source)( void ), void (* sink)( const char * ), const char *options, ... )`

Parameters:

source

Pointer to a source function that takes no arguments and returns a pointer to a null-terminated string. If no value has been set for the SourceFile attribute, this function will be used by the StcsChan to obtain lines of input text. On each invocation, it should return a pointer to the next input line read from some external data store, and a NULL pointer when there are no more lines to read.

If "source" is NULL and no value has been set for the SourceFile attribute, the StcsChan will read from standard input instead.

sink

Pointer to a sink function that takes a pointer to a null-terminated string as an argument and returns void. If no value has been set for the SinkFile attribute, this function will be used by the StcsChan to deliver lines of output text. On each invocation, it should deliver the contents of the string supplied to some external data store.

If "sink" is NULL, and no value has been set for the SinkFile attribute, the StcsChan will write to standard output instead.

options

Pointer to a null-terminated string containing an optional comma-separated list of attribute assignments to be used for initialising the new StcsChan. The syntax used is identical to that for the astSet function and may include "printf" format specifiers identified by "%" symbols in the normal way.

...

If the "options" string contains "%" format specifiers, then an optional list of additional arguments may follow it in order to supply values to be substituted for these specifiers. The rules for supplying these are identical to those for the astSet function (and for the C "printf" function).

Returned Value:

astStcsChan()

A pointer to the new StcsChan.

Notes:

- If the external data source or sink uses a character encoding other than ASCII, the supplied source and sink functions should translate between the external character encoding and the internal ASCII encoding used by AST.
- A null Object pointer (AST__NULL) will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

---

astStripEscapes Remove AST escape sequences from a string astStripEscapes

Description: This function removes AST escape sequences from a supplied string, returning the resulting text as the function value. The behaviour of this function can be controlled by invoking the astEscapes function, which can be used to supress or enable the removal of escape sequences by this function.

AST escape sequences are used by the Plot class to modify the appearance and position of substrings within a plotted text string. See the "Escape" attribute for further information.

Synopsis: `const char *astStripEscapes( const char *text )`

Parameters:

text
Pointer to the string to be checked.

Returned Value:

astStripEscapes()
Pointer to the modified string. If no escape sequences were found in the supplied string, then a copy of the supplied pointer is returned. Otherwise, the pointer will point to a static buffer holding the modified text. This text will be over-written by subsequent invocations of this function. If the astEscapes function has been called indicating that escape sequences should not be stripped, then the supplied string is returned without change.

---

astSwitchMap Create a SwitchMap astSwitchMap

Description: This function creates a new SwitchMap and optionally initialises its attributes.

A SwitchMap is a Mapping which represents a set of alternate Mappings, each of which is used to transform positions within a particular region of the input or output coordinate system of the SwitchMap.

A SwitchMap can encapsulate any number of Mappings, but they must all have the same number of inputs (Nin attribute value) and the same number of outputs (Nout attribute value). The SwitchMap itself inherits these same values for its Nin and Nout attributes. Each of these Mappings represents a "route" through the switch, and are referred to as "route" Mappings below. Each route Mapping transforms positions between the input and output coordinate space of the entire SwitchMap, but only one Mapping will be used to transform any given position. The selection of the appropriate route Mapping to use with any given input position is made by another Mapping, called the "selector" Mapping. Each SwitchMap encapsulates two selector Mappings in addition to its route Mappings; one for use with the SwitchMap's forward transformation (called the "forward selector Mapping"), and one for use with the SwitchMap's inverse transformation (called the "inverse selector Mapping"). The forward selector Mapping must have the same number of inputs as the route Mappings, but should have only one output. Likewise, the inverse selector Mapping must have the same number of outputs as the route Mappings, but should have only one input.

When the SwitchMap is used to transform a position in the forward direction (from input to output), each supplied input position is first transformed by the forward transformation of the forward selector Mapping. This produces a single output value for each input position referred to as the selector value. The nearest integer to the selector value is found, and is used to index the array of route Mappings (the first supplied route Mapping has index 1, the second route Mapping has index 2, etc). If the nearest integer to the selector value is less than 1 or greater than the number of route Mappings, then the SwitchMap output position is set to a value of AST__BAD on every axis. Otherwise, the forward transformation of the selected route Mapping is used to transform the supplied input position to produce the SwitchMap output position.

When the SwitchMap is used to transform a position in the inverse direction (from "output" to "input"), each supplied "output" position is first transformed by the inverse transformation of the inverse selector Mapping. This produces a selector value for each "output" position. Again, the nearest integer to the selector value is found, and is used to index the array of route Mappings. If this selector index value is within the bounds of the array of route Mappings, then the inverse transformation of the selected route Mapping is used to transform the supplied "output" position to produce the SwitchMap "input" position. If the selector index value is outside the bounds of the array of route Mappings, then the SwitchMap "input" position is set to a value of AST__BAD on every axis.

In practice, appropriate selector Mappings should be chosen to associate a different route Mapping with each region of coordinate space. Note that the SelectorMap class of Mapping is particularly appropriate for this purpose.

If a compound Mapping contains a SwitchMap in series with its own inverse, the combination of the two adjacent SwitchMaps will be replaced by a UnitMap when the compound Mapping is simplified using astSimplify.

Synopsis: `AstSwitchMap *astSwitchMap( AstMapping *fsmap, AstMapping *ismap, int nroute, AstMapping *routemaps[], const char *options, ... )`

Parameters:

fsmap

Pointer to the forward selector Mapping. This must have a defined forward transformation, but need not have a defined inverse transformation. It must have one output, and the number of inputs must match the number of inputs of each of the supplied route Mappings. NULL may be supplied, in which case the SwitchMap will have an undefined forward Mapping.

ismap

Pointer to the inverse selector Mapping. This must have a defined inverse transformation, but need not have a defined forward transformation. It must have one input, and the number of outputs must match the number of outputs of each of the supplied route Mappings. NULL may be supplied, in which case the SwitchMap will have an undefined inverse Mapping.

nroute

The number of supplied route Mappings.

routemaps

An array of pointers to the route Mappings. All the supplied route Mappings must have common values for the Nin and Nout attributes, and these values define the number of inputs and outputs of the SwitchMap.

options

Pointer to a null-terminated string containing an optional comma-separated list of attribute assignments to be used for initialising the new SwitchMap. The syntax used is identical to that for the astSet function and may include "printf" format specifiers identified by "%" symbols in the normal way.

...

If the "options" string contains "%" format specifiers, then an optional list of additional arguments may follow it in order to supply values to be substituted for these specifiers. The rules for supplying these are identical to those for the astSet function (and for the C "printf" function).

Returned Value:

astSwitchMap()

A pointer to the new SwitchMap.

Notes:

- Note that the component Mappings supplied are not copied by astSwitchMap (the new SwitchMap simply retains a reference to them). They may continue to be used for other purposes, but should not be deleted. If a SwitchMap containing a copy of its component Mappings is required, then a copy of the SwitchMap should be made using astCopy.
- A null Object pointer (AST__NULL) will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

---

astTable Create a Table astTable

Description: This function creates a new empty Table and optionally initialises its attributes.

The Table class is a type of KeyMap that represents a two-dimensional table of values. The astMapGet... and astMapPut... methods provided by the KeyMap class should be used for storing and retrieving values from individual cells within a Table. Each entry in the KeyMap represents a single cell of the table and has an associated key of the form "<COL>(i)" where "<COL>" is the name of a table column and "i" is the row index (the first row is row 1). Keys of this form should always be used when using KeyMap methods to access entries within a Table.

Columns must be declared using the astAddColumn method before values can be stored within them. This also fixes the type and shape of the values that may be stored in any cell of the column. Cells may contain scalar or vector values of any data type supported by the KeyMap class. Multi-dimensional arrays may also be stored, but these must be vectorised when storing and retrieving them within a table cell. All cells within a single column must have the same type and shape (specified when the column is declared).

Tables may have parameters that describe global properties of the entire table. These are stored as entries in the parent KeyMap and can be access using the get and set method of the KeyMap class. However, parameters must be declared using the astAddParameter method before being accessed.

Note - since accessing entries within a KeyMap is a relatively slow process, it is not recommended to use the Table class to store very large tables.

Synopsis: `AstTable *astTable( const char *options, ... )`

Parameters:

options

Pointer to a null-terminated string containing an optional comma-separated list of attribute assignments to be used for initialising the new Table. The syntax used is identical to that for the astSet function and may include "printf" format specifiers identified by "%" symbols in the normal way.

...

If the "options" string contains "%" format specifiers, then an optional list of additional arguments may follow it in order to supply values to be substituted for these specifiers. The rules for supplying these are identical to those for the astSet function (and for the C "printf" function).

Returned Value:

astTable()

A pointer to the new Table.

Notes:

- A null Object pointer (AST__NULL) will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

Status Handling:

The protected interface to this function includes an extra parameter at the end of the parameter list described above. This parameter is a pointer to the integer inherited status variable: "int *status".

astTableSource Register a source function for accessing tables in FITS files astTableSource

Description: This function can be used to register a call-back function with a FitsChan. The registered function is called when-ever the FitsChan needs to read information from a binary table contained within a FITS file. This occurs if the astRead function is invoked to read a FrameSet from a set of FITS headers that use the "-TAB" algorithm to describe one or more axes. Such axes use a FITS binary table to store a look-up table of axis values. The FitsChan will fail to read such axes unless the "TabOK" attribute is set to a non-zero positive integer value. The table containing the axis values must be made available to the FitsChan either by storing the table contents in the FitsChan (using astPutTables or astPutTable) prior to invoking astRead or by registering a call-back function using astTableSource. The first method is possibly simpler, but requires that the name of the extension containing the table be known in advance. Since the table name is embedded in the FITS headers, the name is often not known in advance. If a call-back is registered, the FitsChan will determine the name of the required table and invoke the call-back function to supply the table at the point where it is needed (i.e. within the astRead method).

Synopsis: `void astTableSource( AstFitsChan *this, void (* tabsource)( AstFitsChan *, const char *, int, int, int * ) )`

Parameters:

this

Pointer to the FitsChan.

tabsource

Pointer to the table source function to use. It takes five arguments - the first is a pointer to the FitsChan, the second is a string holding the name of the FITS extension containing the required binary table ("EXTNAME"), the third is the integer FITS "EXTVER" header value for the required extension, the fourth is the integer FITS "EXTLEVEL" header value for the required extension, and the fifth is a pointer to the inherited integer status value.

The call-back should read the entire contents (header and data) of the binary table in the named extension of the external FITS file, storing the contents in a newly created FitsTable object. It should then store this FitsTable in the FitsChan using the astPutTables or astPutTable method, and finally annull its local copy of the FitsTable pointer. If the table cannot be read for any reason, or if any other error occurs, it should return a non-zero integer for the final (third) argument.

If "tabsource" is NULL, any registered call-back function will be removed.

Notes:

- Application code can pass arbitrary data (such as file descriptors, etc) to the table source function using the astPutChannelData function. The source function should use the astChannelData macro to retrieve this data.

astTest Test if an Object attribute value is set astTest

Description: This function returns a boolean result (0 or 1) to indicate whether a value has been explicitly set for one of an Object's attributes.

Synopsis: `int astTest( AstObject *this, const char *attrib )`

Parameters:

this

Pointer to the Object.

attrib
Pointer to a null-terminated character string containing the name of the attribute to be tested.

Class Applicability:

Object
This function applies to all Objects.

Returned Value:

astTest()
One if a value has previously been explicitly set for the attribute (and hasn't been cleared), otherwise zero.

Notes:

- Attribute names are not case sensitive and may be surrounded by white space.
- A value of zero will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.
- A value of zero will also be returned if this function is used to test a read-only attribute, although no error will result.

---

astTestFits See if a named keyword has a defined value in a FitsChan astTestFits

Description: This function serches for a named keyword in a FitsChan. If found, and if the keyword has a value associated with it, a non-zero value is returned. If the keyword is not found, or if it does not have an associated value, a zero value is returned.

Synopsis: `int astTestFits( AstFitsChan *this, const char *name, int *there )`

Parameters:

this
Pointer to the FitsChan.

name
Pointer to a null-terminated character string containing the FITS keyword name. This may be a complete FITS header card, in which case the keyword to use is extracted from it. No more than 80 characters are read from this string.

there
Pointer to an integer which will be returned holding a non-zero value if the keyword was found in the header, and zero otherwise. This parameter allows a distinction to be made between the case where a keyword is not present, and the case where a keyword is present but has no associated value. A NULL pointer may be supplied if this information is not required.

Returned Value:

astTestFits()
A value of zero is returned if the keyword was not found in the FitsChan or has no associated value. Otherwise, a value of one is returned.

Notes:

- The current card is left unchanged by this function.

- The card following the current card is checked first. If this is not the required card, then the rest of the FitsChan is searched, starting with the first card added to the FitsChan. Therefore cards should be accessed in the order they are stored in the FitsChan (if possible) as this will minimise the time spent searching for cards.
- An error will be reported if the keyword name does not conform to FITS requirements.
- Zero is returned as the function value if an error has already occurred, or if this function should fail for any reason.

---

astText Draw a text string for a Plot astText

Description: This function draws a string of text at a position specified in the physical coordinate system of a Plot. The physical position is transformed into graphical coordinates to determine where the text should appear within the plotting area.

Synopsis: `void astText( AstPlot *this, const char *text, const double pos[], const float up[], const char *just )`

Parameters:

this

Pointer to the Plot.

text

Pointer to a null-terminated character string containing the text to be drawn. Trailing white space is ignored.

pos

An array, with one element for each axis of the Plot, giving the physical coordinates of the point where the reference position of the text string is to be placed.

up

An array holding the components of a vector in the "up" direction of the text (in graphical coordinates). For example, to get horizontal text, the vector {0.0f,1.0f} should be supplied. For a basic Plot, 2 values should be supplied. For a Plot3D, 3 values should be supplied, and the actual up vector used is the projection of the supplied up vector onto the text plane specified by the current value of the Plot3D's Norm attribute.

just

Pointer to a null-terminated character string identifying the reference point for the text being drawn. The first character in this string identifies the reference position in the "up" direction and may be "B" (baseline), "C" (centre), "T" (top) or "M" (bottom). The second character identifies the side-to-side reference position and may be "L" (left), "C" (centre) or "R" (right). The string is case-insensitive, and only the first two characters are significant.

For example, a value of "BL" means that the left end of the baseline of the original (unrotated) text is to be drawn at the position given by "pos".

Notes:

- The Plot3D class currently does not interpret graphical escape sequences contained within text displayed using this method.
- Text is not drawn at positions which have any coordinate equal to the value AST__BAD (or where the transformation into graphical coordinates yields coordinates containing the value AST__BAD).
- If the plotting position is clipped (see astClip), then no text is drawn.
- An error results if the base Frame of the Plot is not 2-dimensional or (for a Plot3D) 3-dimensional.

- An error also results if the transformation between the current and base Frames of the Plot is not defined (i.e. the Plot's TranInverse attribute is zero).

---

astThread Determine the thread that owns an Object astThread

Description: Returns an integer that indicates whether the supplied Object (or Object pointer) is currently unlocked, or is currently locked by the running thread, or another thread.

Synopsis: `int astThread( AstObject *this, int ptr )`

Parameters:

this

Pointer to the Object to be checked.

ptr

If non-zero, returns information about the supplied Object pointer, rather than the Object structure itself. See "Object Pointers and Structures" below.

Returned Value:

astThread()

A value of AST__UNLOCKED is returned if the Object (or pointer) is currently unlocked (i.e. has been unlocked using astUnlock but has not yet been locked using astLock). A value of AST__RUNNING is returned if the Object (or pointer) is currently locked by the running thread. A value of AST__OTHER is returned if the Object (or pointer) is currently locked by the another thread.

Notes:

- This function attempts to execute even if the global error status is set, but no further error report will be made if it subsequently fails under these circumstances.
- This function is only available in the C interface.
- This function always returns AST__RUNNING if the AST library has been built without POSIX thread support (i.e. the "-with-pthreads" option was not specified when running the "configure" script).

Object Pointers and Structures:

At any one time, an AST Object can have several distinct pointers, any one of which can be used to access the Object structure. For instance, the astClone function will produce a new distinct pointer for a given Object. In fact, an AST "pointer" is not a real pointer at all - it is an identifier for a "handle" structure, encoded to make it look like a pointer. Each handle contains (amongst othere things) a "real" pointer to the Object structure. This allows more than one handle to refer to the same Object structure. So when you call astClone (for instance) you get back an identifier for a new handle that refers to the same Object as the supplied handle.

In order to use an Object for anything useful, it must be locked for use by the running thread (either implicitly at creation or explicitly using astLock). The identity of the thread is stored in both the Object structure, and in the handle that was passed to astLock (or returned by the constructor function). Thus it is possible for a thread to have active pointers for Objects that are currently locked by another thread. In general, if such a pointer is passed to an AST function an error will be reported indicating that the Object is currently locked by another thread. The two exceptions to this is that astAnnul can be used to annull such a pointer, and this function can be used to return information about the pointer.

The other practical consequence of this is that when astEnd is called, all active pointers currently owned by the running thread (at the current context level) are annulled. This includes pointers for Objects that are currently locked by other threads.

If the "ptr" parameter is zero, then the returned value describes the Object structure itself. If "ptr" is non-zero, then the returned value describes the supplied Object pointer (i.e. handle), rather than the Object structure.

---

astTimeAdd Add a time coordinate conversion to a TimeMap astTimeAdd

Description: This function adds one of the standard time coordinate system conversions listed below to an existing TimeMap.

When a TimeMap is first created (using astTimeMap), it simply performs a unit (null) Mapping. By using astTimeAdd (repeatedly if necessary), one or more coordinate conversion steps may then be added, which the TimeMap will perform in sequence. This allows multi-step conversions between a variety of time coordinate systems to be assembled out of the building blocks provided by this class.

Normally, if a TimeMap's Invert attribute is zero (the default), then its forward transformation is performed by carrying out each of the individual coordinate conversions specified by astTimeAdd in the order given (i.e. with the most recently added conversion applied last).

This order is reversed if the TimeMap's Invert attribute is non-zero (or if the inverse transformation is requested by any other means) and each individual coordinate conversion is also replaced by its own inverse. This process inverts the overall effect of the TimeMap. In this case, the first conversion to be applied would be the inverse of the one most recently added.

Synopsis: `void astTimeAdd( AstTimeMap *this, const char *cvt, const double args[] )`

Parameters:

this

Pointer to the TimeMap.

cvt

Pointer to a null-terminated string which identifies the time coordinate conversion to be added to the TimeMap. See the "Available Conversions" section for details of those available.

args

An array containing argument values for the time coordinate conversion. The number of arguments required, and hence the number of array elements used, depends on the conversion specified (see the "Available Conversions" section). This array is ignored and a NULL pointer may be supplied if no arguments are needed.

Notes:

- When assembling a multi-stage conversion, it can sometimes be difficult to determine the most economical conversion path. A solution to this is to include all the steps which are (logically) necessary, but then to use astSimplify to simplify the resulting TimeMap. The simplification process will eliminate any steps which turn out not to be needed.
- This function does not check to ensure that the sequence of coordinate conversions added to a TimeMap is physically meaningful.

Available Conversions:

The following strings (which are case-insensitive) may be supplied via the "cvt" parameter to indicate which time coordinate conversion is to be added to the TimeMap. Where arguments are needed by the conversion, they are listed in parentheses. Values for these arguments should be given, via the "args" array, in the order indicated. Units and argument names are described at the end of the list of conversions, and "MJD" means Modified Julian Date.

- `"MJDTOMJD"` (MJDOFF1,MJDOFF2): Convert MJD from one offset to another.
- `"MJDTOJD"` (MJDOFF,JDOFF): Convert MJD to Julian Date.
- `"JDTOMJD"` (JDOFF,MJDOFF): Convert Julian Date to MJD.
- `"MJDTOBEP"` (MJDOFF,BEPOFF): Convert MJD to Besselian epoch.
- `"BEPTOMJD"` (BEPOFF,MJDOFF): Convert Besselian epoch to MJD.
- `"MJDTOJEP"` (MJDOFF,JEPOFF): Convert MJD to Julian epoch.
- `"JEPTOMJD"` (JEPOFF,MJDOFF): Convert Julian epoch to MJD.
- `"TAITOUTC"` (MJDOFF): Convert a TAI MJD to a UTC MJD.
- `"UTCTOTAI"` (MJDOFF): Convert a UTC MJD to a TAI MJD.
- `"TAITOTT"` (MJDOFF): Convert a TAI MJD to a TT MJD.
- `"TTTOTAI"` (MJDOFF): Convert a TT MJD to a TAI MJD.
- `"TTTOTDB"` (MJDOFF, OBSLON, OBSLAT, OBSALT): Convert a TT MJD to a TDB MJD.
- `"TDBTOTT"` (MJDOFF, OBSLON, OBSLAT, OBSALT): Convert a TDB MJD to a TT MJD.
- `"TTTOTCG"` (MJDOFF): Convert a TT MJD to a TCG MJD.
- `"TCGTOTT"` (MJDOFF): Convert a TCG MJD to a TT MJD.
- `"TDBTOTCB"` (MJDOFF): Convert a TDB MJD to a TCB MJD.
- `"TCBTOTDB"` (MJDOFF): Convert a TCB MJD to a TDB MJD.
- `"UTTOGMST"` (MJDOFF): Convert a UT MJD to a GMST MJD.
- `"GMSTTOUT"` (MJDOFF): Convert a GMST MJD to a UT MJD.
- `"GMSTTOLMST"` (MJDOFF, OBSLON, OBSLAT): Convert a GMST MJD to a LMST MJD.
- `"LMSTTOGMST"` (MJDOFF, OBSLON, OBSLAT): Convert a LMST MJD to a GMST MJD.
- `"LASTTOLMST"` (MJDOFF, OBSLON, OBSLAT): Convert a GMST MJD to a LMST MJD.
- `"LMSTTOLAST"` (MJDOFF, OBSLON, OBSLAT): Convert a LMST MJD to a GMST MJD.
- `"UTTOUTC"` (DUT1): Convert a UT1 MJD to a UTC MJD.
- `"UTCTOUT"` (DUT1): Convert a UTC MJD to a UT1 MJD.
- `"LTTOUTC"` (LTOFF): Convert a Local Time MJD to a UTC MJD.
- `"UTCTOLT"` (LTOFF): Convert a UTC MJD to a Local Time MJD.

The units for the values processed by the above conversions are as follows:

- Julian epochs and offsets: Julian years
- Besselian epochs and offsets: Tropical years
- Modified Julian Dates and offsets: days
- Julian Dates and offsets: days

The arguments used in the above conversions are the zero-points used by the astTransform function. The axis values supplied and returned by astTransform are offsets away from these zero-points:

- MJDOFF: The zero-point being used with MJD values.

- JDOFF: The zero-point being used with Julian Date values.
- BEPOFF: The zero-point being used with Besselian epoch values.
- JEPOFF: The zero-point being used with Julian epoch values.
- OBSLON: Observer longitude in radians (+ve westwards).
- OBSLAT: Observer geodetic latitude (IAU 1975) in radians (+ve northwards).
- OBSALT: Observer geodetic altitude (IAU 1975) in metres.
- DUT1: The UT1-UTC value to use.
- LTOFF: The offset between Local Time and UTC (in hours, positive for time zones east of Greenwich).

---

astTimeFrame Create a TimeFrame astTimeFrame

Description: This function creates a new TimeFrame and optionally initialises its attributes.

A TimeFrame is a specialised form of one-dimensional Frame which represents various coordinate systems used to describe positions in time.

A TimeFrame represents a moment in time as either an Modified Julian Date (MJD), a Julian Date (JD), a Besselian epoch or a Julian epoch, as determined by the System attribute. Optionally, a zero point can be specified (using attribute TimeOrigin) which results in the TimeFrame representing time offsets from the specified zero point.

Even though JD and MJD are defined as being in units of days, the TimeFrame class allows other units to be used (via the Unit attribute) on the basis of simple scalings (60 seconds = 1 minute, 60 minutes = 1 hour, 24 hours = 1 day, 365.25 days = 1 year). Likewise, Julian epochs can be described in units other than the usual years. Besselian epoch are always represented in units of (tropical) years.

The TimeScale attribute allows the time scale to be specified (that is, the physical proces used to define the rate of flow of time). MJD, JD and Julian epoch can be used to represent a time in any supported time scale. However, Besselian epoch may only be used with the "TT" (Terrestrial Time) time scale. The list of supported time scales includes universal time and siderial time. Strictly, these represent angles rather than time scales, but are included in the list since they are in common use and are often thought of as time scales.

When a time value is formatted it can be formated either as a simple floating point value, or as a Gregorian date (see the Format attribute).

Synopsis: `AstTimeFrame *astTimeFrame( const char *options, ... )`

Parameters:

options

Pointer to a null-terminated string containing an optional comma-separated list of attribute assignments to be used for initialising the new TimeFrame. The syntax used is identical to that for the astSet function and may include "printf" format specifiers identified by "%" symbols in the normal way. If no initialisation is required, a zero-length string may be supplied.

...

If the "options" string contains "%" format specifiers, then an optional list of additional arguments may follow it in order to supply values to be substituted for these specifiers. The rules for supplying these are identical to those for the astSet function (and for the C "printf" function).

Returned Value:

astTimeFrame()

A pointer to the new TimeFrame.

Notes:

- When conversion between two TimeFrames is requested (as when supplying TimeFrames to astConvert), account will be taken of the nature of the time coordinate systems they represent, together with any qualifying time scale, offset, unit, etc. The AlignSystem and AlignTimeScale attributes will also be taken into account.
- A null Object pointer (AST__NULL) will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

---

astTimeMap Create a TimeMap astTimeMap

Description: This function creates a new TimeMap and optionally initialises its attributes.

A TimeMap is a specialised form of 1-dimensional Mapping which can be used to represent a sequence of conversions between standard time coordinate systems.

When a TimeMap is first created, it simply performs a unit (null) Mapping. Using the astTimeAdd function, a series of coordinate conversion steps may then be added. This allows multi-step conversions between a variety of time coordinate systems to be assembled out of a set of building blocks.

For details of the individual coordinate conversions available, see the description of the astTimeAdd function.

Synopsis: `AstTimeMap *astTimeMap( int flags, const char *options, ... )`

Parameters:

flags

This parameter is reserved for future use and should currently always be set to zero.

options

Pointer to a null-terminated string containing an optional comma-separated list of attribute assignments to be used for initialising the new TimeMap. The syntax used is identical to that for the astSet function and may include "printf" format specifiers identified by "%" symbols in the normal way. If no initialisation is required, a zero-length string may be supplied.

...

If the "options" string contains "%" format specifiers, then an optional list of additional arguments may follow it in order to supply values to be substituted for these specifiers. The rules for supplying these are identical to those for the astSet function (and for the C "printf" function).

Returned Value:

astTimeMap()

A pointer to the new TimeMap.

Notes:

- The nature and units of the coordinate values supplied for the first input (i.e. the time input) of a TimeMap must be appropriate to the first conversion step applied by the TimeMap. For instance, if the first conversion step is "MJDTOBEP" (Modified Julian Date to Besselian epoch) then the coordinate values for the first input should be date in units of days. Similarly, the nature and units of the coordinate values returned by a TimeMap will be determined by the last conversion step applied by the TimeMap.
- A null Object pointer (AST__NULL) will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

astToString Create an in-memory serialisation of an Object astToString

Description: This function returns a string holding a minimal textual serialisation of the supplied AST Object. The Object can re re-created from the serialisation using astFromString.

Synopsis: `char *astToString( AstObject *this )`

Parameters:

this
Pointer to the Object to be serialised.

Returned Value:

astToString()
Pointer to dynamically allocated memory holding the serialisation, or NULL if an error occurs. The pointer should be freed when no longer needed using astFree.

astTran1 Transform 1-dimensional coordinates astTran1

Description: This function applies a Mapping to transform the coordinates of a set of points in one dimension.

Synopsis: `void astTran1( AstMapping *this, int npoint, const double xin[], int forward, double xout[] )`

Parameters:

this
Pointer to the Mapping to be applied.

npoint
The number of points to be transformed.

xin
An array of "npoint" coordinate values for the input (untransformed) points.

forward
A non-zero value indicates that the Mapping's forward coordinate transformation is to be applied, while a zero value indicates that the inverse transformation should be used.

xout
An array (with "npoint" elements) into which the coordinates of the output (transformed) points will be written.

Notes:

- The Mapping supplied must have the value 1 for both its Nin and Nout attributes.

astTran2 Transform 2-dimensional coordinates astTran2

Description: This function applies a Mapping to transform the coordinates of a set of points in two dimensions.

Synopsis: `void astTran2( AstMapping *this, int npoint, const double xin[], const double yin[], int forward, double xout[], double yout[] )`

Parameters:

this
Pointer to the Mapping to be applied.

npoint
The number of points to be transformed.

xin
An array of "npoint" X-coordinate values for the input (untransformed) points.

yin
An array of "npoint" Y-coordinate values for the input (untransformed) points.

forward
A non-zero value indicates that the Mapping's forward coordinate transformation is to be applied, while a zero value indicates that the inverse transformation should be used.

xout
An array (with "npoint" elements) into which the X-coordinates of the output (transformed) points will be written.

yout
An array (with "npoint" elements) into which the Y-coordinates of the output (transformed) points will be written.

Notes:

- The Mapping supplied must have the value 2 for both its Nin and Nout attributes.

---

astTranGrid Transform a grid of positions astTranGrid

Description: This function uses the supplied Mapping to transforms a regular square grid of points covering a specified box. It attempts to do this quickly by first approximating the Mapping with a linear transformation applied over the whole region of the input grid which is being used. If this proves to be insufficiently accurate, the input region is sub-divided into two along its largest dimension and the process is repeated within each of the resulting sub-regions. This process of sub-division continues until a sufficiently good linear approximation is found, or the region to which it is being applied becomes too small (in which case the original Mapping is used directly).

Synopsis:

```
void astTranGrid( AstMapping *this, int ncoord_in, const int lbnd[], const int
ubnd[], double tol, int maxpix, int forward, int ncoord_out, int outdim, double *out
);
```

Parameters:

this
Pointer to the Mapping to be applied.

ncoord_in
The number of coordinates being supplied for each box corner (i.e. the number of dimensions of the space in which the input points reside).

lbnd
Pointer to an array of integers, with "ncoord_in" elements, containing the coordinates of the centre of the first pixel in the input grid along each dimension.

ubnd
Pointer to an array of integers, with "ncoord_in" elements, containing the coordinates of the centre of the last pixel in the input grid along each dimension.

Note that "lbnd" and "ubnd" together define the shape and size of the input grid, its extent along a particular (j'th) dimension being ubnd[j]-lbnd[j]+1 (assuming the index "j" to be zero-based). They also define the input grid's coordinate system, each pixel having unit extent along each dimension with integral coordinate values at its centre.

tol

The maximum tolerable geometrical distortion which may be introduced as a result of approximating non-linear Mappings by a set of piece-wise linear transformations. This should be expressed as a displacement within the output coordinate system of the Mapping.

If piece-wise linear approximation is not required, a value of zero may be given. This will ensure that the Mapping is used without any approximation, but may increase execution time.

If the value is too high, discontinuities between the linear approximations used in adjacent panel will be higher. If this is a problem, reduce the tolerance value used.

maxpix

A value which specifies an initial scale size (in input grid points) for the adaptive algorithm which approximates non-linear Mappings with piece-wise linear transformations. Normally, this should be a large value (larger than any dimension of the region of the input grid being used). In this case, a first attempt to approximate the Mapping by a linear transformation will be made over the entire input region.

If a smaller value is used, the input region will first be divided into sub-regions whose size does not exceed "maxpix" grid points in any dimension. Only at this point will attempts at approximation commence.

This value may occasionally be useful in preventing false convergence of the adaptive algorithm in cases where the Mapping appears approximately linear on large scales, but has irregularities (e.g. holes) on smaller scales. A value of, say, 50 to 100 grid points can also be employed as a safeguard in general-purpose software, since the effect on performance is minimal.

If too small a value is given, it will have the effect of inhibiting linear approximation altogether (equivalent to setting "tol" to zero). Although this may degrade performance, accurate results will still be obtained.

forward

A non-zero value indicates that the Mapping's forward coordinate transformation is to be applied, while a zero value indicates that the inverse transformation should be used.

ncoord_out

The number of coordinates being generated by the Mapping for each output point (i.e. the number of dimensions of the space in which the output points reside). This need not be the same as "ncoord_in".

outdim

The number of elements along the second dimension of the "out" array (which will contain the output coordinates). The value given should not be less than the number of points in the grid.

out

The address of the first element in a 2-dimensional array of shape "[ncoord_out][outdim]", into which the coordinates of the output (transformed) points will be written. These will be stored such that the value of coordinate number "coord" for output point number "point" will be found in element "out[coord][point]". The points are ordered such that the first axis of the input grid changes most rapidly. For example, if the input grid is 2-dimensional and extends from (2,-1) to (3,1), the output points will be stored in the order (2,-1), (3, -1), (2,0), (3,0), (2,1), (3,1).

Notes:

- If the forward coordinate transformation is being applied, the Mapping supplied must have the value of "ncoord_in" for its Nin attribute and the value of "ncoord_out" for its Nout attribute. If the inverse transformation is being applied, these values should be reversed.

---

astTranMap Create a TranMap astTranMap

Description: This function creates a new TranMap and optionally initialises its attributes.

A TranMap is a Mapping which combines the forward transformation of a supplied Mapping with the inverse transformation of another supplied Mapping, ignoring the un-used transformation in each Mapping (indeed the un-used transformation need not exist).

When the forward transformation of the TranMap is referred to, the transformation actually used is the forward transformation of the first Mapping supplied when the TranMap was constructed. Likewise, when the inverse transformation of the TranMap is referred to, the transformation actually used is the inverse transformation of the second Mapping supplied when the TranMap was constructed.

Synopsis: `AstTranMap *astTranMap( AstMapping *map1, AstMapping *map2, const char *options, ... )`

Parameters:

map1

Pointer to the first component Mapping, which defines the forward transformation.

map2

Pointer to the second component Mapping, which defines the inverse transformation.

options

Pointer to a null-terminated string containing an optional comma-separated list of attribute assignments to be used for initialising the new TranMap. The syntax used is identical to that for the astSet function and may include "printf" format specifiers identified by "%" symbols in the normal way.

...

If the "options" string contains "%" format specifiers, then an optional list of additional arguments may follow it in order to supply values to be substituted for these specifiers. The rules for supplying these are identical to those for the astSet function (and for the C "printf" function).

Returned Value:

astTranMap()

A pointer to the new TranMap.

Notes:

- The number of output coordinates generated by the two Mappings (their Nout attribute) must be equal, as must the number of input coordinates accepted by each Mapping (their Nin attribute).
- The forward transformation of the first Mapping must exist.
- The inverse transformation of the second Mapping must exist.
- Note that the component Mappings supplied are not copied by astTranMap (the new TranMap simply retains a reference to them). They may continue to be used for other purposes, but should not be deleted. If a TranMap containing a copy of its component Mappings is required, then a copy of the TranMap should be made using astCopy.
- A null Object pointer (AST__NULL) will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

Status Handling:

The protected interface to this function includes an extra parameter at the end of the parameter list described above. This parameter is a pointer to the integer inherited status variable: "int *status".

astTranN Transform N-dimensional coordinates **astTranN**

Description: This function applies a Mapping to transform the coordinates of a set of points in an arbitrary number of dimensions. It is the appropriate routine to use if the coordinates are not purely 1- or 2-dimensional and are stored in a single array (which they need not fill completely).

If the coordinates are not stored in a single array, then the astTranP function might be more suitable.

Synopsis: `void astTranN( AstMapping *this, int npoint, int ncoord_in, int indim, const double *in, int forward, int ncoord_out, int outdim, double *out )`

Parameters:

this

Pointer to the Mapping to be applied.

npoint

The number of points to be transformed.

ncoord_in

The number of coordinates being supplied for each input point (i.e. the number of dimensions of the space in which the input points reside).

indim

The number of elements along the second dimension of the "in" array (which contains the input coordinates). This value is required so that the coordinate values can be correctly located if they do not entirely fill this array. The value given should not be less than "npoint".

in

The address of the first element in a 2-dimensional array of shape "[ncoord_in][indim]", containing the coordinates of the input (untransformed) points. These should be stored such that the value of coordinate number "coord" for input point number "point" is found in element "in[coord][point]".

forward

A non-zero value indicates that the Mapping's forward coordinate transformation is to be applied, while a zero value indicates that the inverse transformation should be used.

ncoord_out

The number of coordinates being generated by the Mapping for each output point (i.e. the number of dimensions of the space in which the output points reside). This need not be the same as "ncoord_in".

outdim

The number of elements along the second dimension of the "out" array (which will contain the output coordinates). This value is required so that the coordinate values can be correctly located if they will not entirely fill this array. The value given should not be less than "npoint".

out

The address of the first element in a 2-dimensional array of shape "[ncoord_out][outdim]", into which the coordinates of the output (transformed) points will be written. These will be stored such that the value of coordinate number "coord" for output point number "point" will be found in element "out[coord][point]".

Notes:

- If the forward coordinate transformation is being applied, the Mapping supplied must have the value of "ncoord_in" for its Nin attribute and the value of "ncoord_out" for its Nout attribute. If the inverse transformation is being applied, these values should be reversed.

---

astTranP Transform N-dimensional coordinates held in separate arrays astTranP

Description: This function applies a Mapping to transform the coordinates of a set of points in an arbitrary number of dimensions. It is the appropriate routine to use if the coordinates are not purely 1- or 2-dimensional and are stored in separate arrays, since each coordinate array is located by supplying a separate pointer to it.

If the coordinates are stored in a single (2-dimensional) array, then the astTranN function might be more suitable.

Synopsis: `void astTranP( AstMapping *this, int npoint, int ncoord_in, const double *ptr_in[], int forward, int ncoord_out, double *ptr_out[] )`

Parameters:

this
Pointer to the Mapping to be applied.

npoint
The number of points to be transformed.

ncoord_in
The number of coordinates being supplied for each input point (i.e. the number of dimensions of the space in which the input points reside).

ptr_in
An array of pointers to double, with "ncoord_in" elements. Element "ptr_in[coord]" should point at the first element of an array of double (with "npoint" elements) which contain the values of coordinate number "coord" for each input (untransformed) point. The value of coordinate number "coord" for input point number "point" is therefore given by "ptr_in[coord][point]" (assuming both indices are zero-based).

forward
A non-zero value indicates that the Mapping's forward coordinate transformation is to be applied, while a zero value indicates that the inverse transformation should be used.

ncoord_out
The number of coordinates being generated by the Mapping for each output point (i.e. the number of dimensions of the space in which the output points reside). This need not be the same as "ncoord_in".

ptr_out
An array of pointers to double, with "ncoord_out" elements. Element "ptr_out[coord]" should point at the first element of an array of double (with "npoint" elements) into which the values of coordinate number "coord" for each output (transformed) point will be written. The value of coordinate number "coord" for output point number "point" will therefore be found in "ptr_out[coord][point]".

Notes:

- If the forward coordinate transformation is being applied, the Mapping supplied must have the value of "ncoord_in" for its Nin attribute and the value of "ncoord_out" for its Nout attribute. If the inverse transformation is being applied, these values should be reversed.
- This routine is not available in the Fortran 77 interface to the AST library.

---

astTune Set or get an integer-valued AST global tuning parameter astTune

Description: This function returns the current value of an integer-valued AST global tuning parameter, optionally storing a new value for the parameter. For character-valued tuning parameters, see astTuneC.

Synopsis: `int astTune( const char *name, int value )`

Parameters:

name
The name of the tuning parameter (case-insensitive).

value
The new value for the tuning parameter. If this is AST__TUNULL, the existing current value will be retained.

Returned Value:

astTune()
The original value of the tuning parameter. A default value will be returned if no value has been set for the parameter.

Notes:

- This function attempts to execute even if the AST error status is set on entry, although no further error report will be made if it subsequently fails under these circumstances.
- All threads in a process share the same AST tuning parameters values.

Tuning Parameters :

ObjectCaching
A boolean flag which indicates what should happen to the memory occupied by an AST Object when the Object is deleted (i.e. when its reference count falls to zero or it is deleted using astDelete). If this is zero, the memory is simply freed using the systems "free" function. If it is non-zero, the memory is not freed. Instead a pointer to it is stored in a pool of such pointers, all of which refer to allocated but currently unused blocks of memory. This allows AST to speed up subsequent Object creation by re-using previously allocated memory blocks rather than allocating new memory using the systems malloc function. The default value for this parameter is zero. Setting it to a non-zero value will result in Object memory being cached in future. Setting it back to zero causes any memory blocks currently in the pool to be freed. Note, this tuning parameter only controls the caching of memory used to store AST Objects. To cache other memory blocks allocated by AST, use MemoryCaching.

MemoryCaching
A boolean flag similar to ObjectCaching except that it controls caching of all memory blocks of less than 300 bytes allocated by AST (whether for internal or external use), not just memory used to store AST Objects.

---

astTuneC Set or get a character-valued AST global tuning parameter astTuneC

Description: This function returns the current value of a character-valued AST global tuning parameter, optionally storing a new value for the parameter. For integer-valued tuning parameters, see astTune.

Synopsis: `void astTuneC( const char *name, const char *value, char *buff, int bufflen )`

Parameters:

name
The name of the tuning parameter (case-insensitive).

value
The new value for the tuning parameter. If this is NULL, the existing current value will be retained.

buff
A character string in which to return the original value of the tuning parameter. An error will be reported if the buffer is too small to hold the value. NULL may be supplied if the old value is not required.

bufflen
The size of the supplied "buff" array. Ignored if "buff" is NULL.

Notes:

- This function attempts to execute even if the AST error status is set on entry, although no further error report will be made if it subsequently fails under these circumstances.
- All threads in a process share the same AST tuning parameters values.

Tuning Parameters :

HRDel
A string to be drawn following the hours field in a formatted sky axis value when "g" format is in use (see the Format attribute). This string may include escape sequences to produce super-scripts, etc. (see the Escapes attribute for details of the escape sequences allowed). The default value is "%-%^50+%s70+h%+" which produces a super-script "h".

MNDel
A string to be drawn following the minutes field in a formatted sky axis value when "g" format is in use. The default value is "%-%^50+%s70+m%+" which produces a super-script "m".

SCDel
A string to be drawn following the seconds field in a formatted sky axis value when "g" format is in use. The default value is "%-%^50+%s70+s%+" which produces a super-script "s".

DGDel
A string to be drawn following the degrees field in a formatted sky axis value when "g" format is in use. The default value is "%-%^53+%s60+o%+" which produces a super-script "o".

AMDel
A string to be drawn following the arc-minutes field in a formatted sky axis value when "g" format is in use. The default value is "%-%^20+%s85+'%+" which produces a super-script "'" (single quote).

ASDel
A string to be drawn following the arc-seconds field in a formatted sky axis value when "g" format is in use. The default value is "%-%^20+%s85+\"%+" which produces a super-script """ (double quote).

EXDel
A string to be drawn to introduce the exponent in a value when "g" format is in use. The default value is "10%-%^50+%s70+" which produces "10" followed by the exponent as a super-script.

astUinterp Perform sub-pixel interpolation on a grid of data **astUinterp**

Description: This is a fictitious function which does not actually exist. Instead, this description constitutes a template so that you may implement a function with this interface for yourself (and give it any name you wish). A pointer to such a function may be passed via the "finterp" parameter of the astResample<X> functions (q.v.) in order to perform sub-pixel interpolation during resampling of gridded data (you must also set the "interp" parameter of astResample<X> to the value AST__UINTERP). This allows you to use your own interpolation algorithm in addition to those which are pre-defined.

The function interpolates an input grid of data (and, optionally, processes associated statistical variance estimates) at a specified set of points.

Synopsis: `void astUinterp( int ndim_in, const int lbnd_in[], const int ubnd_in[], const <Xtype> in[], const <Xtype> in_var[], int npoint, const int offset[], const double *const coords[], const double params[], int flags, <Xtype> badval, <Xtype> out[], <Xtype> out_var[], int *nbad )`

Parameters:

ndim_in

The number of dimensions in the input grid. This will be at least one.

lbnd_in

Pointer to an array of integers, with "ndim_in" elements, containing the coordinates of the centre of the first pixel in the input grid along each dimension.

ubnd_in

Pointer to an array of integers, with "ndim_in" elements, containing the coordinates of the centre of the last pixel in the input grid along each dimension.

Note that "lbnd_in" and "ubnd_in" together define the shape, size and coordinate system of the input grid in the same way as they do in astResample<X>.

in

Pointer to an array, with one element for each pixel in the input grid, containing the input data. This will be the same array as was passed to astResample<X> via the "in" parameter. The numerical type of this array should match that of the data being processed.

in_var

Pointer to an optional second array with the same size and type as the "in" array. If given, this will contain the set of variance values associated with the input data and will be the same array as was passed to astResample<X> via the "in_var" parameter.

If no variance values are being processed, this will be a NULL pointer.

npoint

The number of points at which the input grid is to be interpolated. This will be at least one.

offset

Pointer to an array of integers with "npoint" elements. For each interpolation point, this will contain the zero-based index in the "out" (and "out_var") array(s) at which the interpolated value (and its variance, if required) should be stored. For example, the interpolated value for point number "point" should be stored in "out[offset[point]]" (assuming the index "point" is zero-based).

coords

An array of pointers to double, with "ndim_in" elements. Element "coords[coord]" will point at the first element of an array of double (with "npoint" elements) which contains the values of coordinate number "coord" for each interpolation point. The value of coordinate number "coord" for interpolation point number "point" is therefore given by "coords[coord][point]" (assuming both indices are zero-based).

If any interpolation point has any of its coordinates equal to the value AST__BAD (as defined in the "ast.h" header file), then the corresponding output data (and variance) should either be set to the value given by "badval", or left unchanged, depending on whether the AST__NOBAD flag is specified by "flags".

params

This will be a pointer to the same array as was given via the "params" parameter of astResample<X>. You may use this to pass any additional parameter values required by your interpolation algorithm.

flags

This will be the same value as was given via the "flags" parameter of astResample<X>. You may test this value to provide additional control over the operation of your resampling algorithm. Note that the special flag values AST__URESAMP1, 2, 3 & 4 are reserved for you to use for your own purposes and will not clash with other pre-defined flag values (see astResample<X>).

badval

This will be the same value as was given via the "badval" parameter of astResample<X>, and will have the same numerical type as the data being processed (i.e. as elements of the "in" array). It should be used to test for bad pixels in the input grid (but only if the AST__USEBAD flag is set via the "flags" parameter) and (unless the AST__NOBAD flag is set in "flags") for identifying bad output values in the "out" (and "out_var") array(s).

out

Pointer to an array with the same numerical type as the "in" array, into which the interpolated data values should be returned. Note that details of the storage order and number of dimensions of this array are not required, since the "offset" array contains all necessary information about where each returned value should be stored.

In general, not all elements of this array (or the "out_var" array below) may be used in any particular invocation of the function. Those which are not used should be returned unchanged.

out_var

Pointer to an optional array with the same type and size as the "out" array, into which variance estimates for the resampled values should be returned. This array will only be given if the "in_var" array has also been given.

If given, it is addressed in exactly the same way (via the "offset" array) as the "out" array. The values returned should be estimates of the statistical variance of the corresponding values in the "out" array, on the assumption that all errors in input data values are statistically independent and that their variance estimates may simply be summed (with appropriate weighting factors).

If no output variance estimates are required, a NULL pointer will be given.

nbad

Pointer to an int in which to return the number of interpolation points at which no valid interpolated value could be obtained. The maximum value that should be returned is "npoint", and the minimum is zero (indicating that all output values were successfully obtained).

Notes:

- The data type <Xtype> indicates the numerical type of the data being processed, as for astResample<X>.
- This function will typically be invoked more than once for each invocation of astResample<X>.
- If an error occurs within this function, it should use astSetStatus to set the AST error status to an error value. This will cause an immediate return from astResample<X>. The error value AST__UINER is available for this purpose, but other values may also be used (e.g. if you wish to distinguish different types of error).

astUkern1 1-dimensional sub-pixel interpolation kernel astUkern1

Description: This is a fictitious function which does not actually exist. Instead, this description constitutes a template so that you may implement a function with this interface for yourself (and give it any name you wish). A pointer to such a function may be passed via the "finterp" parameter of the astResample<X> functions (q.v.) in order to supply a 1-dimensional interpolation kernel to the algorithm which performs sub-pixel interpolation during resampling of gridded data (you must also set the "interp" parameter of astResample<X> to the value AST__UKERN1). This allows you to use your own interpolation kernel in addition to those which are pre-defined.

The function calculates the value of a 1-dimensional sub-pixel interpolation kernel. This determines how the weight given to neighbouring pixels in calculating an interpolated value depends on the pixel's offset from the interpolation point. In more than one dimension, the weight assigned to a pixel is formed by evaluating this 1-dimensional kernel using the offset along each dimension in turn. The product of the returned values is then used as the pixel weight.

Synopsis: `void astUkern1( double offset, const double params[], int flags, double *value )`

Parameters:

offset
This will be the offset of the pixel from the interpolation point, measured in pixels. This value may be positive or negative, but for most practical interpolation schemes its sign should be ignored.

params
This will be a pointer to the same array as was given via the "params" parameter of astResample<X>. You may use this to pass any additional parameter values required by your kernel, but note that "params[0]" will already have been used to specify the number of neighbouring pixels which contribute to the interpolated value.

flags
This will be the same value as was given via the "flags" parameter of astResample<X>. You may test this value to provide additional control over the operation of your function. Note that the special flag values AST__URESAMP1, 2, 3 & 4 are reserved for you to use for your own purposes and will not clash with other pre-defined flag values (see astResample<X>).

value
Pointer to a double to receive the calculated kernel value, which may be positive or negative.

Notes:

- Not all functions make good interpolation kernels. In general, acceptable kernels tend to be symmetrical about zero, to have a positive peak (usually unity) at zero, and to evaluate to zero whenever the pixel offset has any other integral value (this ensures that the interpolated values pass through the original data). An interpolation kernel may or may not have regions with negative values. You should consult a good book on image processing for more details.
- If an error occurs within this function, it should use astSetStatus to set the AST error status to an error value. This will cause an immediate return from astResample<X>. The error value AST__UK1ER is available for this purpose, but other values may also be used (e.g. if you wish to distinguish different types of error).

---

astUnformat Read a formatted coordinate value for a Frame axis astUnformat

Description: This function reads a formatted coordinate value (given as a character string) for a Frame axis and returns the equivalent numerical (double) value. It also returns the number of characters read from the string.

The principle use of this function is in decoding user-supplied input which contains formatted coordinate values. Free-format input is supported as far as possible. If input is ambiguous, it is interpreted with reference to the Frame's attributes (in particular, the Format string associated with the Frame's axis). This function is, in essence, the inverse of astFormat.

Synopsis: `int astUnformat( AstFrame *this, int axis, const char *string, double *value )`

Parameters:

this
Pointer to the Frame.

axis
The number of the Frame axis for which a coordinate value is to be read (axis numbering starts at 1 for the first axis).

string
Pointer to a null-terminated character string containing the formatted coordinate value. This string may contain additional information following the value to be read, in which case reading stops at the first character which cannot be interpreted as part of the value. Any white space before or after the value is discarded.

value
Pointer to a double in which the coordinate value read will be returned.

Class Applicability:

Frame
This function applies to all Frames. See the "Frame Input Format" section below for details of the input formats accepted by a basic Frame.

SkyFrame
The SkyFrame class re-defines the input format to be suitable for representing angles and times, with the resulting coordinate value returned in radians. See the "SkyFrame Input Format" section below for details of the formats accepted.

FrameSet
The input formats accepted by a FrameSet are determined by its current Frame (as specified by the Current attribute).

Returned Value:

astUnformat()
The number of characters read from the string in order to obtain the coordinate value. This will include any white space which occurs before or after the value.

Notes:

- A function value of zero (and no coordinate value) will be returned, without error, if the string supplied does not contain a suitably formatted value.

- Beware that it is possible for a formatting error part-way through an input string to terminate input before it has been completely read, but to yield a coordinate value that appears valid. For example, if a user types "1.5r6" instead of "1.5e6", the "r" will terminate input, giving an incorrect coordinate value of 1.5. It is therefore most important to check the return value of this function to ensure that the correct number of characters have been read.
- An error will result if a value is read which appears to have the correct format, but which cannot be converted into a valid coordinate value (for instance, because the value of one or more of its fields is invalid).
- The string "<bad>" is recognised as a special case and will yield the coordinate value AST__BAD without error. The test for this string is case-insensitive and also permits embedded white space.
- A function result of zero will be returned and no coordinate value will be returned via the "value" pointer if this function is invoked with the AST error status set, or if it should fail for any reason.

Frame Input Format:

The input format accepted for a basic Frame axis is as follows:

- An optional sign, followed by:
- A sequence of one or more digits possibly containing a decimal point, followed by:
- An optional exponent field.
- The exponent field, if present, consists of "E" or "e" followed by a possibly signed integer.

Examples of acceptable Frame input formats include:

- 99
- 1.25
- -1.6
- 1E8
- -.99e-17
- <bad>

SkyFrame Input Format:

The input format accepted for a SkyFrame axis is as follows:

- An optional sign, followed by between one and three fields representing either degrees, arc-minutes, arc-seconds or hours, minutes, seconds (e.g. "-12 42 03").
- Each field should consist of a sequence of one or more digits, which may include leading zeros. At most one field may contain a decimal point, in which case it is taken to be the final field (e.g. decimal degrees might be given as "124.707", while degrees and decimal arc-minutes might be given as "-13 33.8").
- The first field given may take any value, allowing angles and times outside the conventional ranges to be represented. However, subsequent fields must have values of less than 60 (e.g. "720 45 31" is valid, whereas "11 45 61" is not).
- Fields may be separated by white space or by ":" (colon), but the choice of separator must be used consistently throughout the value. Additional white space may be present around fields and separators (e.g. "- 2: 04 : 7.1").
- The following field identification characters may be used as separators to replace either of those above (or may be appended to the final field), in order to identify the field to which they are appended: "d"—degrees; "h"—hours; "m"—minutes of arc or time; "s"—seconds of arc or time; "’" (single quote)—minutes of arc; """ (double quote)—seconds of arc. Either lower or upper case may be used. Fields must be given in order of decreasing significance (e.g. "-11D 3’ 14.4"" or "22h14m11.2s").

- The presence of any of the field identification characters `"d"`, `"'"` (single quote) or `"""` (double quote) indicates that the value is to be interpreted as an angle. Conversely, the presence of `"h"` indicates that it is to be interpreted as a time (with 24 hours corresponding to 360 degrees). Incompatible angle/time identification characters may not be mixed (e.g. `"10h14'3""` is not valid). The remaining field identification characters and separators do not specify a preference for an angle or a time and may be used with either.
- If no preference for an angle or a time is expressed anywhere within the value, it is interpreted as an angle if the Format attribute string associated with the SkyFrame axis generates an angle and as a time otherwise. This ensures that values produced by astFormat are correctly interpreted by astUnformat.
- Fields may be omitted, in which case they default to zero. The remaining fields may be identified by using appropriate field identification characters (see above) and/or by adding extra colon separators (e.g. "-05m13s" is equivalent to "-:05:13"). If a field is not identified explicitly, it is assumed that adjacent fields have been given, after taking account of any extra separator characters (e.g. "14:25.4s" specifies minutes and seconds, while "14::25.4s" specifies degrees and seconds).
- If fields are omitted in such a way that the remaining ones cannot be identified uniquely (e.g. "01:02"), then the first field (either given explicitly or implied by an extra leading colon separator) is taken to be the most significant field that astFormat would produce when formatting a value (using the Format attribute associated with the SkyFrame axis). By default, this means that the first field will normally be interpreted as degrees or hours. However, if this does not result in consistent field identification, then the last field (either given explicitly or implied by an extra trailing colon separator) is taken to to be the least significant field that astFormat would produce.

This final convention is intended to ensure that values formatted by astFormat which contain less than three fields will be correctly interpreted if read back using astUnformat, even if they do not contain field identification characters.

Examples of acceptable SkyFrame input formats (with interpretation in parentheses) include:

- -14d 13m 22.2s (-14d 13' 22.2")
- + 12:34:56.7 (12d 34' 56.7" or 12h 34m 56.7s)
- 001 : 02 : 03.4 (1d 02' 03.4" or 1h 02m 03.4s)
- 22h 30 (22h 30m 00s)
- 136::10" (136d 00' 10" or 136h 00m 10s)
- -14M 27S (-0d 14' 27" or -0h 14m 27s)
- -:14: (-0d 14' 00" or -0h 14m 00s)
- -::4.1 (-0d 00' 04.1" or -0h 00m 04.1s)
- .9" (0d 00' 00.9")
- d12m (0d 12' 00")
- H 12:22.3s (0h 12m 22.3s)
- <bad> (AST__BAD)

Where alternative interpretations are shown, the choice of angle or time depends on the associated Format(axis) attribute.

---

astUnitMap Create a UnitMap astUnitMap

Description: This function creates a new UnitMap and optionally initialises its attributes.

A UnitMap is a unit (null) Mapping that has no effect on the coordinates supplied to it. They are simply copied. This can be useful if a Mapping is required (e.g. to pass to another function) but you do not want it to have any effect.

Synopsis: `AstUnitMap *astUnitMap( int ncoord, const char *options, ... )`

Parameters:

ncoord

The number of input and output coordinates (these numbers are necessarily the same).

options

Pointer to a null-terminated string containing an optional comma-separated list of attribute assignments to be used for initialising the new UnitMap. The syntax used is identical to that for the astSet function and may include "printf" format specifiers identified by "%" symbols in the normal way.

...

If the "options" string contains "%" format specifiers, then an optional list of additional arguments may follow it in order to supply values to be substituted for these specifiers. The rules for supplying these are identical to those for the astSet function (and for the C "printf" function).

Returned Value:

astUnitMap()

A pointer to the new UnitMap.

Notes:

- A null Object pointer (AST__NULL) will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

---

astUnlock Unlock an Object for use by other threads astUnlock

Description: Unlocks an Object previously locked using astLock, so that other threads can use the Object. See astLock for further details.

Synopsis: `void astUnlock( AstObject *this, int report )`

Parameters:

this

Pointer to the Object to be unlocked.

report

If non-zero, an error will be reported if the supplied Object, or any Object contained within the supplied Object, is not currently locked by the running thread. If zero, such Objects will be left unchanged, and no error will be reported.

Class Applicability:

Object

This function applies to all Objects.

Notes:

- This function attempts to execute even if the global error status is set, but no further error report will be made if it subsequently fails under these circumstances.
- All unlocked Objects are excluded from AST context handling until they are re-locked using astLock.
- This function is only available in the C interface.
- This function returns without action if the Object is not currently locked by any thread. If it is locked by the running thread, it is unlocked. If it is locked by another thread, an error will be reported if "error" is non-zero.
- This function returns without action if the AST library has been built without POSIX thread support (i.e. the "-with-pthreads" option was not specified when running the "configure" script).

---

astVersion Return the version of the AST library being used astVersion

Description: This macro invokes a function which returns an integer representing the version of the AST library being used. The library version is formatted as a string such as "2.0-7" which contains integers representing the "major version" (2), the "minor version" (0) and the "release" (7). The integer returned by this function combines all three integers together into a single integer using the expresion:

(major version)∗1E6 + (minor version)∗1E3 + (release)

Synopsis: `int astVersion`

Class Applicability:

Object
This macro applies to all Objects.

Returned Value:

astVersion
The major version, minor version and release numbers for the AST library, encoded as a single integer.

---

astWarnings Returns any warnings issued by the previous read or write operation astWarnings

Description: This function returns an AST KeyMap object holding the text of any warnings issued as a result of the previous invocation of the astRead or astWrite function on the Channel. If no warnings were issued, a a NULL value will be returned.

Such warnings are non-fatal and will not prevent the read or write operation succeeding. However, the converted object may not be identical to the original object in all respects. Differences which would usually be deemed as insignificant in most usual cases will generate a warning, whereas more significant differences will generate an error.

The "Strict" attribute allows this warning facility to be switched off, so that a fatal error is always reported for any conversion error.

Synopsis: `AstKeyMap *astWarnings( AstChannel *this )`

Parameters:

this
Pointer to the Channel.

Class Applicability:

Channel

The basic Channel class generates a warning when ever an un-recognised item is encountered whilst reading an Object from an external data source. If Strict is zero (the default), then unexpected items in the Object description are simply ignored, and any remaining items are used to construct the returned Object. If Strict is non-zero, an error will be reported and a NULL Object pointer returned if any unexpected items are encountered.

As AST continues to be developed, new attributes are added occasionally to selected classes. If an older version of AST is used to read external Object descriptions created by a more recent version of AST, then the Channel class will, by default, ignore the new attributes, using the remaining attributes to construct the Object. This is usually a good thing. However, since external Object descriptions are often stored in plain text, it is possible to edit them using a text editor. This gives rise to the possibility of genuine errors in the description due to finger-slips, typos, or simple mis-understanding. Such inappropriate attributes will be ignored if Strict is left at its default zero value. This will cause the mis-spelled attribute to revert to its default value, potentially causing subtle changes in the behaviour of application software. If such an effect is suspected, the Strict attribute can be set non-zero, resulting in the erroneous attribute being identified in an error message.

FitsChan

The returned KeyMap will contain warnings for all conditions listed in the Warnings attribute.

XmlChan

Reports conversion errors that result in what are usally insignificant changes.

Returned Value:

astWarnings()

A pointer to the KeyMap holding the warning messages, or NULL if no warnings were issued during the previous read operation.

Notes:

- The returned KeyMap uses keys of the form "Warning_1", "Warning_2", etc.
- A value of NULL will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

---

astWatch Identify a new error status variable for the AST library astWatch

Description: This function allows a new error status variable to be accessed by the AST library when checking for and reporting error conditions.

By default, the library uses an internal integer error status which is set to an error value if an error occurs. Use of astWatch allows the internal error status to be replaced by an integer variable of your choosing, so that the AST library can share its error status directly with other code which uses the same error detection convention.

If an alternative error status variable is supplied, it is used by all related AST functions and macros (e.g. astOK, astStatus and astClearStatus).

Synopsis: `int *astWatch( int *status_ptr )`

Parameters:

status_ptr

Pointer to an int whose value is to be used subsequently as the AST inherited status value. If a NULL pointer is supplied, the AST library will revert to using its own internal error status.

Returned Value:

astWatch()
Address of the previous error status variable. This may later be passed back to astWatch to restore the previous behaviour of the library. (Note that on the first invocation of astWatch the returned value will be the address of the internal error status variable.)

Notes:

- This function is not available in the FORTRAN 77 interface to the AST library.

---

astWcsMap Create a WcsMap astWcsMap

Description: This function creates a new WcsMap and optionally initialises its attributes.

A WcsMap is used to represent sky coordinate projections as described in the (draft) FITS world coordinate system (FITS-WCS) paper by E.W. Griesen and M. Calabretta (A & A, in preparation). This paper defines a set of functions, or sky projections, which transform longitude-latitude pairs representing spherical celestial coordinates into corresponding pairs of Cartesian coordinates (and vice versa).

A WcsMap is a specialised form of Mapping which implements these sky projections and applies them to a specified pair of coordinates. All the projections in the FITS-WCS paper are supported, plus the now deprecated "TAN with polynomial correction terms" projection which is refered to here by the code "TPN". Using the FITS-WCS terminology, the transformation is between "native spherical" and "projection plane" coordinates. These coordinates may, optionally, be embedded in a space with more than two dimensions, the remaining coordinates being copied unchanged. Note, however, that for consistency with other AST facilities, a WcsMap handles coordinates that represent angles in radians (rather than the degrees used by FITS-WCS).

The type of FITS-WCS projection to be used and the coordinates (axes) to which it applies are specified when a WcsMap is first created. The projection type may subsequently be determined using the WcsType attribute and the coordinates on which it acts may be determined using the WcsAxis(lonlat) attribute.

Each WcsMap also allows up to 100 "projection parameters" to be associated with each axis. These specify the precise form of the projection, and are accessed using PVi_m attribute, where "i" is the integer axis index (starting at 1), and m is an integer "parameter index" in the range 0 to 99. The number of projection parameters required by each projection, and their meanings, are dependent upon the projection type (most projections either do not use any projection parameters, or use parameters 1 and 2 associated with the latitude axis). Before creating a WcsMap you should consult the FITS-WCS paper for details of which projection parameters are required, and which have defaults. When creating the WcsMap, you must explicitly set values for all those required projection parameters which do not have defaults defined in this paper.

Synopsis: `AstWcsMap *astWcsMap( int ncoord, int type, int lonax, int latax, const char *options, ... )`

Parameters:

ncoord
The number of coordinate values for each point to be transformed (i.e. the number of dimensions of the space in which the points will reside). This must be at least 2. The same number is applicable to both input and output points.

type
The type of FITS-WCS projection to apply. This should be given using a macro value such as AST__TAN (for a tangent plane projection), where the characters following the double underscore give the projection type code (in upper case) as used in the FITS-WCS "CTYPEi"

keyword. You should consult the FITS-WCS paper for a list of the available projections. The additional code of AST__TPN can be supplied which represents a TAN projection with polynomial correction terms as defined in an early draft of the FITS-WCS paper.

lonax

The index of the longitude axis. This should lie in the range 1 to "ncoord".

latax

The index of the latitude axis. This should lie in the range 1 to "ncoord" and be distinct from "lonax".

options

Pointer to a null-terminated string containing an optional comma-separated list of attribute assignments to be used for initialising the new WcsMap. The syntax used is identical to that for the astSet function and may include "printf" format specifiers identified by "%" symbols in the normal way.

If the sky projection to be implemented requires projection parameter values to be set, then this should normally be done here via the PVi_m attribute (see the "Examples" section). Setting values for these parameters is mandatory if they do not have default values (as defined in the FITS-WCS paper).

...

If the "options" string contains "%" format specifiers, then an optional list of additional arguments may follow it in order to supply values to be substituted for these specifiers. The rules for supplying these are identical to those for the astSet function (and for the C "printf" function).

Returned Value:

astWcsMap()

A pointer to the new WcsMap.

Examples:

`wcsmap = astWcsMap( 2, AST__MER, 1, 2, "" );`

Creates a WcsMap that implements a FITS-WCS Mercator projection on pairs of coordinates, with coordinates 1 and 2 representing the longitude and latitude respectively. Note that the FITS-WCS Mercator projection does not require any projection parameters.

`wcsmap = astWcsMap( 3, AST__COE, 2, 3, "PV3_1=40.0" );`

Creates a WcsMap that implements a FITS-WCS conical equal area projection. The WcsMap acts on points in a 3-dimensional space; coordinates 2 and 3 represent longitude and latitude respectively, while the values of coordinate 1 are copied unchanged. Projection parameter 1 associatyed with the latitude axis (corresponding to FITS keyword "PV3_1") is required and has no default, so is set explicitly to 40.0 degrees. Projection parameter 2 (corresponding to FITS keyword "PV3_2") is required but has a default of zero, so need not be specified.

Notes:

- The forward transformation of a WcsMap converts between FITS-WCS "native spherical" and "relative physical" coordinates, while the inverse transformation converts in the opposite direction. This arrangement may be reversed, if required, by using astInvert or by setting the Invert attribute to a non-zero value.
- If any set of coordinates cannot be transformed (for example, many projections do not cover the entire celestial sphere), then a WcsMap will yield coordinate values of AST__BAD.
- The validity of any projection parameters given via the PVi_m parameter in the "options" string is not checked by this function. However, their validity is checked when the resulting WcsMap is used to transform coordinates, and an error will result if the projection parameters do not satisfy all the required constraints (as defined in the FITS-WCS paper).

- A null Object pointer (AST__NULL) will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

Status Handling:

The protected interface to this function includes an extra parameter at the end of the parameter list descirbed above. This parameter is a pointer to the integer inherited status variable: "int ∗status".

---

astWinMap Create a WinMap astWinMap

Description: This function creates a new WinMap and optionally initialises its attributes.

A Winmap is a linear Mapping which transforms a rectangular window in one coordinate system into a similar window in another coordinate system by scaling and shifting each axis (the window edges being parallel to the coordinate axes).

A WinMap is specified by giving the coordinates of two opposite corners (A and B) of the window in both the input and output coordinate systems.

Synopsis: `AstWinMap *astWinMap( int ncoord, const double ina[], const double inb[], const double outa[], const double outb[], const char *options, ... )`

Parameters:

ncoord

The number of coordinate values for each point to be transformed (i.e. the number of dimensions of the space in which the points will reside). The same number is applicable to both input and output points.

ina

An array containing the "ncoord" coordinates of corner A of the window in the input coordinate system.

inb

An array containing the "ncoord" coordinates of corner B of the window in the input coordinate system.

outa

An array containing the "ncoord" coordinates of corner A of the window in the output coordinate system.

outb

An array containing the "ncoord" coordinates of corner B of the window in the output coordinate system.

options

Pointer to a null-terminated string containing an optional comma-separated list of attribute assignments to be used for initialising the new WinMap. The syntax used is identical to that for the astSet function and may include "printf" format specifiers identified by "%" symbols in the normal way.

...

If the "options" string contains "%" format specifiers, then an optional list of additional arguments may follow it in order to supply values to be substituted for these specifiers. The rules for supplying these are identical to those for the astSet function (and for the C "printf" function).

Returned Value:

astWinMap()

A pointer to the new WinMap.

Notes:

- A null Object pointer (AST__NULL) will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

Status Handling:

The protected interface to this function includes an extra parameter at the end of the parameter list descirbed above. This parameter is a pointer to the integer inherited status variable: "int ∗status".

---

astWrite Write an Object to a Channel astWrite

Description: This function writes an Object to a Channel, appending it to any previous Objects written to that Channel.

Synopsis: `int astWrite( AstChannel *this, AstObject *object )`

Parameters:

this
Pointer to the Channel.

object
Pointer to the Object which is to be written.

Class Applicability:

FitsChan
If the FitsChan uses a foreign encoding (e.g. FITS-WCS) rather than the native AST encoding, then storing values in the FitsChan for keywords NAXIS1, NAXIS2, etc., before invoking astWrite can help to produce a successful write.

Returned Value:

astWrite()
The number of Objects written to the Channel by this invocation of astWrite (normally, this will be one).

Notes:

- A value of zero will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.
- Invoking this function will usually cause the sink function associated with the channel to be called in order to transfer a textual description of the supplied object to some external data store. However, the FitsChan class behaves differently. Invoking this function on a FitsChan causes new FITS header cards to be added to an internal buffer (the sink function is not invoked). This buffer is written out through the sink function only when the FitsChan is deleted.

---

astWriteFits Write out all cards in a FitsChan to the sink function astWriteFits

Description: This function writes out all cards currently in the FitsChan. If the SinkFile attribute is set, they will be written out to the specified sink file. Otherwise, they will be written out using the sink function specified when the FitsChan was created. All cards are then deleted from the FitsChan.

Synopsis: `void astWriteFits( AstFitsChan *this )`

Parameters:

this

Pointer to the FitsChan.

Notes:

- If the SinkFile is unset, and no sink function is available, this method simply empties the FitsChan, and is then equivalent to astEmptyFits.
- This method attempt to execute even if an error has occurred previously.

---

astXmlChan Create an XmlChan astXmlChan

Description: This function creates a new XmlChan and optionally initialises its attributes.

A XmlChan is a specialised form of Channel which supports XML I/O operations. Writing an Object to an XmlChan (using astWrite) will, if the Object is suitable, generate an XML description of that Object, and reading from an XmlChan will create a new Object from its XML description.

Normally, when you use an XmlChan, you should provide "source" and "sink" functions which connect it to an external data store by reading and writing the resulting XML text. By default, however, an XmlChan will read from standard input and write to standard output.

Alternatively, an XmlChan can be told to read or write from specific text files using the SinkFile and SourceFile attributes, in which case no sink or source function need be supplied.

Synopsis: `AstXmlChan *astXmlChan( const char *(* source)( void ), void (* sink)( const char * ), const char *options, ... )`

Parameters:

source

Pointer to a source function that takes no arguments and returns a pointer to a null-terminated string. If no value has been set for the SourceFile attribute, this function will be used by the XmlChan to obtain lines of input text. On each invocation, it should return a pointer to the next input line read from some external data store, and a NULL pointer when there are no more lines to read.

If "source" is NULL and no value has been set for the SourceFile attribute, the XmlChan will read from standard input instead.

sink

Pointer to a sink function that takes a pointer to a null-terminated string as an argument and returns void. If no value has been set for the SinkFile attribute, this function will be used by the XmlChan to deliver lines of output text. On each invocation, it should deliver the contents of the string supplied to some external data store.

If "sink" is NULL, and no value has been set for the SinkFile attribute, the XmlChan will write to standard output instead.

options

Pointer to a null-terminated string containing an optional comma-separated list of attribute assignments to be used for initialising the new XmlChan. The syntax used is identical to that for the astSet function and may include "printf" format specifiers identified by "%" symbols in the normal way.

...

If the "options" string contains "%" format specifiers, then an optional list of additional arguments may follow it in order to supply values to be substituted for these specifiers. The rules for supplying these are identical to those for the astSet function (and for the C "printf" function).

Returned Value:

astXmlChan()

A pointer to the new XmlChan.

Notes:

- If the external data source or sink uses a character encoding other than ASCII, the supplied source and sink functions should translate between the external character encoding and the internal ASCII encoding used by AST.
- A null Object pointer (AST__NULL) will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

---

astZoomMap Create a ZoomMap astZoomMap

Description: This function creates a new ZoomMap and optionally initialises its attributes.

A ZoomMap is a Mapping which "zooms" a set of points about the origin by multiplying all coordinate values by the same scale factor (the inverse transformation is performed by dividing by this scale factor).

Synopsis: `AstZoomMap *astZoomMap( int ncoord, double zoom, const char *options, ... )`

Parameters:

ncoord

The number of coordinate values for each point to be transformed (i.e. the number of dimensions of the space in which the points will reside). The same number is applicable to both input and output points.

zoom

Initial scale factor by which coordinate values should be multiplied (by the forward transformation) or divided (by the inverse transformation). This factor may subsequently be changed via the ZoomMap's Zoom attribute. It may be positive or negative, but should not be zero.

options

Pointer to a null-terminated string containing an optional comma-separated list of attribute assignments to be used for initialising the new ZoomMap. The syntax used is identical to that for the astSet function and may include "printf" format specifiers identified by "%" symbols in the normal way.

...

If the "options" string contains "%" format specifiers, then an optional list of additional arguments may follow it in order to supply values to be substituted for these specifiers. The rules for supplying these are identical to those for the astSet function (and for the C "printf" function).

Returned Value:

astZoomMap()
A pointer to the new ZoomMap.

Notes:

- A null Object pointer (AST__NULL) will be returned if this function is invoked with the AST error status set, or if it should fail for any reason.

Status Handling:

The protected interface to this function includes an extra parameter at the end of the parameter list descirbed above. This parameter is a pointer to the integer inherited status variable: "int ∗status".

C AST Attribute Descriptions

Abbrev(axis) Abbreviate leading fields within numerical axis labels? Abbrev(axis)

Description: This attribute controls the appearance of an annotated coordinate grid (drawn with the astGrid function) by determining whether matching leading fields should be removed from adjacent numerical axis labels. It takes a separate value for each physical axis of a Plot so that, for instance, the setting "Abbrev(2)=0" specifies that matching leading fields should not be removed on the second axis.

If the Abbrev value of a Plot is non-zero (the default), then leading fields will be removed from adjacent axis labels if they are equal.

Type:

Integer (boolean).

Class Applicability:

Plot

All Plots have this attribute.

Notes:

- If no axis is specified, (e.g. "Abbrev" instead of "Abbrev(2)"), then a "set" or "clear" operation will affect the attribute value of all the Plot axes, while a "get" or "test" operation will use just the Abbrev(1) value.

Adaptive Should the area adapt to changes in the coordinate system? Adaptive

Description: The coordinate system represented by a Region may be changed by assigning new values to attributes such as System, Unit, etc. For instance, a Region representing an area on the sky in ICRS coordinates may have its System attribute changed so that it represents (say) Galactic coordinates instead of ICRS. This attribute controls what happens when the coordinate system represented by a Region is changed in this way.

If Adaptive is non-zero (the default), then area represented by the Region adapts to the new coordinate system. That is, the numerical values which define the area represented by the Region are changed by mapping them from the old coordinate system into the new coordinate system. Thus the Region continues to represent the same physical area.

If Adaptive is zero, then area represented by the Region does not adapt to the new coordinate system. That is, the numerical values which define the area represented by the Region are left unchanged. Thus the physical area represented by the Region will usually change.

As an example, consider a Region describe a range of wavelength from 2000 Angstrom to 4000 Angstrom. If the Unit attribute for the Region is changed from Angstrom to "nm" (nanometre), what happens depends on the setting of Adaptive. If Adaptive is non-zero, the Mapping from the old to the new coordinate system is found. In this case it is a simple scaling by a factor of 0.1 (since 1 Angstrom is 0.1 nm). This Mapping is then used to modify the numerical values within the Region, changing 2000 to 200 and 4000 to 400. Thus the modified region represents 200 nm to 400 nm, the same physical space as the original 2000 Angstrom to 4000 Angstrom. However, if Adaptive had been zero, then the numerical values would not have been changed, resulting in the final Region representing 2000 nm to 4000 nm.

Setting Adaptive to zero can be necessary if you want correct inaccurate attribute settings in an existing Region. For instance, when creating a Region you may not know what Epoch value to use, so you would leave Epoch unset resulting in some default value being used. If at some later point in the application, the correct Epoch value is determined, you could assign the correct value to the Epoch attribute. However, you would first need to set Adaptive temporarily to zero, because otherwise the area represented by the Region would be Mapped from the spurious default Epoch to the new correct Epoch, which is not what is required.

Type:
Integer (boolean).

Class Applicability:

Region
All Regions have this attribute.

AlignOffset Align SkyFrames using the offset coordinate system? AlignOffset

Description: This attribute is a boolean value which controls how a SkyFrame behaves when it is used (by astFindFrame or astConvert) as a template to match another (target) SkyFrame. It determines the coordinate system in which the two SkyFrames are aligned if a match occurs.

If the template and target SkyFrames both have defined offset coordinate systems (i.e. the SkyRefIs attribute is set to either "Origin" or " Pole"), and they both have a non-zero value for AlignOffset, then alignment occurs within the offset coordinate systems (that is, a UnitMap will always be used to align the two SkyFrames). If either the template or target SkyFrame has zero (the default value) for AlignOffset, or if either SkyFrame has SkyRefIs set to "Ignored", then alignment occurring within the coordinate system specified by the AlignSystem attribute.

Type:
Integer (boolean).

Class Applicability:

SkyFrame
All SkyFrames have this attribute.

AlignSideBand Should the SideBand attribute be taken into account when aligning this DSBSpecFrame with another DSBSpecFrame? AlignSideBand

Description: This attribute controls how a DSBSpecFrame behaves when an attempt is made to align it with another DSBSpecFrame using astFindFrame or astConvert. If both DSBSpecFrames have a non-zero value for AlignSideBand, the value of the SideBand attribute in each DSBSpecFrame is used so that alignment occurs between sidebands. That is, if one DSBSpecFrame represents USB and the other represents LSB then astFindFrame and astConvert will recognise that the DSBSpecFrames represent different sidebands and will take this into account when constructing the Mapping that maps positions in one DSBSpecFrame into the other. If AlignSideBand in either DSBSpecFrame is set to zero, then the values of the SideBand attributes are ignored. In the above example, this would result in a frequency in the first DSBSpecFrame being mapped onto the same frequency in the second DSBSpecFrame, even though those frequencies refer to different sidebands. In other words, if either AlignSideBand attribute is zero, then the two DSBSpecFrames aligns like basic SpecFrames. The default value for AlignSideBand is zero.

When astFindFrame or astConvert is used on two DSBSpecFrames (potentially describing different spectral coordinate systems and/or sidebands), it returns a Mapping which can be used to transform a position in one DSBSpecFrame into the corresponding position in the other. The Mapping is made up of the following steps in the indicated order:

- If both DSBSpecFrames have a value of 1 for the AlignSideBand attribute, map values from the target's current sideband (given by its SideBand attribute) to the observed sideband (whether USB or LSB). If the target already represents the observed sideband, this step will leave the values unchanged. If either of the two DSBSpecFrames have a value of zero for its AlignSideBand attribute, then this step is omitted.
- Map the values from the spectral system of the target to the spectral system of the template. This Mapping takes into account all the inherited SpecFrame attributes such as System, StdOfRest, Unit, etc.
- If both DSBSpecFrames have a value of 1 for the AlignSideBand attribute, map values from the result's observed sideband to the result's current sideband (given by its SideBand attribute). If the result already represents the observed sideband, this step will leave the values unchanged. If either of the two DSBSpecFrames have a value of zero for its AlignSideBand attribute, then this step is omitted.

Type:

Integer (boolean).

Class Applicability:

DSBSpecFrame

All DSBSpecFrames have this attribute.

AlignSpecOffset Align SpecFrames using the offset coordinate system? AlignSpecOffset

Description: This attribute is a boolean value which controls how a SpecFrame behaves when it is used (by astFindFrame or astConvert) as a template to match another (target) SpecFrame. It determines whether alignment occurs between the offset values defined by the current value of the SpecOffset attribute, or between the corresponding absolute spectral values.

The default value of zero results in the two SpecFrames being aligned so that a given absolute spectral value in one is mapped to the same absolute value in the other. A non-zero value results in the SpecFrames being aligned so that a given offset value in one is mapped to the same offset value in the other.

Type:

Integer (boolean).

Class Applicability:

SpecFrame

All SpecFrames have this attribute.

AlignStdOfRest Standard of rest to use when aligning SpecFrames AlignStdOfRest

Description: This attribute controls how a SpecFrame behaves when it is used (by astFindFrame or astConvert) as a template to match another (target) SpecFrame. It identifies the standard of rest in which alignment is to occur. See the StdOfRest attribute for a desription of the values which may be assigned to this attribute. The default AlignStdOfRest value is "Helio" (heliographic).

When astFindFrame or astConvert is used on two SpecFrames (potentially describing different spectral coordinate systems), it returns a Mapping which can be used to transform a position in one SpecFrame into the corresponding position in the other. The Mapping is made up of the following steps in the indicated order:

- Map values from the system used by the target (wavelength, apparent radial velocity, etc) to the system specified by the AlignSystem attribute, using the target's rest frequency if necessary.
- Map these values from the target's standard of rest to the standard of rest specified by the AlignStdOfRest attribute, using the Epoch, ObsLat, ObsLon, ObsAlt, RefDec and RefRA attributes of the target to define the two standards of rest.
- Map these values from the standard of rest specified by the AlignStdOfRest attribute, to the template's standard of rest, using the Epoch, ObsLat, ObsLon, ObsAlt, RefDec and RefRA attributes of the template to define the two standards of rest.
- Map these values from the system specified by the AlignSystem attribute, to the system used by the template, using the template's rest frequency if necessary.

Type:

String.

Class Applicability:

SpecFrame

All SpecFrames have this attribute.

AlignSystem Coordinate system in which to align the Frame AlignSystem

Description: This attribute controls how a Frame behaves when it is used (by astFindFrame or astConvert) as a template to match another (target) Frame. It identifies the coordinate system in which the two Frames will be aligned by the match.

The values which may be assigned to this attribute, and its default value, depend on the class of Frame and are described in the "Applicability" section below. In general, the AlignSystem attribute will accept any of the values which may be assigned to the System attribute.

The Mapping returned by AST_FINDFRAME or AST_CONVERT will use the coordinate system specified by the AlignSystem attribute as an intermediate coordinate system. The total returned Mapping will first map positions from the first Frame into this intermediate coordinate system, using the attributes of the first Frame. It will then map these positions from the intermediate coordinate system into the second Frame, using the attributes of the second Frame.

Type:

String.

Class Applicability:

Frame

The AlignSystem attribute for a basic Frame always equals "Cartesian", and may not be altered.

CmpFrame

The AlignSystem attribute for a CmpFrame always equals "Compound", and may not be altered.

FrameSet

The AlignSystem attribute of a FrameSet is the same as that of its current Frame (as specified by the Current attribute).

SkyFrame
The default AlignSystem attribute for a SkyFrame is "ICRS".

SpecFrame
The default AlignSystem attribute for a SpecFrame is "Wave" (wavelength).

TimeFrame
The default AlignSystem attribute for a TimeFrame is "MJD".

AlignTimeScale Time scale to use when aligning TimeFrames AlignTimeScale

Description: This attribute controls how a TimeFrame behaves when it is used (by astFindFrame or astConvert) as a template to match another (target) TimeFrame. It identifies the time scale in which alignment is to occur. See the TimeScale attribute for a desription of the values which may be assigned to this attribute. The default AlignTimeScale value depends on the current value of TimeScale: if TimeScale is UT1, GMST, LMST or LAST, the default for AlignTimeScale is UT1, for all other TimeScales the default is TAI.

When astFindFrame or astConvert is used on two TimeFrames (potentially describing different time coordinate systems), it returns a Mapping which can be used to transform a position in one TimeFrame into the corresponding position in the other. The Mapping is made up of the following steps in the indicated order:

- Map values from the system used by the target (MJD, JD, etc) to the system specified by the AlignSystem attribute.
- Map these values from the target's time scale to the time scale specified by the AlignTimeScale attribute.
- Map these values from the time scale specified by the AlignTimeScale attribute, to the template's time scale.
- Map these values from the system specified by the AlignSystem attribute, to the system used by the template.

Type:
String.

Class Applicability:

TimeFrame
All TimeFrames have this attribute.

AllVariants A list of the variant Mappings associated with the current Frame AllVariants

Description: This attrbute gives a space separated list of the names of all the variant Mappings associated with the current Frame (see attribute "Variant"). If the current Frame has no variant Mappings, then the list will hold a single entry equal to the Domain name of the current Frame.

Type:
String, read-only.

Class Applicability:

FrameSet
All FrameSets have this attribute.

AllWarnings A list of all currently available condition names AllWarnings

Description: This read-only attribute is a space separated list of all the conditions names recognized by the Warnings attribute. The names are listed below.

Type:
String, read-only

Class Applicability:

FitsChan
All FitsChans have this attribute.

Conditions:

The following conditions are currently recognised (all are

case-insensitive):

- "BadCel": This condition arises when reading a FrameSet from a non-Native encoded FitsChan if an unknown celestial co-ordinate system is specified by the CTYPE keywords.
- "BadCTYPE": This condition arises when reading a FrameSet from a non-Native encoded FitsChan if an illegal algorithm code is specified by a CTYPE keyword, and the illegal code can be converted to an equivalent legal code.
- "BadLat": This condition arises when reading a FrameSet from a non-Native encoded FitsChan if the latitude of the reference point has an absolute value greater than 90 degrees. The actual absolute value used is set to exactly 90 degrees in these cases.
- "BadMat": This condition arises if the matrix describing the transformation from pixel offsets to intermediate world coordinates cannot be inverted. This matrix describes the scaling, rotation, shear, etc., applied to the pixel axes, and is specified by keywords such as PCi_j, CDi_j, CROTA, etc. For example, the matrix will not be invertable if any rows or columns consist entirely of zeros. The FITS-WCS Paper I "Representation of World Coordinates in FITS" by Greisen & Calabretta requires that this matrix be invertable. Many operations (such as grid plotting) will not be possible if the matrix cannot be inverted.
- "BadPV": This condition arises when reading a FrameSet from a non-Native encoded FitsChan. It is issued if a PVi_m header is found that refers to a projection parameter that is not used by the projection type specified by CTYPE, or the PV values are otherwise inappropriate for the projection type.
- "BadVal": This condition arises when reading a FrameSet from a non-Native encoded FitsChan if it is not possible to convert the value of a FITS keywords to the expected type. For instance, this can occur if the FITS header contains a string value for a keyword which should have a floating point value, or if the keyword has no value at all (i.e. is a comment card).
- "Distortion": This condition arises when reading a FrameSet from a non-Native encoded FitsChan if any of the CTYPE keywords specify an unsupported distortion code using the "4-3-3" format specified in FITS-WCS paper IV. Such distortion codes are ignored.
- "NoCTYPE": This condition arises if a default CTYPE value is used within astRead, due to no value being present in the supplied FitsChan. This condition is only tested for when using non-Native encodings.
- "NoEquinox": This condition arises if a default equinox value is used within astRead, due to no value being present in the supplied FitsChan. This condition is only tested for when using non-Native encodings.

- "NoRadesys": This condition arises if a default reference frame is used for an equatorial co-ordinate system within astRead, due to no value being present in the supplied FitsChan. This condition is only tested for when using non-Native encodings.
- "NoLonpole": This condition arises if a default value is used for the LONPOLE keyword within astRead, due to no value being present in the supplied FitsChan. This condition is only tested for when using non-Native encodings.
- "NoLatpole": This condition arises if a default value is used for the LATPOLE keyword within astRead, due to no value being present in the supplied FitsChan. This condition is only tested for when using non-Native encodings.
- "NoMjd-obs": This condition arises if a default value is used for the date of observation within astRead, due to no value being present in the supplied FitsChan. This condition is only tested for when using non-Native encodings.
- "Tnx": This condition arises if a FrameSet is read from a FITS header containing an IRAF "TNX" projection which includes terms not supproted by AST. Such terms are ignored and so the resulting FrameSet may be inaccurate.
- "Zpx": This condition arises if a FrameSet is read from a FITS header containing an IRAF "ZPX" projection which includes "lngcor" or "latcor" correction terms. These terms are not supported by AST and are ignored. The resulting FrameSet may therefore be inaccurate.

AsTime(axis) — Format celestial coordinates as times? — AsTime(axis)

Description: This attribute specifies the default style of formatting to be used (e.g. by astFormat) for the celestial coordinate values described by a SkyFrame. It takes a separate boolean value for each SkyFrame axis so that, for instance, the setting "AsTime(2)=0" specifies the default formatting style for celestial latitude values.

If the AsTime attribute for a SkyFrame axis is zero, then coordinates on that axis will be formatted as angles by default (using degrees, minutes and seconds), otherwise they will be formatted as times (using hours, minutes and seconds).

The default value of AsTime is chosen according to the sky coordinate system being represented, as determined by the SkyFrame's System attribute. This ensures, for example, that right ascension values will be formatted as times by default, following normal conventions.

Type:

Integer (boolean).

Class Applicability:

SkyFrame

All SkyFrames have this attribute.

Notes:

- The AsTime attribute operates by changing the default value of the corresponding Format(axis) attribute. This, in turn, may also affect the value of the Unit(axis) attribute.
- Only the default style of formatting is affected by the AsTime value. If an explicit Format(axis) value is set, it will over-ride any effect from the AsTime attribute.

Base FrameSet base Frame index Base

Description: This attribute gives the index of the Frame which is to be regarded as the "base" Frame within a FrameSet. The default is the first Frame added to the FrameSet when it is created (this Frame always has an index of 1).

Type:
Integer.

Class Applicability:

FrameSet
All FrameSets have this attribute.

Notes:

- Inverting a FrameSet (inverting the boolean sense of its Invert attribute, with the astInvert function for example) will interchange the values of its Base and Current attributes.

Border Draw a border around valid regions of a Plot? Border

Description: This attribute controls the appearance of an annotated coordinate grid (drawn with the astGrid function) by determining whether a border is drawn around regions corresponding to the valid physical coordinates of a Plot (c.f. astBorder).

If the Border value of a Plot is non-zero, then this border will be drawn as part of the grid. Otherwise, the border is not drawn (although axis labels and tick marks will still appear, unless other relevant Plot attributes indicate that they should not). The default behaviour is to draw the border if tick marks and numerical labels will be drawn around the edges of the plotting area (see the Labelling attribute), but to omit it otherwise.

Type:
Integer (boolean).

Class Applicability:

Plot
All Plots have this attribute.

Bottom(axis) Lowest axis value to display Bottom(axis)

Description: This attribute gives the lowest axis value to be displayed (for instance, by the astGrid method).

Type:
Floating point.

Class Applicability:

Frame
The default supplied by the Frame class is to display all axis values, without any limit.

SkyFrame
The SkyFrame class re-defines the default Bottom value to -90 degrees for latitude axes, and 0 degrees for co-latitude axes. The default for longitude axes is to display all axis values.

Notes:

- When specifying this attribute by name, it should be subscripted with the number of the Frame axis to which it applies.

Bounded Is the Region bounded? **Bounded**

Description: This is a read-only attribute indicating if the Region is bounded. A Region is bounded if it is contained entirely within some finite-size bounding box.

Type:
Integer (boolean), read-only.

Class Applicability:

Region
All Regions have this attribute.

CDMatrix Use CDi_j keywords to represent pixel scaling, rotation, etc? **CDMatrix**

Description: This attribute is a boolean value which specifies how the linear transformation from pixel coordinates to intermediate world coordinates should be represented within a FitsChan when using FITS-WCS encoding. This transformation describes the scaling, rotation, shear, etc., of the pixel axes.

If the attribute has a non-zero value then the transformation is represented by a set of CDi_j keywords representing a square matrix (where `"i"` is the index of an intermediate world coordinate axis and `"j"` is the index of a pixel axis). If the attribute has a zero value the transformation is represented by a set of PCi_j keywords (which also represent a square matrix) together with a corresponding set of CDELTi keywords representing the axis scalings. See FITS-WCS paper II `"`Representation of Celestial Coordinates in FITS`"` by M. Calabretta & E.W. Greisen, for a complete description of these two schemes.

The default value of the CDMatrix attribute is determined by the contents of the FitsChan at the time the attribute is accessed. If the FitsChan contains any CDi_j keywords then the default value is non-zero. Otherwise it is zero. Note, reading a FrameSet from a FitsChan will in general consume any CDi_j keywords present in the FitsChan. Thus the default value for CDMatrix following a read will usually be zero, even if the FitsChan originally contained some CDi_j keywords. This behaviour is similar to that of the Encoding attribute, the default value for which is determined by the contents of the FitsChan at the time the attribute is accessed. If you wish to retain the original value of the CDMatrix attribute (that is, the value before reading the FrameSet) then you should enquire the default value before doing the read, and then set that value explicitly.

Type:
Integer (boolean).

Class Applicability:

FitsChan
All FitsChans have this attribute.

CarLin Ignore spherical rotations on CAR projections? **CarLin**

Description: This attribute is a boolean value which specifies how FITS `"CAR"` (plate carree, or `"Cartesian"`) projections should be treated when reading a FrameSet from a foreign encoded FITS header. If zero (the default), it is assumed that the CAR projection conforms to the conventions described in the FITS world coordinate system (FITS-WCS) paper II "Representation of Celestial Coordinates in FITS" by M. Calabretta & E.W. Greisen. If CarLin is non-zero, then these conventions are ignored, and it is assumed that the mapping from pixel coordinates to celestial coordinates is a simple linear transformation (hence the attribute name "CarLin"). This is appropriate for some older FITS data which claims to have a "CAR" projection, but which in fact do not conform to the conventions of the FITS-WCS paper. Furthermore, if CarLin is non-zero, it is assumed that CDELT and CD keywords are in units of degrees rather than radians (as required by the FITS-WCS papers).

The FITS-WCS paper specifies that headers which include a CAR projection represent a linear mapping from pixel coordinates to "native spherical coordinates", NOT celestial coordinates. An extra mapping is then required from native spherical to celestial. This mapping is a 3D rotation and so the overall Mapping from pixel to celestial coordinates is NOT linear. See the FITS-WCS papers for further details.

Type:

Integer (boolean).

Class Applicability:

FitsChan

All FitsChans have this attribute.

Card Index of current FITS card in a FitsChan Card

Description: This attribute gives the index of the "current" FITS header card within a FitsChan, the first card having an index of 1. The choice of current card affects the behaviour of functions that access the contents of the FitsChan, such as astDelFits, astFindFits and astPutFits.

A value assigned to Card will position the FitsChan at any desired point, so that a particular card within it can be accessed. Alternatively, the value of Card may be enquired in order to determine the current position of a FitsChan.

The default value of Card is 1. This means that clearing this attribute (using astClear) effectively "rewinds" the FitsChan, so that the first card is accessed next. If Card is set to a value which exceeds the total number of cards in the FitsChan (as given by its Ncard attribute), it is regarded as pointing at the "end-of-file". In this case, the value returned in response to an enquiry is always one more than the number of cards in the FitsChan.

Type:

Integer.

Class Applicability:

FitsChan

All FitsChans have this attribute.

CardType The data type of the current card in a FitsChan CardType

Description: This attribute gives the data type of the keyword value for the current card of the FitsChan. It will be one of the following integer constants: AST__NOTYPE, AST__COMMENT, AST__INT, AST__FLOAT, AST__STRING, AST__COMPLEXF, AST__COMPLEXI, AST__LOGICAL, AST__CONTINUE, AST__UNDEF.

Type:

Integer, read-only.

Class Applicability:

FitsChan

All FitsChans have this attribute.

Class Object class name Class

Description: This attribute gives the name of the class to which an Object belongs.

Type:

Character string, read-only.

Class Applicability:

Object

All Objects have this attribute.

Clean Remove cards used whilst reading even if an error occurs? Clean

Description: This attribute indicates whether or not cards should be removed from the FitsChan if an error occurs within astRead. A succesful read on a FitsChan always results in the removal of the cards which were involved in the description of the returned Object. However, in the event of an error during the read (for instance if the cards in the FitsChan have illegal values, or if some required cards are missing) no cards will be removed from the FitsChan if the Clean attribute is zero (the default). If Clean is non-zero then any cards which were used in the aborted attempt to read an object will be removed.

This provides a means of "cleaning" a FitsChan of WCS related cards which works even in the event of the cards not forming a legal WCS description.

Type:

Integer (boolean).

Class Applicability:

FitsChan

All FitsChans have this attribute.

Clip Clip lines and/or markers at the Plot boundary? Clip

Description: This attribute controls whether curves and markers are clipped at the boundary of the graphics box specified when the Plot was created. A value of 3 implies both markers and curves are clipped at the Plot boundary. A value of 2 implies markers are clipped, but not curves. A value of 1 implies curves are clipped, but not markers. A value of zero implies neither curves nor markers are clipped. The default value is 1. Note, this attributes controls only the clipping performed internally within AST. The underlying graphics system may also apply clipping. In such cases, removing clipping using this attribute does not guarantee that no clipping will be visible in the final plot.

The astClip function can be used to establish generalised clipping within arbitrary regions of the Plot.

Type:

Integer.

Class Applicability:

Plot

All Plots have this attribute.

ClipOp Combine Plot clipping limits using a boolean OR? **ClipOp**

Description: This attribute controls how the clipping limits specified for each axis of a Plot (using the astClip function) are combined. This, in turn, determines which parts of the graphical output will be visible.

If the ClipOp attribute of a Plot is zero (the default), graphical output is visible only if it satisfies the clipping limits on all the axes of the clipping Frame (a boolean AND). Otherwise, if ClipOp is non-zero, output is visible if it satisfies the clipping limits on one or more axes (a boolean OR).

An important use of this attribute is to allow areas of a Plot to be left clear (e.g. as a background for some text). To achieve this, the lower and upper clipping bounds supplied to astClip should be reversed, and the ClipOp attribute of the Plot should be set to a non-zero value.

Type:

Integer (boolean).

Class Applicability:

Plot

All Plots have this attribute.

Closed Should the boundary be considered to be inside the region? **Closed**

Description: This attribute controls whether points on the boundary of a Region are considered to be inside or outside the region. If the attribute value is non-zero (the default), points on the boundary are considered to be inside the region (that is, the Region is "closed"). However, if the attribute value is zero, points on the bounary are considered to be outside the region.

Type:

Integer (boolean).

Class Applicability:

Region

All Regions have this attribute.

PointList

The value of the Closed attribute is ignored by PointList regions. If the PointList region has not been negated, then it is always assumed to be closed. If the PointList region has been negated, then it is always assumed to be open. This is required since points have zero volume and therefore consist entirely of boundary.

CmpRegion

The default Closed value for a CmpRegion is the Closed value of its first component Region.

Stc

The default Closed value for an Stc is the Closed value of its encapsulated Region.

Colour(element) Colour index for a Plot element **Colour(element)**

Description: This attribute determines the colour index used when drawing each element of graphical output produced by a Plot. It takes a separate value for each graphical element so that, for instance, the setting "Colour(title)=2" causes the Plot title to be drawn using colour index 2. The synonym "Color" may also be used.

The range of integer colour indices available and their appearance is determined by the underlying graphics system. The default behaviour is for all graphical elements to be drawn using the default colour index supplied by this graphics system (normally, this is likely to result in white plotting on a black background, or vice versa).

Type:
Integer.

Class Applicability:

Plot
All Plots have this attribute.

Notes:

- For a list of the graphical elements available, see the description of the Plot class.
- If no graphical element is specified, (e.g. "Colour" instead of "Colour(title)"), then a "set" or "clear" operation will affect the attribute value of all graphical elements, while a "get" or "test" operation will use just the Colour(TextLab) value.

ColumnLenC(column) The largest string length of any value in a column ColumnLenC(column)

Description: This attribute holds the minimum length which a character variable must have in order to be able to store the longest value currently present (at any row) in a specified column of the supplied Table. This does not include room for a trailing null character. The required column name should be placed inside the parentheses in the attribute name. If the named column holds vector values, then the attribute value is the length of the longest element of the vector value.

Type:
Integer, read-only.

Class Applicability:

Table
All Tables have this attribute.

Notes:

- If the named column holds numerical values, the length returned is the length of the largest string that would be generated if the column values were accessed as strings.

ColumnLength(column) The number of elements in each value in a column ColumnLength(column)

Description: This attribute holds the number of elements in each value stored in a named column. Each value can be a scalar (in which case the ColumnLength attribute has a value of 1), or a multi-dimensional array (in which case the ColumnLength value is equal to the product of the array dimensions).

Type:
Integer, read-only.

Class Applicability:

Table
All Tables have this attribute.

ColumnNdim(column) The number of axes spanned by each value in a column ColumnNdim(column)

Description: This attribute holds the number of axes spanned by each value in a column. If each cell in the column is a scalar, ColumnNdim will be zero. If each cell in the column is a 1D spectrum, ColumnNdim will be one. If each cell in the column is a 2D image, ColumnNdim will be two, etc. The required column name should be placed inside the parentheses in the attribute name.

Type:
Integer, read-only.

Class Applicability:

Table
All Tables have this attribute.

ColumnType(column) The data type of each value in a column ColumnType(column)

Description: This attribute holds a integer value indicating the data type of a named column in a Table. This is the data type which was used when the column was added to the Table using astAddColumn. The required column name should be placed inside the parentheses in the attribute name.

The attribute value will be one of AST__INTTYPE (for integer), AST__SINTTYPE (for short int), AST__BYTETYPE (for unsigned bytes - i.e. unsigned chars), AST__DOUBLETYPE (for double precision floating point), AST__FLOATTYPE (for single precision floating point), AST__STRINGTYPE (for character string), AST__OBJECTTYPE (for AST Object pointer), AST__POINTERTYPE (for arbitrary C pointer) or AST__UNDEFTYPE (for undefined values created by astMapPutU).

Type:
Integer, read-only.

Class Applicability:

Table
All Tables have this attribute.

Comment Include textual comments in output? Comment

Description: This is a boolean attribute which controls whether textual comments are to be included in the output generated by a Channel. If included, they will describe what each item of output represents.

If Comment is non-zero, then comments will be included. If it is zero, comments will be omitted.

Type:
Integer (boolean).

Class Applicability:

Channel
The default value is non-zero for a normal Channel.

FitsChan
The default value is non-zero for a FitsChan.

XmlChan
The default value is zero for an XmlChan.

Current FrameSet current Frame index Current

Description: This attribute gives the index of the Frame which is to be regarded as the "current" Frame within a FrameSet. The default is the most recent Frame added to the FrameSet (this Frame always has an index equal to the FrameSet's Nframe attribute).

Type:
Integer.

Class Applicability:

FrameSet
All FrameSets have this attribute.

Notes:

- Inverting a FrameSet (inverting the boolean sense of its Invert attribute, with the astInvert function for example) will interchange the values of its Base and Current attributes.

DSBCentre The central position of interest in a dual sideband spectrum DSBCentre

Description: This attribute specifies the central position of interest in a dual sideband spectrum. Its sole use is to determine the local oscillator frequency (the frequency which marks the boundary between the lower and upper sidebands). See the description of the IF (intermediate frequency) attribute for details of how the local oscillator frequency is calculated. The sideband containing this central position is referred to as the "observed" sideband, and the other sideband as the "image" sideband.

The value is accessed as a position in the spectral system represented by the SpecFrame attributes inherited by this class, but is stored internally as topocentric frequency. Thus, if the System attribute of the DSBSpecFrame is set to "VRAD", the Unit attribute set to "m/s" and the StdOfRest attribute set to "LSRK", then values for the DSBCentre attribute should be supplied as radio velocity in units of "m/s" relative to the kinematic LSR (alternative units may be used by appending a suitable units string to the end of the value). This value is then converted to topocentric frequency and stored. If (say) the Unit attribute is subsequently changed to "km/s" before retrieving the current value of the DSBCentre attribute, the stored topocentric frequency will be converted back to LSRK radio velocity, this time in units of "km/s", before being returned.

The default value for this attribute is 30 GHz.

Type:
Floating point.

Class Applicability:

DSBSpecFrame
All DSBSpecFrames have this attribute.

Note:

- The attributes which define the transformation to or from topocentric frequency should be assigned their correct values before accessing this attribute. These potentially include System, Unit, StdOfRest, ObsLon, ObsLat, ObsAlt, Epoch, RefRA, RefDec and RestFreq.

DefB1950 Use FK4 B1950 as defaults? DefB1950

Description: This attribute is a boolean value which specifies a default equinox and reference frame to use when reading a FrameSet from a FitsChan with a foreign (i.e. non-native) encoding. It is only used if the FITS header contains RA and DEC axes but contains no information about the reference frame or equinox. If this is the case, then values of FK4 and B1950 are assumed if the DefB1950 attribute has a non-zero value and ICRS is assumed if DefB1950 is zero. The default value for DefB1950 depends on the value of the Encoding attribute: for FITS-WCS encoding the default is zero, and for all other encodings it is one.

Type:
Integer (boolean).

Class Applicability:

FitsChan
All FitsChans have this attribute.

Digits/Digits(axis) Number of digits of precision Digits/Digits(axis)

Description: This attribute specifies how many digits of precision are required by default when a coordinate value is formatted for a Frame axis (e.g. using astFormat). Its value may be set either for a Frame as a whole, or (by subscripting the attribute name with the number of an axis) for each axis individually. Any value set for an individual axis will over-ride the value for the Frame as a whole.

Note that the Digits value acts only as a means of determining a default Format string. Its effects are over-ridden if a Format string is set explicitly for an axis. However, if the Format attribute specifies the precision using the string ".*", then the Digits attribute is used to determine the number of decimal places to produce.

Type:
Integer.

Class Applicability:

Frame
The default Digits value supplied by the Frame class is 7. If a value less than 1 is supplied, then 1 is used instead.

FrameSet
The Digits attribute of a FrameSet (or one of its axes) is the same as that of its current Frame (as specified by the Current attribute).

Plot
The default Digits value used by the Plot class when drawing annotated axis labels is the smallest value which results in all adjacent labels being distinct.

TimeFrame
The Digits attribute is ignored when a TimeFrame formats a value as a date and time string (see the Format attribute).

Direction(axis) Display axis in conventional direction? Direction(axis)

Description: This attribute is a boolean value which suggests how the axes of a Frame should be displayed (e.g.) in graphical output. By default, it has the value one, indicating that they should be shown in the conventional sense (increasing left to right for an abscissa, and bottom to top for an ordinate). If set to zero, this attribute indicates that the direction should be reversed, as would often be done for an astronomical magnitude or a right ascension axis.

Type:
Integer (boolean).

Class Applicability:

Frame
The default Direction value supplied by the Frame class is 1, indicating that all axes should be displayed in the conventional direction.

SkyFrame
The SkyFrame class re-defines the default Direction value to suggest that certain axes (e.g. right ascension) should be plotted in reverse when appropriate.

FrameSet
The Direction attribute of a FrameSet axis is the same as that of its current Frame (as specified by the Current attribute).

Plot
The Direction attribute of the base Frame in a Plot is set to indicate the sense of the two graphics axes, as implied by the graphics bounding box supplied when the Plot was created.

Notes:

- When specifying this attribute by name, it should be subscripted with the number of the Frame axis to which it applies.
- The Direction attribute does not directly affect the behaviour of the AST library. Instead, it serves as a hint to applications programs about the orientation in which they may wish to display any data associated with the Frame. Applications are free to ignore this hint if they wish.

Disco PcdMap pincushion/barrel distortion coefficient Disco

Description: This attribute specifies the pincushion/barrel distortion coefficient used by a PcdMap. This coefficient is set when the PcdMap is created, but may later be modified. If the attribute is cleared, its default value is zero, which gives no distortion. For pincushion distortion, the value should be positive. For barrel distortion, it should be negative.

Note that the forward transformation of a PcdMap applies the distortion specified by this attribute and the inverse transformation removes this distortion. If the PcdMap is inverted (e.g. using astInvert), then the forward transformation will remove the distortion and the inverse transformation will apply it. The distortion itself will still be given by the same value of Disco.

Type:
Double precision.

Class Applicability:

PcdMap
All PcdMaps have this attribute.

Domain Coordinate system domain Domain

Description: This attribute contains a string which identifies the physical domain of the coordinate system that a Frame describes.

The Domain attribute also controls how a Frame behaves when it is used (by astFindFrame) as a template to match another (target) Frame. It does this by specifying the Domain that the target Frame should have in order to match the template. If the Domain value in the template Frame is set, then only targets with the same Domain value will be matched. If the template's Domain value is not set, however, then the target's Domain will be ignored.

Type:

String.

Class Applicability:

Frame

The default Domain value supplied by the Frame class is an empty string.

SkyFrame

The SkyFrame class re-defines the default Domain value to be "SKY".

CmpFrame

The CmpFrame class re-defines the default Domain value to be of the form "<dom1>-<dom2>", where <dom1> and <dom2> are the Domains of the two component Frames. If both these Domains are blank, then the string "CMP" is used as the default Domain name.

FrameSet

The Domain attribute of a FrameSet is the same as that of its current Frame (as specified by the Current attribute).

SpecFrame

The SpecFrame class re-defines the default Domain value to be "SPECTRUM".

DSBSpecFrame

The DSBSpecFrame class re-defines the default Domain value to be "DSBSPECTRUM".

FluxFrame

The FluxFrame class re-defines the default Domain value to be "FLUX".

SpecFluxFrame

The FluxFrame class re-defines the default Domain value to be "SPECTRUM-FLUX".

TimeFrame

The TimeFrame class re-defines the default Domain value to be "TIME".

Notes:

- All Domain values are converted to upper case and white space is removed before use.

DrawAxes(axis) Draw axes for a Plot? DrawAxes(axis)

Description: This attribute controls the appearance of an annotated coordinate grid (drawn with the astGrid function) by determining whether curves representing coordinate axes should be drawn. It takes a separate value for each physical axis of a Plot so that, for instance, the setting "DrawAxes(2)=0" specifies that no axis should be drawn for the second axis.

If drawn, these axis lines will pass through any tick marks associated with numerical labels drawn to mark values on the axes. The location of these tick marks and labels (and hence the axis lines) is determined by the Plot's LabelAt(axis) attribute.

If the DrawAxes value of a Plot is non-zero (the default), then axis lines will be drawn, otherwise they will be omitted.

Type:

Integer (boolean).

Class Applicability:

Plot

All Plots have this attribute.

Notes:

- Axis lines are drawn independently of any coordinate grid lines (see the Grid attribute) so grid lines may be used to substitute for axis lines if required.
- In some circumstances, numerical labels and tick marks are drawn around the edges of the plotting area (see the Labelling attribute). In this case, the value of the DrawAxes attribute is ignored.
- If no axis is specified, (e.g. `"DrawAxes"` instead of `"DrawAxes(2)"`), then a `"set"` or `"clear"` operation will affect the attribute value of all the Plot axes, while a `"get"` or `"test"` operation will use just the DrawAxes(1) value.

DrawTitle Draw a title for a Plot? DrawTitle

Description: This attribute controls the appearance of an annotated coordinate grid (drawn with the astGrid function) by determining whether a title is drawn.

If the DrawTitle value of a Plot is non-zero (the default), then the title will be drawn, otherwise it will be omitted.

Type:

Integer (boolean).

Class Applicability:

Plot

All Plots have this attribute.

Plot3D

The Plot3D class ignores this attributes, assuming a value of zero.

Notes:

- The text used for the title is obtained from the Plot's Title attribute.
- The vertical placement of the title can be controlled using the TitleGap attribute.

Dut1 The UT1-UTC correction Dut1

Description: This attribute is used when calculating the Local Apparent Sidereal Time corresponding to SkyFrame's Epoch value (used when converting positions to or from the `"AzEl"` system). It should be set to the difference, in seconds, between the UT1 and UTC timescales at the moment in time represented by the SkyFrame's Epoch attribute. The value to use is unpredictable and depends on changes in the earth's rotation speed. Values for UT1-UTC can be obtained from the International Earth Rotation and Reference Systems Service (IERS) at http://www.iers.org/.

Currently, the correction is always less than 1 second. This is ensured by the occasional introduction of leap seconds into the UTC timescale. Therefore no great error will usually result if no value is assigned to this attribute (in which case a default value of zero is used). However, it is possible that a decision may be taken at some time in the future to abandon the introduction of leap seconds, in which case the DUT correction could grow to significant sizes.

Type:
Floating point.

Class Applicability:

Frame
All Frames have this attribute.

Edge(axis) Which edges to label in a Plot Edge(axis)

Description: This attribute controls the appearance of an annotated coordinate grid (drawn with the astGrid function) by determining which edges of a Plot are used for displaying numerical and descriptive axis labels. It takes a separate value for each physical axis of the Plot so that, for instance, the setting "Edge(2)=left" specifies which edge to use to display labels for the second axis.

The values "left", "top", "right" and "bottom" (or any abbreviation) can be supplied for this attribute. The default is usually "bottom" for the first axis and "left" for the second axis. However, if exterior labelling was requested (see the Labelling attribute) but cannot be produced using these default Edge values, then the default values will be swapped if this enables exterior labelling to be produced.

Type:
String.

Class Applicability:

Plot
All Plots have this attribute.

Plot3D
The Plot3D class ignores this attributes. Instead it uses its own RootCorner attribute to determine which edges of the 3D plot to label.

Notes:

- In some circumstances, numerical labels will be drawn along internal grid lines instead of at the edges of the plotting area (see the Labelling attribute). In this case, the Edge attribute only affects the placement of the descriptive labels (these are drawn at the edges of the plotting area, rather than along the axis lines).

Encoding System for encoding Objects as FITS headers Encoding

Description: This attribute specifies the encoding system to use when AST Objects are stored as FITS header cards in a FitsChan. It affects the behaviour of the astWrite and astRead functions when they are used to transfer any AST Object to or from an external representation consisting of FITS header cards (i.e. whenever a write or read operation is performed using a FitsChan as the I/O Channel).

There are several ways (conventions) by which coordinate system information may be represented in the form of FITS headers and the Encoding attribute is used to specify which of these should be used. The encoding options available are outlined in the "Encodings Available" section below, and in more detail in the sections which follow.

Encoding systems differ in the range of possible Objects (e.g. classes) they can represent, in the restrictions they place on these Objects (e.g. compatibility with some externally-defined coordinate system model) and in the number of Objects that can be stored together in any particular set of FITS header cards (e.g. multiple Objects, or only a single Object). The choice of encoding also affects the range of external applications which can potentially read and interpret the FITS header cards produced.

The encoding options available are not necessarily mutually exclusive, and it may sometimes be possible to store multiple Objects (or the same Object several times) using different encodings within the same set of FITS header cards. This possibility increases the likelihood of other applications being able to read and interpret the information.

By default, a FitsChan will attempt to determine which encoding system is already in use, and will set the default Encoding value accordingly (so that subsequent I/O operations adopt the same conventions). It does this by looking for certain critical FITS keywords which only occur in particular encodings. For details of how this works, see the "Choice of Default Encoding" section below. If you wish to ensure that a particular encoding system is used, independently of any FITS cards already present, you should set an explicit Encoding value yourself.

Type:

String.

Class Applicability:

FitsChan

All FitsChans have this attribute.

Encodings Available:

The Encoding attribute can take any of the following (case insensitive) string values to select the corresponding encoding

system:

- "DSS": Encodes coordinate system information in FITS header cards using the convention developed at the Space Telescope Science Institute (STScI) for the Digitised Sky Survey (DSS) astrometric plate calibrations. The main advantages of this encoding are that FITS images which use it are widely available and it is understood by a number of important and well-established astronomy applications. For further details, see the section "The DSS Encoding" below.
- "FITS-WCS": Encodes coordinate system information in FITS header cards using the conventions described in the FITS world coordinate system (FITS-WCS) papers by E.W. Greisen, M. Calabretta, et al. The main advantages of this encoding are that it should be understood by any FITS-WCS compliant application and is likely to be adopted widely for FITS data in future. For further details, see the section "The FITS-WCS Encoding" below.
- "FITS-PC": Encodes coordinate system information in FITS header cards using the conventions described in an earlier draft of the FITS world coordinate system papers by E.W. Greisen and M. Calabretta. This encoding uses a combination of CDELTi and PCiiijjj keywords to describe the scale and rotation of the pixel axes. This encoding is included to support existing data and software which uses these now superceded conventions. In general, the "FITS-WCS" encoding (which uses CDi_j or PCi_j keywords to describe the scale and rotation) should be used in preference to "FITS-PC".
- "FITS-IRAF": Encodes coordinate system information in FITS header cards using the conventions described in the document "World Coordinate Systems Representations Within the FITS Format" by R.J. Hanisch and D.G. Wells, 1988. This encoding is currently employed by the IRAF data analysis facility, so its use will facilitate data exchange with IRAF. Its main advantages are that it is a stable convention which approximates to a subset of the

propsed FITS-WCS encoding (above). This makes it suitable as an interim method for storing coordinate system information in FITS headers until the FITS-WCS encoding becomes stable. Since many datasets currently use the FITS-IRAF encoding, conversion of data from FITS-IRAF to the final form of FITS-WCS is likely to be well supported.

- "FITS-AIPS": Encodes coordinate system information in FITS header cards using the conventions originally introduced by the AIPS data analysis facility. This is base on the use of CDELTi and CROTAi keuwords to desribe the scale and rotation of each axis. These conventions have been superceded but are still widely used.
- "FITS-AIPS++": Encodes coordinate system information in FITS header cards using the conventions used by the AIPS++ project. This is an extension of FITS-AIPS which includes some of the features of FITS-IRAF and FITS-PC.
- "FITS-CLASS": Encodes coordinate system information in FITS header cards using the conventions used by the CLASS project. CLASS is a software package for reducing single-dish radio and sub-mm spectroscopic data. See the section "CLASS FITS format" at http://www.iram.fr/IRAMFR/GIL html/.
- "NATIVE": Encodes AST Objects in FITS header cards using a convention which is private to the AST library (but adheres to the general FITS standard) and which uses FITS keywords that will not clash with other encoding systems. The main advantages of this are that any class of AST Object may be encoded, and any (reasonable) number of Objects may be stored sequentially in the same FITS header. This makes FITS headers an almost lossless communication path for passing AST Objects between applications (although all such applications must, of course, make use of the AST library to interpret the information). For further details, see the section "The NATIVE Encoding" below.

Choice of Default Encoding:

If the Encoding attribute of a FitsChan is not set, the default value it takes is determined by the presence of certain critical FITS keywords within the FitsChan. The sequence of decisions used to arrive at the default value is as follows:

- If the FitsChan contains any keywords beginning with the string "BEGAST", then NATIVE encoding is used,
- Otherwise, FITS-CLASS is used if the FitsChan contains a DELTAV keyword and a keyword of the form VELO-xxx, where xxx indicates one of the rest frames used by class (e.g. "VELO-LSR"), or "VLSR".
- Otherwise, if the FitsChan contains a CTYPE keyword which represents a spectral axis using the conventions of the AIPS and AIPS++ projects (e.g. "FELO-LSR", etc), then one of FITS-AIPS or FITS-AIPS++ encoding is used. FITS-AIPS++ is used if any of the keywords CDi_j, PROJP, LONPOLE or LATPOLE are found in the FitsChan. Otherwise FITS-AIPS is used.
- Otherwise, if the FitsChan contains a keyword of the form "PCiiijjj", where "i" and "j" are single digits, then FITS-PC encoding is used,
- Otherwise, if the FitsChan contains a keyword of the form "CDiiijjj", where "i" and "j" are single digits, then FITS-IRAF encoding is used,
- Otherwise, if the FitsChan contains a keyword of the form "CDi_j", and at least one of RADECSYS, PROJPi, or CjVALi where "i" and "j" are single digits, then FITS-IRAF encoding is used.
- Otherwise, if the FitsChan contains any keywords of the form PROJPi, CjVALi or RADECSYS, where "i" and "j" are single digits, then FITS-PC encoding is used.
- Otherwise, if the FitsChan contains a keyword of the form CROTAi, where "i" is a single digit, then FITS-AIPS encoding is used.

- Otherwise, if the FitsChan contains a keyword of the form CRVALi, where "i" is a single digit, then FITS-WCS encoding is used.
- Otherwise, if the FitsChan contains the "PLTRAH" keyword, then DSS encoding is used,
- Otherwise, if none of these conditions is met (as would be the case when using an empty FitsChan), then NATIVE encoding is used.

Except for the NATIVE and DSS encodings, all the above checks also require that the header contains at least one CTYPE, CRPIX and CRVAL keyword (otherwise the checking process continues to the next case).

Setting an explicit value for the Encoding attribute always over-rides this default behaviour.

Note that when writing information to a FitsChan, the choice of encoding will depend greatly on the type of application you expect to be reading the information in future. If you do not know this, there may sometimes be an advantage in writing the information several times, using a different encoding on each occasion.

The DSS Encoding:

The DSS encoding uses FITS header cards to store a multi-term polynomial which relates pixel positions on a digitised photographic plate to celestial coordinates (right ascension and declination). This encoding may only be used to store a single AST Object in any set of FITS header cards, and that Object must be a FrameSet which conforms to the STScI/DSS coordinate system model (this means the Mapping which relates its base and current Frames must include either a DssMap or a WcsMap with type AST__TAN or AST__TPN).

When reading a DSS encoded Object (using astRead), the FitsChan concerned must initially be positioned at the first card (its Card attribute must equal 1) and the result of the read, if successful, will always be a pointer to a FrameSet. The base Frame of this FrameSet represents DSS pixel coordinates, and the current Frame represents DSS celestial coordinates. Such a read is always destructive and causes the FITS header cards required for the construction of the FrameSet to be removed from the FitsChan, which is then left positioned at the "end-of-file". A subsequent read using the same encoding will therefore not return another FrameSet, even if the FitsChan is rewound.

When astWrite is used to store a FrameSet using DSS encoding, an attempt is first made to simplify the FrameSet to see if it conforms to the DSS model. Specifically, the current Frame must be a FK5 SkyFrame; the projection must be a tangent plane (gnomonic) projection with polynomial corrections conforming to DSS requirements, and north must be parallel to the second base Frame axis.

If the simplification process succeeds, a description of the FrameSet is written to the FitsChan using appropriate DSS FITS header cards. The base Frame of the FrameSet is used to form the DSS pixel coordinate system and the current Frame gives the DSS celestial coordinate system. A successful write operation will over-write any existing DSS encoded data in the FitsChan, but will not affect other (non-DSS) header cards. If a destructive read of a DSS encoded Object has previously occurred, then an attempt will be made to store the FITS header cards back in their original locations.

If an attempt to simplify a FrameSet to conform to the DSS model fails (or if the Object supplied is not a FrameSet), then no data will be written to the FitsChan and astWrite will return zero. No error will result.

The FITS-WCS Encoding:

The FITS-WCS convention uses FITS header cards to describe the relationship between pixels in an image (not necessarily 2-dimensional) and one or more related "world coordinate systems". The FITS-WCS encoding may only be used to store a single AST Object in any set of FITS header cards, and that Object must be a FrameSet which conforms to the FITS-WCS model (the FrameSet may, however, contain multiple Frames which will be result in multiple FITS "alternate axis descriptions"). Details of the use made by this Encoding of the conventions described in the

FITS-WCS papers are given in the appendix "FITS-WCS Coverage" of this document. A few main points are described below.

The rotation and scaling of the intermediate world coordinate system can be specified using either "CDi_j" keywords, or "PCi_j" together with "CDELTi" keywords. When writing a FrameSet to a FitsChan, the the value of the CDMatrix attribute of the FitsChan determines which system is used.

In addition, this encoding supports the "TAN with polynomial correction terms" projection which was included in a draft of the FITS-WCS paper, but was not present in the final version. A "TAN with polynomial correction terms" projection is represented using a WcsMap with type AST__TPN (rather than AST__TAN which is used to represent simple TAN projections). When reading a FITS header, a CTYPE keyword value including a "-TAN" code results in an AST__TPN projection if there are any projection parameters (given by the PVi_m keywords) associated with the latitude axis, or if there are projection parameters associated with the longitude axis for m greater than 4. When writing a FrameSet to a FITS header, an AST__TPN projection gives rise to a CTYPE value including the normal "-TAN" code, but the projection parameters are stored in keywords with names "QVi_m", instead of the usual "PVi_m". Since these QV parameters are not part of the FITS-WCS standard they will be ignored by other non-AST software, resulting in the WCS being interpreted as a simple TAN projection without any corrections. This should be seen as an interim solution until such time as an agreed method for describing projection distortions within FITS-WCS has been published.

AST extends the range of celestial coordinate sytstems which may be described using this encoding by inclusion of the allowing the use of "AZ–" and "EL–" as the coordinate specification within CTYPE values. These form a longitude/latitude pair of axes which describe azimuth and elevation. The geographic position of the observer should be supplied using the OBSGEO-X/Y/Z keywords described in FITS-WCS paper III. Currently, a simple model is used which includes diurnal aberration, but ignores atmospheric refraction, polar motion, etc. These may be added in a leter release.

If an AST SkyFrame that represents offset rather than absolute coordinates (see attribute SkyRefIs) is written to a FitsChan using FITS-WCS encoding, two alternate axis descriptions will be created. One will describe the offset coordinates, and will use "OFLN" and "OFLT" as the axis codes in the CTYPE keywords. The other will describe absolute coordinates as specified by the System attribute of the SkyFrame, using the usual CTYPE codes ("RA–"/"DEC-", etc). Inaddition, the absolute coordinates description will contain AST-specific keywords (SREF1/2, SREFP1/2 and SREFIS) that allow the header to be read back into AST in order to reconstruct the original SkyFrame.

When reading a FITS-WCS encoded Object (using astRead), the FitsChan concerned must initially be positioned at the first card (its Card attribute must equal 1) and the result of the read, if successful, will always be a pointer to a FrameSet. The base Frame of this FrameSet represents FITS-WCS pixel coordinates, and the current Frame represents the physical coordinate system described by the FITS-WCS primary axis descriptions. If secondary axis descriptions are also present, then the FrameSet may contain additional (non-current) Frames which represent these. Such a read is always destructive and causes the FITS header cards required for the construction of the FrameSet to be removed from the FitsChan, which is then left positioned at the "end-of-file". A subsequent read using the same encoding will therefore not return another FrameSet, even if the FitsChan is rewound.

When astWrite is used to store a FrameSet using FITS-WCS encoding, an attempt is first made to simplify the FrameSet to see if it conforms to the FITS-WCS model. If this simplification process succeeds (as it often should, as the model is reasonably flexible), a description of the FrameSet is written to the FitsChan using appropriate FITS header cards. The base Frame of the FrameSet is used to form the FITS-WCS pixel coordinate system and the current Frame gives the physical coordinate system to be described by the FITS-WCS primary axis descriptions. Any additional Frames in the FrameSet may be used to construct secondary axis descriptions, where appropriate.

A successful write operation will over-write any existing FITS-WCS encoded data in the FitsChan, but will not affect other (non-FITS-WCS) header cards. If a destructive read of a FITS-WCS encoded Object has previously occurred, then an attempt will be made to store the FITS header cards back in their original locations. Otherwise, the new cards will be inserted following any other FITS-WCS related header cards present or, failing that, in front of the current card (as given by the Card attribute).

If an attempt to simplify a FrameSet to conform to the FITS-WCS model fails (or if the Object supplied is not a FrameSet), then no data will be written to the FitsChan and astWrite will return zero. No error will result.

The FITS-IRAF Encoding:

The FITS-IRAF encoding can, for most purposes, be considered as a subset of the FITS-WCS encoding (above), although it differs in the details of the FITS keywords used. It is used in exactly the same way and has the same restrictions, but with the

addition of the following:

- The only celestial coordinate systems that may be represented are equatorial, galactic and ecliptic,
- Sky projections can be represented only if any associated projection parameters are set to their default values.
- Secondary axis descriptions are not supported, so when writing a FrameSet to a FitsChan, only information from the base and current Frames will be stored.

Note that this encoding is provided mainly as an interim measure to provide a more stable alternative to the FITS-WCS encoding until the FITS standard for encoding WCS information is finalised. The name "FITS-IRAF" indicates the general keyword conventions used and does not imply that this encoding will necessarily support all features of the WCS scheme used by IRAF software. Nevertheless, an attempt has been made to support a few such features where they are known to be used by important sources of data.

When writing a FrameSet using the FITS-IRAF encoding, axis rotations are specified by a matrix of FITS keywords of the form "CDi_j", where "i" and "j" are single digits. The alternative form "CDiiijjj", which is also in use, is recognised when reading an Object, but is never written.

In addition, the experimental IRAF "ZPX" and "TNX" sky projections will be accepted when reading, but will never be written (the corresponding FITS "ZPN" or "distorted TAN" projection being used instead). However, there are restrictions on the use of these experimental projections. For "ZPX", longitude and latitude correction surfaces (appearing as "lngcor" or "latcor" terms in the IRAF-specific "WAT" keywords) are not supported. For "TNX" projections, only cubic surfaces encoded as simple polynomials with "half cross-terms" are supported. If an un-usable "TNX" or "ZPX" projection is encountered while reading from a FitsChan, a simpler form of TAN or ZPN projection is used which ignores the unsupported features and may therefore be inaccurate. If this happens, a warning message is added to the contents of the FitsChan as a set of cards using the keyword "ASTWARN".

You should not normally attempt to mix the foreign FITS encodings within the same FitsChan, since there is a risk that keyword clashes may occur.

The FITS-PC Encoding:

The FITS-PC encoding can, for most purposes, be considered as equivalent to the FITS-WCS encoding (above), although it differs in the details of the FITS keywords used. It is used in exactly the same way and has the same restrictions.

The FITS-AIPS Encoding:

The FITS-AIPS encoding can, for most purposes, be considered as equivalent to the FITS-WCS encoding (above), although it differs in the details of the FITS keywords used. It is used in exactly the same way and has the same restrictions, but with the

addition of the following:

- The only celestial coordinate systems that may be represented are equatorial, galactic and ecliptic,
- Spectral axes can only be represented if they represent frequency, radio velocity or optical velocity, and are linearly sampled in frequency. In addition, the standard of rest must be LSRK, LSRD, barycentric or geocentric.
- Sky projections can be represented only if any associated projection parameters are set to their default values.
- The AIT, SFL and MER projections can only be written if the CRVAL keywords are zero for both longitude and latitude axes.
- Secondary axis descriptions are not supported, so when writing a FrameSet to a FitsChan, only information from the base and current Frames will be stored.
- If there are more than 2 axes in the base and current Frames, any rotation must be restricted to the celestial plane, and must involve no shear.

The FITS-AIPS++ Encoding:

The FITS-AIPS++ encoding is based on the FITS-AIPS encoding, but includes some features of the FITS-IRAF and FITS-PC encodings. Specifically, any celestial projections supported by FITS-PC may be used, including those which require parameterisation, and the axis rotation and scaling may be specified using CDi_j keywords. When writing a FITS header, rotation will be specified using CROTA/CDELT keywords if possible, otherwise CDi_j keywords will be used instead.

The FITS-CLASS Encoding:

The FITS-CLASS encoding uses the conventions of the CLASS project. These are described in the section "Developer Manual"/"CLASS FITS

Format" contained in the CLASS documentation at:

http://www.iram.fr/IRAMFR/GILDAS/doc/html/class-html/class.html.

This encoding is similar to FITS-AIPS with the following restrictions:

- When a SpecFrame is created by reading a FITS-CLASS header, the attributes describing the observer's position (ObsLat, ObsLon and ObsAlt) are left unset because the CLASS encoding does not specify these values. Conversions to or from the topocentric standard of rest will therefore be inaccurate (typically by up to about 0.5 km/s) unless suitable values are assigned to these attributes after the FrameSet has been created.
- When writing a FrameSet to a FITS-CLASS header, the current Frame in the FrameSet must have at least 3 WCS axes, of which one must be a linear spectral axis. The spectral axis in the created header will always describe frequency. If the spectral axis in the supplied FrameSet refers to some other system (e.g. radio velocity, etc), then it will be converted to frequency.
- There must be a pair of celestial axes - either (RA,Dec) or (GLON,GLAT). RA and Dec must be either FK4/B1950 or FK5/J2000.
- A limited range of projection codes (TAN, ARC, STG, AIT, SFL, SIN) can be used. For AIT and SFL, the reference point must be at the origin of longitude and latitude. For SIN, the associated projection parameters must both be zero.
- No rotation of the celestial axes is allowed, unless the spatial axes are degenerate (i.e. cover only a single pixel).

- The frequency axis in the created header will always describe frequency in the source rest frame. If the supplied FrameSet uses some other standard of rest then suitable conversion will be applied.
- The source velocity must be defined. In other words, the SpecFrame attributes SourceVel and SourceVRF must have been assigned values.
- The frequency axis in a FITS-CLASS header does not represent absolute frequency, but instead represents offsets from the rest frequency in the standard of rest of the source.

When writing a FrameSet out using FITS-CLASS encoding, the current Frame may be temporarily modified if this will allow the header to be produced. If this is done, the associated pixel->WCS Mapping will also be modified to take account of the changes to the Frame. The modifications performed include re-ordering axes (WCS axes, not pixel axes), changing spectral coordinate system and standard of rest, changing the celestial coordinate system and reference equinox, and changing axis units.

The NATIVE Encoding:

The NATIVE encoding may be used to store a description of any class of AST Object in the form of FITS header cards, and (for most practical purposes) any number of these Object descriptions may be stored within a single set of FITS cards. If multiple Object descriptions are stored, they are written and read sequentially. The NATIVE encoding makes use of unique FITS keywords which are designed not to clash with keywords that have already been used for other purposes (if a potential clash is detected, an alternative keyword is constructed to avoid the clash).

When reading a NATIVE encoded object from a FitsChan (using astRead), FITS header cards are read, starting at the current card (as determined by the Card attribute), until the start of the next Object description is found. This description is then read and converted into an AST Object, for which a pointer is returned. Such a read is always destructive and causes all the FITS header cards involved in the Object description to be removed from the FitsChan, which is left positioned at the following card.

The Object returned may be of any class, depending on the description that was read, and other AST routines may be used to validate it (for example, by examining its Class or ID attribute using astGetC). If further NATIVE encoded Object descriptions exist in the FitsChan, subsequent calls to astRead will return the Objects they describe in sequence (and destroy their descriptions) until no more remain between the current card and the "end-of-file".

When astWrite is used to write an Object using NATIVE encoding, a description of the Object is inserted immediately before the current card (as determined by the Card attribute). Multiple Object descriptions may be written in this way and are stored separately (and sequentially if the Card attribute is not modified between the writes). A write operation using the NATIVE encoding does not over-write previously written Object descriptions. Note, however, that subsequent behaviour is undefined if an Object description is written inside a previously-written description, so this should be avoided.

When an Object is written to a FitsChan using NATIVE encoding, astWrite should (barring errors) always transfer data and return a value of 1.

Epoch Epoch of observation Epoch

Description: This attribute is used to qualify the coordinate systems described by a Frame, by giving the moment in time when the coordinates are known to be correct. Often, this will be the date of observation, and is important in cases where coordinates systems move with respect to each other over the course of time.

The Epoch attribute is stored as a Modified Julian Date, but when setting its value it may be given in a variety of formats. See the "Input Formats" section (below) for details. Strictly, the Epoch value should be supplied in the TDB timescale, but for some purposes (for instance, for converting sky positions between different types of equatorial system) the timescale is not significant, and UTC may be used.

Type:

Floating point.

Class Applicability:

Frame

All Frames have this attribute. The basic Frame class provides a default of J2000.0 (Julian) but makes no use of the Epoch value. This is because the Frame class does not distinguish between different Cartesian coordinate systems (see the System attribute).

CmpFrame

The default Epoch value for a CmpFrame is selected as follows; if the Epoch attribute has been set in the first component Frame then the Epoch value from the first component Frame is used as the default for the CmpFrame. Otherwise, if the Epoch attribute has been set in the second component Frame then the Epoch value from the second component Frame is used as the default for the CmpFrame. Otherwise, the default Epoch value from the first component Frame is used as the default for the CmpFrame. When the Epoch attribute of a CmpFrame is set or cleared, it is also set or cleared in the two component Frames.

FrameSet

The Epoch attribute of a FrameSet is the same as that of its current Frame (as specified by the Current attribute).

SkyFrame

The coordinates of sources within a SkyFrame can changed with time for various reasons, including: (i) changing aberration of light caused by the observer's velocity (e.g. due to the Earth's motion around the Sun), (ii) changing gravitational deflection by the Sun due to changes in the observer's position with time, (iii) fictitious motion due to rotation of non-inertial coordinate systems (e.g. the old FK4 system), and (iv) proper motion of the source itself (although this last effect is not handled by the SkyFrame class because it affects individual sources rather than the coordinate system as a whole).

The default Epoch value in a SkyFrame is B1950.0 (Besselian) for the old FK4-based coordinate systems (see the System attribute) and J2000.0 (Julian) for all others.

Care must be taken to distinguish the Epoch value, which relates to motion (or apparent motion) of the source, from the superficially similar Equinox value. The latter is used to qualify a coordinate system which is itself in motion in a (notionally) predictable way as a result of being referred to a slowly moving reference plane (e.g. the equator).

See the description of the System attribute for details of which qualifying attributes apply to each celestial coordinate system.

TimeFrame

A TimeFrame describes a general time axis and so cannot be completely characterised by a single Epoch value. For this reason the TimeFrame class makes no use of the Epoch attribute. However, user code can still make use of the attribute if necessary to represent a "typical" time spanned by the TimeFrame. The default Epoch value for a TimeFrame will be the TDB equivalent of the current value of the TimeFrame's TimeOrigin attribute. If no value has been set for TimeOrigin, then the default Epoch value is J2000.0.

Input Formats:

The formats accepted when setting an Epoch value are listed below. They are all case-insensitive and are generally tolerant of extra white space and alternative field delimiters:

- Besselian Epoch: Expressed in decimal years, with or without decimal places ("B1950" or "B1976.13" for example).

- Julian Epoch: Expressed in decimal years, with or without decimal places ("J2000" or "J2100.9" for example).
- Year: Decimal years, with or without decimal places ("1996.8" for example). Such values are interpreted as a Besselian epoch (see above) if less than 1984.0 and as a Julian epoch otherwise.
- Julian Date: With or without decimal places ("JD 2454321.9" for example).
- Modified Julian Date: With or without decimal places ("MJD 54321.4" for example).
- Gregorian Calendar Date: With the month expressed either as an integer or a 3-character abbreviation, and with optional decimal places to represent a fraction of a day ("1996-10-2" or "1996-Oct-2.6" for example). If no fractional part of a day is given, the time refers to the start of the day (zero hours).
- Gregorian Date and Time: Any calendar date (as above) but with a fraction of a day expressed as hours, minutes and seconds ("1996-Oct-2 12:13:56.985" for example). The date and time can be separated by a space or by a "T" (as used by ISO8601 format).

Output Format:

When enquiring Epoch values, the format used is the "Year" format described under "Input Formats". This is a value in decimal years which will be a Besselian epoch if less than 1984.0 and a Julian epoch otherwise. By omitting any character prefix, this format allows the Epoch value to be obtained as either a character string or a floating point value.

Equinox Epoch of the mean equinox Equinox

Description: This attribute is used to qualify those celestial coordinate systems described by a SkyFrame which are notionally based on the ecliptic (the plane of the Earth's orbit around the Sun) and/or the Earth's equator.

Both of these planes are in motion and their positions are difficult to specify precisely. In practice, therefore, a model ecliptic and/or equator are used instead. These, together with the point on the sky that defines the coordinate origin (the intersection of the two planes termed the "mean equinox") move with time according to some model which removes the more rapid fluctuations. The SkyFrame class supports both the FK4 and FK5 models.

The position of a fixed source expressed in any of these coordinate systems will appear to change with time due to movement of the coordinate system itself (rather than motion of the source). Such coordinate systems must therefore be qualified by a moment in time (the "epoch of the mean equinox" or "equinox" for short) which allows the position of the model coordinate system on the sky to be determined. This is the role of the Equinox attribute.

The Equinox attribute is stored as a Modified Julian Date, but when setting or getting its value you may use the same formats as for the Epoch attribute (q.v.).

The default Equinox value is B1950.0 (Besselian) for the old FK4-based coordinate systems (see the System attribute) and J2000.0 (Julian) for all others.

Type:

Floating point.

Class Applicability:

SkyFrame

All SkyFrames have this attribute.

Notes:

- Care must be taken to distinguish the Equinox value, which relates to the definition of a time-dependent coordinate system (based on solar system reference planes which are in motion), from the superficially similar Epoch value. The latter is used to qualify coordinate systems where the positions of sources change with time (or appear to do so) for a variety of other reasons, such as aberration of light caused by the observer's motion, etc.
- See the description of the System attribute for details of which qualifying attributes apply to each celestial coordinate system.

Escape Allow changes of character attributes within strings? Escape

Description: This attribute controls the appearance of text strings and numerical labels drawn by the astGrid and (for the Plot class) astText functions, by determining if any escape sequences contained within the strings should be used to control the appearance of the text, or should be printed literally. Note, the Plot3D class only interprets escape sequences within the astGrid function.

If the Escape value of a Plot is one (the default), then any escape sequences in text strings produce the effects described below when printed. Otherwise, they are printed literally.

See also the astEscapes function.

Type:

Integer (boolean).

Class Applicability:

Plot

All Plots have this attribute.

Escape Sequences:

Escape sequences are introduced into the text string by a percent "%" character. Any unrecognised, illegal or incomplete escape sequences are printed literally. The following escape sequences are currently recognised ("..." represents a string of one or more decimal digits):

%% - Print a literal "%" character.

%^...+ - Draw subsequent characters as super-scripts. The digits "..." give the distance from the base-line of "normal" text to the base-line of the super-script text, scaled so that a value of "100" corresponds to the height of "normal" text. %^+ - Draw subsequent characters with the normal base-line.

%v...+ - Draw subsequent characters as sub-scripts. The digits "..." give the distance from the base-line of "normal" text to the base-line of the sub-script text, scaled so that a value of "100" corresponds to the height of "normal" text.

%v+ - Draw subsequent characters with the normal base-line (equivalent to %^+).

%>...+ - Leave a gap before drawing subsequent characters. The digits "..." give the size of the gap, scaled so that a value of "100" corresponds to the height of "normal" text.

%<...+ - Move backwards before drawing subsequent characters. The digits "..." give the size of the movement, scaled so that a value of "100" corresponds to the height of "normal" text.

%s...+ - Change the Size attribute for subsequent characters. The digits "..." give the new Size as a fraction of the "normal" Size, scaled so that a value of "100" corresponds to 1.0;

%s+ - Reset the Size attribute to its "normal" value.

%w...+ - Change the Width attribute for subsequent characters. The digits "..." give the new width as a fraction of the "normal" Width, scaled so that a value of "100" corresponds to 1.0;

%w+ - Reset the Size attribute to its "normal" value.

%f...+ - Change the Font attribute for subsequent characters. The digits "..." give the new Font value.

%f+ - Reset the Font attribute to its "normal" value.

%c...+ - Change the Colour attribute for subsequent characters. The digits "..." give the new Colour value.

%c+ - Reset the Colour attribute to its "normal" value.

%t...+ - Change the Style attribute for subsequent characters. The digits "..." give the new Style value.

%t+ - Reset the Style attribute to its "normal" value.

%h+ - Remember the current horizontal position (see "%g+")

%g+ - Go to the horizontal position of the previous "%h+" (if any).

%- - Push the current graphics attribute values onto the top of the stack (see "%+").

%+ - Pop attributes values of the top the stack (see "%-"). If the stack is empty, "normal" attribute values are restored.

FillFactor Fraction of the Region which is of interest FillFactor

Description: This attribute indicates the fraction of the Region which is of interest. AST does not use this attribute internally for any purpose. Typically, it could be used to indicate the fraction of the Region for which data is available.

The supplied value must be in the range 0.0 to 1.0, and the default value is 1.0 (except as noted below).

Type:

Floating point.

Class Applicability:

Region

All Regions have this attribute.

CmpRegion

The default FillFactor for a CmpRegion is the FillFactor of its first component Region.

Prism

The default FillFactor for a Prism is the product of the FillFactors of its two component Regions.

Stc

The default FillFactor for an Stc is the FillFactor of its encapsulated Region.

FitsDigits Digits of precision for floating point FITS values FitsDigits

Description: This attribute gives the number of significant decimal digits to use when formatting floating point values for inclusion in the FITS header cards within a FitsChan.

By default, a positive value is used which results in no loss of information, assuming that the value's precision is double. Usually, this causes no problems.

However, to adhere strictly to the recommendations of the FITS standard, the width of the formatted value (including sign, decimal point and exponent) ought not to be more than 20 characters. If you are concerned about this, you should set FitsDigits to a negative value, such as -15. In this case, the absolute value (+15) indicates the maximum number of significant digits to use, but the actual number used may be fewer than this to ensure that the FITS recommendations are satisfied. When using this approach, the resulting number of significant digits may depend on the value being formatted and on the presence of any sign, decimal point or exponent.

The value of this attribute is effective when FITS header cards are output, either using astFindFits or by the action of the FitsChan's sink function when it is finally deleted.

Type:
Integer.

Class Applicability:

FitsChan
All FitsChans have this attribute.

Font(element) Character font for a Plot element Font(element)

Description: This attribute determines the character font index used when drawing each element of graphical output produced by a Plot. It takes a separate value for each graphical element so that, for instance, the setting "Font(title)=2" causes the Plot title to be drawn using font number 2.

The range of integer font indices available and the appearance of the resulting text is determined by the underlying graphics system. The default behaviour is for all graphical elements to be drawn using the default font supplied by this graphics system.

Type:
Integer.

Class Applicability:

Plot
All Plots have this attribute.

Notes:

- For a list of the graphical elements available, see the description of the Plot class.
- If no graphical element is specified, (e.g. "Font" instead of "Font(title)"), then a "set" or "clear" operation will affect the attribute value of all graphical elements, while a "get" or "test" operation will use just the Font(TextLab) value.

Format(axis) Format specification for axis values Format(axis)

Description: This attribute specifies the format to be used when displaying coordinate values associated with a particular Frame axis (i.e. to convert values from binary to character form). It is interpreted by the astFormat function and determines the formatting which it applies.

If no Format value is set for a Frame axis, a default value is supplied instead. This is based on the value of the Digits, or Digits(axis), attribute and is chosen so that it displays the requested number of digits of precision.

Type:
String.

Class Applicability:

Frame
The Frame class interprets this attribute as a format specification string to be passed to the C "printf" function (e.g. "%1.7G") in order to format a single coordinate value (supplied as a double precision number).

When supplying a value for this attribute, beware that the "%" character may be interpreted directly as a format specification by some printf-like functions (such as astSet). You may need to double it (i.e. use "%%") to avoid this.

SkyFrame
The SkyFrame class re-defines the syntax and default value of the Format string to allow the formatting of sexagesimal values as appropriate for the particular celestial coordinate system being represented. The syntax of SkyFrame Format strings is described (below) in the "SkyFrame Formats" section.

FrameSet
The Format attribute of a FrameSet axis is the same as that of its current Frame (as specified by the Current attribute). Note that the syntax of the Format string is also determined by the current Frame.

TimeFrame
The TimeFrame class extends the syntax of the Format string to allow the formatting of TimeFrame axis values as Gregorian calendar dates and times. The syntax of TimeFrame Format strings is described (below) in the "TimeFrame Formats" section.

Notes:

- When specifying this attribute by name, it should be subscripted with the number of the Frame axis to which it applies.

SkyFrame Formats:

The Format string supplied for a SkyFrame should contain zero or more of the following characters. These may occur in any order, but the following is recommended for clarity:

- "+": Indicates that a plus sign should be prefixed to positive values. By default, no plus sign is used.
- "z": Indicates that leading zeros should be prefixed to the value so that the first field is of constant width, as would be required in a fixed-width table (leading zeros are always prefixed to any fields that follow). By default, no leading zeros are added.
- "i": Use the standard ISO field separator (a colon) between fields. This is the default behaviour.
- "b": Use a blank to separate fields.
- "l": Use a letter ("h"/"d", "m" or "s" as appropriate) to separate fields.
- "g": Use a letter and symbols to separate fields ("h"/"d", "m" or "s", etc, as appropriate), but include escape sequences in the formatted value so that the Plot class will draw the separators as small super-scripts. The default escape sequences are optimised for the pgplot graphics package, but new escape sequences may be specified using function astSetSkyDelim.
- "d": Include a degrees field. Expressing the angle purely in degrees is also the default if none of "h", "m", "s" or "t" are given.
- "h": Express the angle as a time and include an hours field (where 24 hours correspond to 360 degrees). Expressing the angle purely in hours is also the default if "t" is given without either "m" or "s".
- "m": Include a minutes field. By default this is not included.
- "s": Include a seconds field. By default this is not included. This request is ignored if "d" or "h" is given, unless a minutes field is also included.
- "t": Express the angle as a time (where 24 hours correspond to 360 degrees). This option is ignored if either "d" or "h" is given and is intended for use where the value is to be expressed purely in minutes and/or seconds of time (with no hours field). If "t" is given without "d", "h", "m" or "s" being present, then it is equivalent to "h".

- ".": Indicates that decimal places are to be given for the final field in the formatted string (whichever field this is). The "." should be followed immediately by an unsigned integer which gives the number of decimal places required, or by an asterisk. If an asterisk is supplied, a default number of decimal places is used which is based on the value of the Digits attribute.

All of the above format specifiers are case-insensitive. If several characters make conflicting requests (e.g. if both "i" and "b" appear), then the character occurring last takes precedence, except that "d" and "h" always override "t".

If the format string starts with a percentage sign (%), then the whole format string is assumed to conform to the syntax defined by the Frame class, and the axis values is formated as a decimal radians value.

TimeFrame Formats:

The Format string supplied for a TimeFrame should either use the syntax defined by the base Frame class (i.e. a C "printf" format string), or the extended "iso" syntax described below (the default value is inherited from the Frame class):

- C "printf" syntax: If the Format string is a C "printf" format description such as "%1.7G", the TimeFrame axis value will be formatted without change as a floating point value using this format. The formatted string will thus represent an offset from the zero point specified by the TimeFrame's TimeOrigin attribute, measured in units given by the TimeFrame's Unit attribute.
- "iso" syntax: This is used to format a TimeFrame axis value as a Gregorian date followed by an optional time of day. If the Format value commences with the string "iso" then the TimeFrame axis value will be converted to an absolute MJD, including the addition of the current TimeOrigin value, and then formatted as a Gregorian date using the format "yyyy-mm-dd". Optionally, the Format value may include an integer precision following the "iso" specification (e.g. "iso.2"), in which case the time of day will be appended to the formatted date (if no time of day is included, the date field is rounded to the nearest day). The integer value in the Format string indicates the number of decimal places to use in the seconds field. For instance, a Format value of "iso.0" produces a time of day of the form "hh:mm:ss", and a Format value of "iso.2" produces a time of day of the form "hh:mm:ss.ss". The date and time fields will be separated by a space unless 'T' is appended to the end of string, in which case the letter T (upper case) will be used as the separator. The value of the Digits attribute is ignored when using this "iso" format.

Full Set level of output detail Full

Description: This attribute is a three-state flag and takes values of -1, 0 or +1. It controls the amount of information included in the output generated by a Channel.

If Full is zero, then a modest amount of non-essential but useful information will be included in the output. If Full is negative, all non-essential information will be suppressed to minimise the amount of output, while if it is positive, the output will include the maximum amount of detailed information about the Object being written.

Type:

Integer.

Class Applicability:

Channel

The default value is zero for a normal Channel.

FitsChan

The default value is zero for a FitsChan.

XmlChan
The default value is -1 for an XmlChan.

StcsChan
The default value is zero for an StcsChan. Set a positive value to cause default values to be included in STC-S descriptions.

Notes:

- All positive values supplied for this attribute are converted to +1 and all negative values are converted to -1.

Gap(axis) Interval between linearly spaced major axis values of a Plot Gap(axis)

Description: This attribute controls the appearance of an annotated coordinate grid (drawn with the astGrid function) by determining the linear interval between the "major" axis values of a Plot, at which (for example) major tick marks are drawn. It takes a separate value for each physical axis of the Plot so that, for instance, the setting "Gap(2)=3.0" specifies the difference between adjacent major values along the second axis. The Gap attribute is only used when the LogTicks attribute indicates that the spacing between major axis values is to be linear. If major axis values are logarithmically spaced then the gap is specified using attribute LogGap.

The Gap value supplied will usually be rounded to the nearest "nice" value, suitable (e.g.) for generating axis labels, before use. To avoid this "nicing" you should set an explicit format for the axis using the Format(axis) or Digits/Digits(axis) attribute. The default behaviour is for the Plot to generate its own Gap value when required, based on the range of axis values to be represented.

Type:
Floating point.

Class Applicability:

Plot
All Plots have this attribute.

Notes:

- The Gap value should use the same units as are used internally for storing coordinate values on the corresponding axis. For example, with a celestial coordinate system, the Gap value should be in radians, not hours or degrees.
- If no axis is specified, (e.g. "Gap" instead of "Gap(2)"), then a "set" or "clear" operation will affect the attribute value of all the Plot axes, while a "get" or "test" operation will use just the Gap(1) value.

Grf Use Grf functions registered through astGrfSet? Grf

Description: This attribute selects the functions which are used to draw graphics by the Plot class. If it is zero, then the functions in the graphics interface selected at link-time are used (see the ast_link script). Otherwise, functions registered using astGrfSet are used. In this case, if a function is needed which has not been registered, then the function in the graphics interface selected at link-time is used.

The default is to use the graphics interface

Type:
Integer (boolean).

Class Applicability:

Plot
All Plots have this attribute.

Plot3D
The Plot3D class ignores this attributes, assuming a value of zero.

Notes:

- The value of this attribute is not saved when the Plot is written out through a Channel to an external data store. On re-loading such a Plot using astRead, the attribute will be cleared, resulting in the graphics interface selected at link-time being used.

Grid Draw grid lines for a Plot? Grid

Description: This attribute controls the appearance of an annotated coordinate grid (drawn with the astGrid function) by determining whether grid lines (a grid of curves marking the "major" values on each axis) are drawn across the plotting area.

If the Grid value of a Plot is non-zero, then grid lines will be drawn. Otherwise, short tick marks on the axes are used to mark the major axis values. The default behaviour is to use tick marks if the entire plotting area is filled by valid physical coordinates, but to draw grid lines otherwise.

Type:
Integer (boolean).

Class Applicability:

Plot
All Plots have this attribute.

Notes:

- The spacing between major axis values, which determines the spacing of grid lines, may be set using the Gap(axis) attribute.

GrismAlpha The angle of incidence of the incoming light on the grating surface GrismAlpha

Description: This attribute holds the angle between the incoming light and the normal to the grating surface, in radians. The default value is 0.

Type:
Double precision.

Class Applicability:

GrismMap
All GrismMaps have this attribute.

GrismEps The angle between the normal and the dispersion plane **GrismEps**

Description: This attribute holds the angle (in radians) between the normal to the grating or exit prism face, and the dispersion plane. The dispersion plane is the plane spanned by the incoming and outgoing ray. The default value is 0.0.

Type:
Double precision.

Class Applicability:

GrismMap
All GrismMaps have this attribute.

GrismG The grating ruling density **GrismG**

Description: This attribute holds the number of grating rulings per unit length. The unit of length used should be consistent with the units used for attributes GrismWaveR and GrismNRP. The default value is 0.0. (the appropriate value for a pure prism disperser with no grating).

Type:
Double precision.

Class Applicability:

GrismMap
All GrismMaps have this attribute.

GrismM The interference order **GrismM**

Description: This attribute holds the interference order being considered. The default value is 0.

Type:
Integer.

Class Applicability:

GrismMap
All GrismMaps have this attribute.

GrismNR The refractive index at the reference wavelength **GrismNR**

Description: This attribute holds refractive index of the grism material at the reference wavelength (given by attribute GrismWaveR). The default value is 1.0.

Type:
Double precision.

Class Applicability:

GrismMap
All GrismMaps have this attribute.

GrismNRP The rate of change of refractive index with wavelength GrismNRP

Description: This attribute holds the rate of change of the refractive index of the grism material with respect to wavelength at the reference wavelength (given by attribute GrismWaveR). The default value is 0.0 (the appropriate value for a pure grating disperser with no prism). The units of this attribute should be consistent with those of attributes GrismWaveR and GrismG.

Type:
Double precision.

Class Applicability:

GrismMap
All GrismMaps have this attribute.

GrismTheta Angle between normal to detector plane and reference ray GrismTheta

Description: This attribute gives the angle of incidence of light of the reference wavelength (given by attribute GrismWaveR) onto the detector. Specifically, it holds the angle (in radians) between the normal to the detector plane and an incident ray at the reference wavelength. The default value is 0.0.

Type:
Double precision.

Class Applicability:

GrismMap
All GrismMaps have this attribute.

GrismWaveR The reference wavelength GrismWaveR

Description: This attribute holds reference wavelength. The default value is 5000 (Angstrom). The units of this attribute should be consistent with those of attributes GrismNRP and GrismG.

Type:
Double precision.

Class Applicability:

GrismMap
All GrismMaps have this attribute.

ID Object identification string ID

Description: This attribute contains a string which may be used to identify the Object to which it is attached. There is no restriction on the contents of this string, which is not used internally by the AST library, and is simply returned without change when required. The default value is an empty string.

An identification string can be valuable when, for example, several Objects have been stored in a file (using astWrite) and are later retrieved (using astRead). Consistent use of the ID attribute allows the retrieved Objects to be identified without depending simply on the order in which they were stored.

This attribute may also be useful during debugging, to distinguish similar Objects when using astShow to display them.

Type:

String.

Class Applicability:

Object

All Objects have this attribute.

Notes:

- Unlike most other attributes, the value of the ID attribute is not transferred when an Object is copied. Instead, its value is undefined (and therefore defaults to an empty string) in any copy. However, it is retained in any external representation of an Object produced by the astWrite function.

IF The intermediate frequency in a dual sideband spectrum IF

Description: This attribute specifies the (topocentric) intermediate frequency in a dual sideband spectrum. Its sole use is to determine the local oscillator (LO) frequency (the frequency which marks the boundary between the lower and upper sidebands). The LO frequency is equal to the sum of the centre frequency and the intermediate frequency. Here, the "centre frequency" is the topocentric frequency in Hz corresponding to the current value of the DSBCentre attribute. The value of the IF attribute may be positive or negative: a positive value results in the LO frequency being above the central frequency, whilst a negative IF value results in the LO frequency being below the central frequency. The sign of the IF attribute value determines the default value for the SideBand attribute.

When setting a new value for this attribute, the units in which the frequency value is supplied may be indicated by appending a suitable string to the end of the formatted value. If the units are not specified, then the supplied value is assumed to be in units of GHz. For instance, the following strings all result in an IF of 4 GHz being used: "4.0", "4.0 GHz", "4.0E9 Hz", etc.

When getting the value of this attribute, the returned value is always in units of GHz. The default value for this attribute is 4 GHz.

Type:

Floating point.

Class Applicability:

DSBSpecFrame

All DSBSpecFrames have this attribute.

Ident Permanent Object identification string Ident

Description: This attribute is like the ID attribute, in that it contains a string which may be used to identify the Object to which it is attached. The only difference between ID and Ident is that Ident is transferred when an Object is copied, but ID is not.

Type:

String.

Class Applicability:

Object
All Objects have this attribute.

ImagFreq The image sideband equivalent of the rest frequency ImagFreq

Description: This is a read-only attribute giving the frequency which corresponds to the rest frequency but is in the opposite sideband.

The value is calculated by first transforming the rest frequency (given by the RestFreq attribute) from the standard of rest of the source (given by the SourceVel and SourceVRF attributes) to the standard of rest of the observer (i.e. the topocentric standard of rest). The resulting topocentric frequency is assumed to be in the same sideband as the value given for the DSBCentre attribute (the "observed" sideband), and is transformed to the other sideband (the "image" sideband). The new frequency is converted back to the standard of rest of the source, and the resulting value is returned as the attribute value, in units of GHz.

Type:
Floating point, read-only.

Class Applicability:

DSBSpecFrame
All DSBSpecFrames have this attribute.

Indent Specifies the indentation to use in text produced by a Channel Indent

Description: This attribute controls the indentation within the output text produced by the astWrite function. It gives the increase in the indentation for each level in the object heirarchy. If it is set to zero, no indentation will be used. [3]

Type:
Integer (boolean).

Class Applicability:

Channel
The default value is zero for a basic Channel.

FitsChan
The FitsChan class ignores this attribute.

StcsChan
The default value for an StcsChan is zero, which causes the entire STC-S description is written out by a single invocation of the sink function. The text supplied to the sink function will not contain any linefeed characters, and each pair of adjacent words will be separated by a single space. The text may thus be arbitrarily large and the StcsLength attribute is ignored.

If Indent is non-zero, then the text is written out via multiple calls to the sink function, each call corresponding to a single "line" of text (although no line feed characters will be inserted by AST). The complete STC-S description is broken into lines so that:

- the line length specified by attribute StcsLength is not exceeded
- each sub-phrase (time, space, etc.) starts on a new line
- each argument in a compound spatial region starts on a new line

If this causes a sub-phrase to extend to two or more lines, then the second and subsequent lines will be indented by three spaces compared to the first line. In addition, lines within a compound spatial region will have extra indentation to highlight the nesting produced by the parentheses. Each new level of nesting will be indented by a further three spaces.

XmlChan

The default value for an XmlChan is zero, which results in no linefeeds or indentation strings being added to output text. If any non-zero value is assigned to Indent, then extra linefeed and space characters will be inserted as necessary to ensure that each XML tag starts on a new line, and each tag will be indented by a further 3 spaces to show its depth in the containment hierarchy.

IntraFlag IntraMap identification string IntraFlag

Description: This attribute allows an IntraMap to be flagged so that it is distinguishable from other IntraMaps. The transformation function associated with the IntraMap may then enquire the value of this attribute and adapt the transformation it provides according to the particular IntraMap involved.

Although this is a string attribute, it may often be useful to store numerical values here, encoded as a character string, and to use these as data within the transformation function. Note, however, that this mechanism is not suitable for transferring large amounts of data (more than about 1000 characters) to an IntraMap. For that purpose, global variables are recommended, although the IntraFlag value can be used to supplement this approach. The default IntraFlag value is an empty string.

Type:

String.

Class Applicability:

IntraMap

All IntraMaps have this attribute.

Notes:

- A pair of IntraMaps whose transformations may potentially cancel cannot be simplified to produce a UnitMap (e.g. using astSimplify) unless they have the same IntraFlag values. The test for equality is case-sensitive.

Invert Mapping inversion flag Invert

Description: This attribute controls which one of a Mapping's two possible coordinate transformations is considered the "forward" transformation (the other being the "inverse" transformation). If the attribute value is zero (the default), the Mapping's behaviour will be the same as when it was first created. However, if it is non-zero, its two transformations will be inter-changed, so that the Mapping displays the inverse of its original behaviour.

Inverting the boolean sense of the Invert attribute will cause the values of a Mapping's Nin and Nout attributes to be interchanged. The values of its TranForward and TranInverse attributes will also be interchanged. This operation may be performed with the astInvert function.

Type:

Integer (boolean).

Class Applicability:

Mapping
All Mappings have this attribute.

UnitMap
The value of the Invert attribute has no effect on the behaviour of a UnitMap.

FrameSet
Inverting the boolean sense of the Invert attribute for a FrameSet will cause its base and current Frames (and its Base and Current attributes) to be interchanged. This, in turn, may affect other properties and attributes of the FrameSet (such as Nin, Nout, Naxes, TranForward, TranInverse, etc.). The Invert attribute of a FrameSet is not itself affected by selecting a new base or current Frame.

Invisible Draw graphics using invisible ink? Invisible

Description: This attribute controls the appearance of all graphics produced by Plot methods by determining whether graphics should be visible or invisible.

If the Invisible value of a Plot is non-zero, then all the Plot methods which normally generate graphical output do not do so (you can think of them drawing with "invisible ink"). Such methods do, however, continue to do all the calculations which would be needed to produce the graphics. In particular, the bounding box enclosing the graphics is still calculated and can be retrieved as normal using astBoundingBox. The default value is zero, resulting in all methods drawing graphics as normal, using visible ink.

Type:
Integer (boolean).

Class Applicability:

Plot
All Plots have this attribute.

IsLatAxis(axis) Is the specified celestial axis a latitude axis? IsLatAxis(axis)

Description: This is a read-only boolean attribute that indicates the nature of the specified axis. The attribute has a non-zero value if the specified axis is a celestial latitude axis (Declination, Galactic latitude, etc), and is zero otherwise.

Type:
Integer (boolean), read-only.

Class Applicability:

SkyFrame
All SkyFrames have this attribute.

Notes:

- When specifying this attribute by name, it should be subscripted with the number of the SkyFrame axis to be tested.

IsLinear Is the Mapping linear? IsLinear

Description: This attribute indicates whether a Mapping is an instance of a class that always represents a linear transformation. Note, some Mapping classes can represent linear or non-linear transformations (the MathMap class for instance). Such classes have a zero value for the IsLinear attribute. Specific instances of such classes can be tested for linearity using the astLinearApprox function. AST_LINEARAPPROX routine.

Type:
Integer (boolean), read-only.

Class Applicability:

Mapping
All Mappings have this attribute.

CmpMap
The IsLinear value for a CmpMap is determined by the classes of the encapsulated Mappings. For instance, a CmpMap that combines a ZoomMap and a ShiftMap will have a non-zero value for its IsLinear attribute, but a CmpMap that contains a MathMap will have a value of zero for its IsLinear attribute.

Frame
The IsLinear value for a Frame is 1 (since a Frame is equivalent to a UnitMap).

FrameSet
The IsLinear value for a FrameSet is obtained from the Mapping from the base Frame to the current Frame.

IsLonAxis(axis) Is the specified celestial axis a longitude axis? IsLonAxis(axis)

Description: This is a read-only boolean attribute that indicates the nature of the specified axis. The attribute has a non-zero value if the specified axis is a celestial longitude axis (Right Ascension, Galactic longitude, etc), and is zero otherwise.

Type:
Integer (boolean), read-only.

Class Applicability:

SkyFrame
All SkyFrames have this attribute.

Notes:

- When specifying this attribute by name, it should be subscripted with the number of the SkyFrame axis to be tested.

IsSimple Has the Mapping been simplified? IsSimple

Description: This attribute indicates whether a Mapping has been simplified by the astSimplify method. If the IsSimple value is non-zero, then the Mapping has been simplified and so there is nothing to be gained by simplifying it again. Indeed, the astSimplify method will immediately return the Mapping unchanged if the IsSimple attribute indicates that the Mapping has already been simplified.

Type:
Integer (boolean), read-only.

Class Applicability:

Mapping
All Mappings have this attribute.

Frame
All classes of Frame return zero for the IsSimple attribute. This is because changes can be made to a Frame which affect the Mapping represented by the Frame, and so there can be no guarantee that the Mapping may not need re-simplifying. Most non-Frame Mappings, on the other hand, are immutable and so when they are simplified it is certain that they weill remain in a simple state.

IterInverse Provide an iterative inverse transformation? IterInverse

Description: This attribute indicates whether the inverse transformation of the PolyMap should be implemented via an iterative Newton-Raphson approximation that uses the forward transformation to transform candidate input positions until an output position is found which is close to the required output position. By default, an iterative inverse is provided if, and only if, no inverse polynomial was supplied when the PolyMap was constructed.

The NiterInverse and TolInverse attributes provide parameters that control the behaviour of the inverse approcimation method.

Type:
Integer (boolean).

Class Applicability:

PolyMap
All PolyMaps have this attribute.

Notes:

- An iterative inverse can only be used if the PolyMap has equal numbers of inputs and outputs, as given by the Nin and Nout attributes. An error will be reported if IterInverse is set non-zero for a PolyMap that does not meet this requirement.

Iwc Include a Frame representing FITS-WCS intermediate world coordinates? Iwc

Description: This attribute is a boolean value which is used when a FrameSet is read from a FitsChan with a foreign FITS encoding (e.g. FITS-WCS) using astRead. If it has a non-zero value then the returned FrameSet will include Frames representing "intermediate world coordinates" (IWC). These Frames will have Domain name "IWC" for primary axis descriptions, and "IWCa" for secondary axis descriptions, where "a" is replaced by the single alternate axis description character, as used in the FITS-WCS header. The default value for "Iwc" is zero.

FITS-WCS paper 1 defines IWC as a Cartesian coordinate system with one axis for each WCS axis, and is the coordinate system produced by the rotation matrix (represented by FITS keyword PCi_j, CDi_j, etc). For instance, for a 2-D FITS-WCS header describing projected celestial longitude and latitude, the intermediate world coordinates represent offsets in degrees from the reference point within the plane of projection.

Type:
Integer (boolean).

Class Applicability:

FitsChan
All FitsChans have this attribute.

KeyCase Are keys case sensitive? KeyCase

Description: This attribute is a boolean value which controls how keys are used. If KeyCase is zero, then key strings supplied to any method are automatically converted to upper case before being used. If KeyCase is non-zero (the default), then supplied key strings are used without modification.

The value of this attribute can only be changed if the KeyMap is empty. Its value can be set conveniently when creating the KeyMap. An error will be reported if an attempt is made to change the attribute value when the KeyMap contains any entries.

Type:
Integer (boolean).

Class Applicability:

KeyMap
All KeyMaps have this attribute.

Table
The Table class over-rides this attribute by forcing it to zero. That is, keys within a Table are always case insensitive.

KeyError Report an error when getting the value of a non-existant KeyMap entry? KeyError

Description: This attribute is a boolean value which controls how the astMapGet... functions behave if the requested key is not found in the KeyMap. If KeyError is zero (the default), then these functions will return zero but no error will be reported. If KeyError is non-zero, then the same values are returned but an error is also reported.

Type:
Integer (boolean).

Class Applicability:

KeyMap
All KeyMaps have this attribute.

Notes:

- When setting a new value for KeyError, the supplied value is propagated to any KeyMaps contained within the supplied KeyMap.
- When clearing the KeyError attribute, the attribute is also cleared in any KeyMaps contained within the supplied KeyMap.

LTOffset The offset from UTC to Local Time, in hours LTOffset

Description: This specifies the offset from UTC to Local Time, in hours (fractional hours can be supplied). It is positive for time zones east of Greenwich. AST uses the figure as given, without making any attempt to correct for daylight saving. The default value is zero.

Type:
Floating point.

Class Applicability:

TimeFrame
All TimeFrames have this attribute.

Label(axis) Axis label Label(axis)

Description: This attribute specifies a label to be attached to each axis of a Frame when it is represented (e.g.) in graphical output.

If a Label value has not been set for a Frame axis, then a suitable default is supplied.

Type:
String.

Class Applicability:

Frame
The default supplied by the Frame class is the string "Axis <n>", where <n> is 1, 2, etc. for each successive axis.

SkyFrame
The SkyFrame class re-defines the default Label value (e.g. to "Right ascension" or "Galactic latitude") as appropriate for the particular celestial coordinate system being represented.

TimeFrame
The TimeFrame class re-defines the default Label value as appropriate for the particular time system being represented.

FrameSet
The Label attribute of a FrameSet axis is the same as that of its current Frame (as specified by the Current attribute).

Notes:

- Axis labels are intended purely for interpretation by human readers and not by software.
- When specifying this attribute by name, it should be subscripted with the number of the Frame axis to which it applies.

LabelAt(axis) Where to place numerical labels for a Plot LabelAt(axis)

Description: This attribute controls the appearance of an annotated coordinate grid (drawn with the astGrid function) by determining where numerical axis labels and associated tick marks are placed. It takes a separate value for each physical axis of a Plot so that, for instance, the setting "LabelAt(2)=10.0" specifies where the numerical labels and tick marks for the second axis should be drawn.

For each axis, the LabelAt value gives the value on the other axis at which numerical labels and tick marks should be placed (remember that Plots suitable for use with astGrid may only have two axes). For example, in a celestial (RA,Dec) coordinate system, LabelAt(1) gives a Dec value which defines a line (of constant Dec) along which the numerical RA labels and their associated tick marks will be drawn. Similarly, LabelAt(2) gives the RA value at which the Dec labels and ticks will be drawn.

The default bahaviour is for the Plot to generate its own position for numerical labels and tick marks.

Type:

Floating point.

Class Applicability:

Plot

All Plots have this attribute.

Notes:

- The LabelAt value should use the same units as are used internally for storing coordinate values on the appropriate axis. For example, with a celestial coordinate system, the LabelAt value should be in radians, not hours or degrees.
- Normally, the LabelAt value also determines where the lines representing coordinate axes will be drawn, so that the tick marks will lie on these lines (but also see the DrawAxes attribute).
- In some circumstances, numerical labels and tick marks are drawn around the edges of the plotting area (see the Labelling attribute). In this case, the value of the LabelAt attribute is ignored.

LabelUnits(axis) Use axis unit descriptions in a Plot? LabelUnits(axis)

Description: This attribute controls the appearance of an annotated coordinate grid (drawn with the astGrid function) by determining whether the descriptive labels drawn for each axis of a Plot should include a description of the units being used on the axis. It takes a separate value for each physical axis of a Plot so that, for instance, the setting "LabelUnits(2)=1" specifies that a unit description should be included in the label for the second axis.

If the LabelUnits value of a Plot axis is non-zero, a unit description will be included in the descriptive label for that axis, otherwise it will be omitted. The default behaviour is to include a unit description unless the current Frame of the Plot is a SkyFrame representing equatorial, ecliptic, galactic or supergalactic coordinates, in which case it is omitted.

Type:

Integer (boolean).

Class Applicability:

Plot

All Plots have this attribute.

Notes:

- The text used for the unit description is obtained from the Plot's Unit(axis) attribute.
- If no axis is specified, (e.g. "LabelUnits" instead of "LabelUnits(2)"), then a "set" or "clear" operation will affect the attribute value of all the Plot axes, while a "get" or "test" operation will use just the LabelUnits(1) value.

- If the current Frame of the Plot is not a SkyFrame, but includes axes which were extracted from a SkyFrame, then the default behaviour is to include a unit description only for those axes which were not extracted from a SkyFrame.

LabelUp(axis) Draw numerical Plot labels upright? LabelUp(axis)

Description: This attribute controls the appearance of an annotated coordinate grid (drawn with the astGrid function) by determining whether the numerical labels for each axis of a Plot should be drawn upright or not. It takes a separate value for each physical axis of a Plot so that, for instance, the setting "LabelUp(2)=1" specifies that numerical labels for the second axis should be drawn upright.

If the LabelUp value of a Plot axis is non-zero, it causes numerical labels for that axis to be plotted upright (i.e. as normal, horizontal text), otherwise labels are drawn parallel to the axis to which they apply.

The default is to produce upright labels if the labels are placed around the edge of the plot, and to produce labels that follow the axes if the labels are placed within the interior of the plot (see attribute Labelling).

Type:

Integer (boolean).

Class Applicability:

Plot

All Plots have this attribute.

Notes:

- In some circumstances, numerical labels and tick marks are drawn around the edges of the plotting area (see the Labelling attribute). In this case, the value of the LabelUp attribute is ignored.
- If no axis is specified, (e.g. "LabelUp" instead of "LabelUp(2)"), then a "set" or "clear" operation will affect the attribute value of all the Plot axes, while a "get" or "test" operation will use just the LabelUp(1) value.

Labelling Label and tick placement option for a Plot Labelling

Description: This attribute controls the appearance of an annotated coordinate grid (drawn with the astGrid function) by determining the strategy for placing numerical labels and tick marks for a Plot.

If the Labelling value of a Plot is "exterior" (the default), then numerical labels and their associated tick marks are placed around the edges of the plotting area, if possible. If this is not possible, or if the Labelling value is "interior", then they are placed along grid lines inside the plotting area.

Type:

String.

Class Applicability:

Plot

All Plots have this attribute.

Notes:

- The LabelAt(axis) attribute may be used to determine the exact placement of labels and tick marks that are drawn inside the plotting area.

LatAxis Index of the latitude axis LatAxis

Description: This read-only attribute gives the index (1 or 2) of the latitude axis within the SkyFrame (taking into account any current axis permutations).

Type:

Integer.

Class Applicability:

SkyFrame

All SkyFrames have this attribute.

ListSize Number of points in a PointList ListSize

Description: This is a read-only attribute giving the number of points in a PointList. This value is determined when the PointList is created.

Type:

Integer, read-only.

Class Applicability:

PointList

All PointLists have this attribute.

LogGap(axis) Interval between major axis values of a Plot LogGap(axis)

Description: This attribute controls the appearance of an annotated coordinate grid (drawn with the astGrid function) by determining the logarithmic interval between the "major" axis values of a Plot, at which (for example) major tick marks are drawn. It takes a separate value for each physical axis of the Plot so that, for instance, the setting "LogGap(2)=100.0" specifies the ratio between adjacent major values along the second axis. The LogGap attribute is only used when the LogTicks attribute indicates that the spacing between major axis values is to be logarithmic. If major axis values are linearly spaced then the gap is specified using attribute Gap.

The LogGap value supplied will be rounded to the nearest power of 10. The reciprocal of the supplied value may be used if this is necessary to produce usable major axis values. If a zero or negative value is supplied, an error will be reported when the grid is drawn. The default behaviour is for the Plot to generate its own LogGap value when required, based on the range of axis values to be represented.

Type:

Floating point.

Class Applicability:

Plot

All Plots have this attribute.

Notes:

- The LogGap value is a ratio between axis values and is therefore dimensionless.

- If no axis is specified, (e.g. "LogGap" instead of "LogGap(2)"), then a "set" or "clear" operation will affect the attribute value of all the Plot axes, while a "get" or "test" operation will use just the LogGap(1) value.

LogLabel(axis) Use exponential format for numerical axis labels? LogLabel(axis)

Description: This attribute controls the appearance of an annotated coordinate grid (drawn with the astGrid function) by determining whether the numerical axis labels should be in normal decimal form or should be represented as 10 raised to the appropriate power. That is, an axis value of 1000.0 will be drawn as "1000.0" if LogLabel is zero, but as "10∧3" if LogLabel is non-zero. If graphical escape sequences are supported (see attribute Escape), the power in such exponential labels will be drawn as a small superscript instead of using a "∧" character to represent exponentiation.

The default is to produce exponential labels if the major tick marks are logarithmically spaced (see the LogTicks attribute).

Type:
Integer (boolean).

Class Applicability:

Plot
All Plots have this attribute.

Notes:

- If no axis is specified, (e.g. "LogLabel" instead of "LogLabel(2)"), then a "set" or "clear" operation will affect the attribute value of all the Plot axes, while a "get" or "test" operation will use just the LogLabel(1) value.

LogPlot(axis) Map the plot logarithmically onto the screen? LogPlot(axis)

Description: This attribute controls the appearance of all graphics produced by the Plot, by determining whether the axes of the plotting surface are mapped logarithmically or linearly onto the base Frame of the FrameSet supplied when the Plot was constructed. It takes a separate value for each axis of the graphics coordinate system (i.e. the base Frame in the Plot) so that, for instance, the setting "LogPlot(2)=1" specifies that the second axis of the graphics coordinate system (usually the vertical axis) should be mapped logarithmically onto the second axis of the base Frame of the FrameSet supplied when the Plot was constructed.

If the LogPlot value of a Plot axis is non-zero, it causes that axis to be mapped logarithmically, otherwise (the default) the axis is mapped linearly.

Type:
Integer (boolean).

Class Applicability:

Plot
All Plots have this attribute.

Notes:

- The setting of the LogPlot attribute provides the default value for the related LogTicks attribute. By selecting suitable values for LogPlot and LogTicks, it is possible to have tick marks which are evenly spaced in value but which are mapped logarithmically onto the screen (and vice-versa).
- An axis may only be mapped logarithmically if the visible part of the axis does not include the value zero. The visible part of the axis is that part which is mapped onto the plotting area, and is measured within the base Frame of the FrameSet which was supplied when the Plot was constructed. Any attempt to set LogPlot to a non-zero value will be ignored (without error) if the visible part of the axis includes the value zero
- If no axis is specified, (e.g. "LogPlot" instead of "LogPlot(2)"), then a "set" or "clear" operation will affect the attribute value of all the Plot axes, while a "get" or "test" operation will use just the LogPlot(1) value.

LogTicks(axis) Space the major tick marks logarithmically? LogTicks(axis)

Description: This attribute controls the appearance of an annotated coordinate grid (drawn with the astGrid function) by determining whether the major tick marks should be spaced logarithmically or linearly in axis value. It takes a separate value for each physical axis of the Plot so that, for instance, the setting "LogTicks(2)=1" specifies that the major tick marks on the second axis should be spaced logarithmically.

If the LogTicks value for a physical axis is non-zero, the major tick marks on that axis will be spaced logarithmically (that is, there will be a constant ratio between the axis values at adjacent major tick marks). An error will be reported if the dynamic range of the axis (the ratio of the largest to smallest displayed axis value) is less than 10.0. If the LogTicks value is zero, the major tick marks will be evenly spaced (that is, there will be a constant difference between the axis values at adjacent major tick marks). The default is to produce logarithmically spaced tick marks if the corresponding LogPlot attribute is non-zero and the ratio of maximum axis value to minimum axis value is 100 or more. If either of these conditions is not met, the default is to produce linearly spaced tick marks.

Type:

Integer (boolean).

Class Applicability:

Plot

All Plots have this attribute.

Notes:

- The setting of the LogTicks attribute does not affect the mapping of the plot onto the screen, which is controlled by attribute LogPlot. By selecting suitable values for LogPlot and LogTicks, it is possible to have tick marks which are evenly spaced in value but which are mapped logarithmically onto the screen (and vica-versa).
- An error will be reported when drawing an annotated axis grid if the visible part of the physical axis includes the value zero.
- If no axis is specified, (e.g. "LogTicks" instead of "LogTicks(2)"), then a "set" or "clear" operation will affect the attribute value of all the Plot axes, while a "get" or "test" operation will use just the LogTicks(1) value.

LonAxis Index of the longitude axis LonAxis

Description: This read-only attribute gives the index (1 or 2) of the longitude axis within the SkyFrame (taking into account any current axis permutations).

Type:

Integer.

Class Applicability:

SkyFrame

All SkyFrames have this attribute.

LutInterp Look-up table interpolation method LutInterp

Description: This attribute indicates the method to be used when finding the output value of a LutMap for an input value part way between two table entries. If it is set to 0 (the default) then linear interpolation is used. Otherwise, nearest neighbour interpolation is used.

Using nearest neighbour interpolation causes AST__BAD to be returned for any point which falls outside the bounds of the table. Linear interpolation results in an extrapolated value being returned based on the two end entries in the table.

Type:

Integer.

Class Applicability:

LutMap

All LutMaps have this attribute.

MajTickLen(axis) Length of major tick marks for a Plot MajTickLen(axis)

Description: This attribute controls the appearance of an annotated coordinate grid (drawn with the astGrid function) by determining the length of the major tick marks drawn on the axes of a Plot. It takes a separate value for each physical axis of the Plot so that, for instance, the setting "MajTickLen(2)=0" specifies the length of the major tick marks drawn on the second axis.

The MajTickLen value should be given as a fraction of the minimum dimension of the plotting area. Negative values cause major tick marks to be placed on the outside of the corresponding grid line or axis (but subject to any clipping imposed by the underlying graphics system), while positive values cause them to be placed on the inside.

The default behaviour depends on whether a coordinate grid is drawn inside the plotting area (see the Grid attribute). If so, the default MajTickLen value is zero (so that major ticks are not drawn), otherwise the default is +0.015.

Type:

Floating point.

Class Applicability:

Plot

All Plots have this attribute.

Notes:

- If no axis is specified, (e.g. "MajTickLen" instead of "MajTickLen(2)"), then a "set" or "clear" operation will affect the attribute value of all the Plot axes, while a "get" or "test" operation will use just the MajTickLen(1) value.

MapLocked Prevent new entries being added to a KeyMap? MapLocked

Description: If this boolean attribute is set to a non-zero value, an error will be reported if an attempt is made to add a new entry to the KeyMap. Note, the value associated with any existing entries can still be changed, but no new entries can be stored in the KeyMap. The default value (zero) allows new entries to be added to the KeyMap.

Type:
Integer (boolean).

Class Applicability:

KeyMap
All KeyMaps have this attribute.

Notes:

- When setting a new value for MapLocked, the supplied value is propagated to any KeyMaps contained within the supplied KeyMap.
- When clearing the MapLocked attribute, the attribute is also cleared in any KeyMaps contained within the supplied KeyMap.

MatchEnd Match trailing axes? MatchEnd

Description: This attribute is a boolean value which controls how a Frame behaves when it is used (by astFindFrame) as a template to match another (target) Frame. It applies only in the case where a match occurs between template and target Frames with different numbers of axes.

If the MatchEnd value of the template Frame is zero, then the axes which occur first in the target Frame will be matched and any trailing axes (in either the target or template) will be disregarded. If it is non-zero, the final axes in each Frame will be matched and any un-matched leading axes will be disregarded instead.

Type:
Integer (boolean).

Class Applicability:

Frame
The default MatchEnd value for a Frame is zero, so that trailing axes are disregarded.

FrameSet
The MatchEnd attribute of a FrameSet is the same as that of its current Frame (as specified by the Current attribute).

MaxAxes Maximum number of Frame axes to match MaxAxes

Description: This attribute controls how a Frame behaves when it is used (by astFindFrame) as a template to match another (target) Frame. It specifies the maximum number of axes that the target Frame may have in order to match the template.

Normally, this value will equal the number of Frame axes, so that a template Frame will only match another Frame with the same number of axes as itself. By setting a different value, however, the matching process may be used to identify Frames with specified numbers of axes.

Type:

Integer.

Class Applicability:

Frame

The default MaxAxes value for a Frame is equal to the number of Frame axes (Naxes attribute).

CmpFrame

The MaxAxes attribute of a CmpFrame defaults to a large number (1000000) which is much larger than any likely number of axes in a Frame. Combined with the MinAxes default of zero (for a CmpFrame), this means that the default behaviour for a CmpFrame is to match any target Frame that consists of a subset of the axes in the template CmpFrame. To change this so that a CmpFrame will only match Frames that have the same number of axes, you should set the CmpFrame MaxAxes and MinAxes attributes to the number of axes in the CmpFrame.

FrameSet

The MaxAxes attribute of a FrameSet is the same as that of its current Frame (as specified by the Current attribute).

Notes:

- When setting a MaxAxes value, the value of the MinAxes attribute may also be silently changed so that it remains consistent with (i.e. does not exceed) the new value. The default MaxAxes value may also be reduced to remain consistent with the MinAxes value.
- If a template Frame is used to match a target with a different number of axes, the MatchEnd attribute of the template is used to determine how the individual axes of each Frame should match.

MeshSize Number of points used to represent the boundary of a Region MeshSize

Description: This attribute controls how many points are used when creating a mesh of points covering the boundary or volume of a Region. Such a mesh is returned by the astGetRegionMesh method. The boundary mesh is also used when testing for overlap between two Regions: each point in the bomdary mesh of the first Region is checked to see if it is inside or outside the second Region. Thus, the reliability of the overlap check depends on the value assigned to this attribute. If the value used is very low, it is possible for overlaps to go unnoticed. High values produce more reliable results, but can result in the overlap test being very slow. The default value is 200 for two dimensional Regions and 2000 for three or more dimensional Regions (this attribute is not used for 1-dimensional regions since the boundary of a simple 1-d Region can only ever have two points). A value of five is used if the supplied value is less than five.

Type:
Integer.

Class Applicability:

Region
All Regions have this attribute.

CmpRegion
The default MeshSize for a CmpRegion is the MeshSize of its first component Region.

Stc
The default MeshSize for an Stc is the MeshSize of its encapsulated Region.

MinAxes Minimum number of Frame axes to match MinAxes

Description: This attribute controls how a Frame behaves when it is used (by astFindFrame) as a template to match another (target) Frame. It specifies the minimum number of axes that the target Frame may have in order to match the template.

Normally, this value will equal the number of Frame axes, so that a template Frame will only match another Frame with the same number of axes as itself. By setting a different value, however, the matching process may be used to identify Frames with specified numbers of axes.

Type:
Integer.

Class Applicability:

Frame
The default MinAxes value for a Frame is equal to the number of Frame axes (Naxes attribute).

CmpFrame
The MinAxes attribute of a CmpFrame defaults to zero. Combined with the MaxAxes default of 1000000 (for a CmpFrame), this means that the default behaviour for a CmpFrame is to match any target Frame that consists of a subset of the axes in the template CmpFrame. To change this so that a CmpFrame will only match Frames that have the same number of axes, you should set the CmpFrame MinAxes and MaxAxes attributes to the number of axes in the CmpFrame.

FrameSet
The MinAxes attribute of a FrameSet is the same as that of its current Frame (as specified by the Current attribute).

Notes:

- When setting a MinAxes value, the value of the MaxAxes attribute may also be silently changed so that it remains consistent with (i.e. is not less than) the new value. The default MinAxes value may also be reduced to remain consistent with the MaxAxes value.
- If a template Frame is used to match a target with a different number of axes, the MatchEnd attribute of the template is used to determine how the individual axes of each Frame should match.

MinTick(axis) Density of minor tick marks for a Plot MinTick(axis)

Description: This attribute controls the appearance of an annotated coordinate grid (drawn with the astGrid function) by determining the density of minor tick marks which appear between the major axis values of a Plot. It takes a separate value for each physical axis of a Plot so that, for instance, the setting "MinTick(2)=2" specifies the density of minor tick marks along the second axis.

The value supplied should be the number of minor divisions required between each pair of major axis values, this being one more than the number of minor tick marks to be drawn. By default, a value is chosen that depends on the gap between major axis values and the nature of the axis.

Type:
Integer.

Class Applicability:

Plot
All Plots have this attribute.

Notes:

- If no axis is specified, (e.g. "MinTick" instead of "MinTick(2)"), then a "set" or "clear" operation will affect the attribute value of all the Plot axes, while a "get" or "test" operation will use just the MinTick(1) value.

MinTickLen(axis) Length of minor tick marks for a Plot MinTickLen(axis)

Description: This attribute controls the appearance of an annotated coordinate grid (drawn with the astGrid function) by determining the length of the minor tick marks drawn on the axes of a Plot. It takes a separate value for each physical axis of the Plot so that, for instance, the setting "MinTickLen(2)=0" specifies the length of the minor tick marks drawn on the second axis.

The MinTickLen value should be given as a fraction of the minimum dimension of the plotting area. Negative values cause minor tick marks to be placed on the outside of the corresponding grid line or axis (but subject to any clipping imposed by the underlying graphics system), while positive values cause them to be placed on the inside.

The default value is +0.007.

Type:
Floating point.

Class Applicability:

Plot
All Plots have this attribute.

Notes:

- The number of minor tick marks drawn is determined by the Plot's MinTick(axis) attribute.
- If no axis is specified, (e.g. "MinTickLen" instead of "MinTickLen(2)"), then a "set" or "clear" operation will affect the attribute value of all the Plot axes, while a "get" or "test" operation will use just the MinTickLen(1) value.

NatLat Native latitude of the reference point of a FITS-WCS projection NatLat

Description: This attribute gives the latitude of the reference point of the FITS-WCS projection implemented by a WcsMap. The value is in radians in the "native spherical" coordinate system. This value is fixed for most projections, for instance it is PI/2 (90 degrees) for all zenithal projections. For some projections (e.g. the conics) the value is not fixed, but is specified by parameter one on the latitude axis.

FITS-WCS paper II introduces the concept of a "fiducial point" which is logical distinct from the projection reference point. It is easy to confuse the use of these two points. The fiducial point is the point which has celestial coordinates given by the CRVAL FITS keywords. The native spherical coordinates for this point default to the values of the NatLat and NatLon, but these defaults mey be over-ridden by values stored in the PVi_j keywords. Put another way, the CRVAL keywords will by default give the celestial coordinates of the projection reference point, but may refer to some other point if alternative native longitude and latitude values are provided through the PVi_j keywords.

The NatLat attribute is read-only.

Type:

Floating point, read-only.

Class Applicability:

WcsMap

All WcsMaps have this attribute.

Notes:

- A default value of AST__BAD is used if no latitude value is available.

NatLon Native longitude of the reference point of a FITS-WCS projection NatLon

Description: This attribute gives the longitude of the reference point of the FITS-WCS projection implemented by a WcsMap. The value is in radians in the "native spherical" coordinate system, and will usually be zero. See the description of attribute NatLat for further information.

The NatLon attribute is read-only.

Type:

Floating point, read-only.

Class Applicability:

WcsMap

All WcsMaps have this attribute.

Naxes Number of Frame axes Naxes

Description: This is a read-only attribute giving the number of axes in a Frame (i.e. the number of dimensions of the coordinate space which the Frame describes). This value is determined when the Frame is created.

Type:

Integer, read-only.

Class Applicability:

Frame
All Frames have this attribute.

FrameSet
The Naxes attribute of a FrameSet is the same as that of its current Frame (as specified by the Current attribute).

CmpFrame
The Naxes attribute of a CmpFrame is equal to the sum of the Naxes values of its two component Frames.

Ncard Number of FITS header cards in a FitsChan Ncard

Description: This attribute gives the total number of FITS header cards stored in a FitsChan. It is updated as cards are added or deleted.

Type:
Integer, read-only.

Class Applicability:

FitsChan
All FitsChans have this attribute.

Ncolumn The number of columns in the table Ncolumn

Description: This attribute holds the number of columns currently in the table. Columns are added and removed using the astAddColumn and astRemoveColumn functions.

Type:
Integer, read-only.

Class Applicability:

Table
All Tables have this attribute.

NegLon Display negative longitude values? NegLon

Description: This attribute is a boolean value which controls how longitude values are normalized for display by astNorm.

If the NegLon attribute is zero, then normalized longitude values will be in the range zero to 2.pi. If NegLon is non-zero, then normalized longitude values will be in the range -pi to pi.

The default value depends on the current value of the SkyRefIs attribute, If SkyRefIs has a value of "Origin", then the default for NegLon is one, otherwise the default is zero.

Type:
Integer (boolean).

Class Applicability:

SkyFrame
All SkyFrames have this attribute.

Negated Region negation flag Negated

Description: This attribute controls whether a Region represents the "inside" or the "outside" of the area which was supplied when the Region was created. If the attribute value is zero (the default), the Region represents the inside of the original area. However, if it is non-zero, it represents the outside of the original area. The value of this attribute may be toggled using the astNegate function.

Note, whether the boundary is considered to be inside the Region or not is controlled by the Closed attribute. Changing the value of the Negated attribute does not change the value of the Closed attribute. Thus, if Region is closed, then the boundary of the Region will be inside the Region, whatever the setting of the Negated attribute.

Type:

Integer (boolean).

Class Applicability:

Region

All Regions have this attribute.

Nframe Number of Frames in a FrameSet Nframe

Description: This attrbute gives the number of Frames in a FrameSet. This value will change as Frames are added or removed, but will always be at least one.

Type:

Integer, read-only.

Class Applicability:

FrameSet

All FrameSets have this attribute.

Nin Number of input coordinates for a Mapping Nin

Description: This attribute gives the number of coordinate values required to specify an input point for a Mapping (i.e. the number of dimensions of the space in which the Mapping's input points reside).

Type:

Integer, read-only.

Class Applicability:

Mapping

All Mappings have this attribute.

CmpMap

If a CmpMap's component Mappings are joined in series, then its Nin attribute is equal to the Nin attribute of the first component (or to the Nout attribute of the second component if the the CmpMap's Invert attribute is non-zero).

If a CmpMap's component Mappings are joined in parallel, then its Nin attribute is given by the sum of the Nin attributes of each component (or to the sum of their Nout attributes if the CmpMap's Invert attribute is non-zero).

Frame

The Nin attribute for a Frame is always equal to the number of Frame axes (Naxes attribute).

FrameSet
The Nin attribute of a FrameSet is equal to the number of axes (Naxes attribute) of its base Frame (as specified by the FrameSet's Base attribute). The Nin attribute value may therefore change if a new base Frame is selected.

NiterInverse Maximum number of iterations for the iterative inverse transformation NiterInverse

Description: This attribute controls the iterative inverse transformation used if the IterInverse attribute is non-zero.

Its value gives the maximum number of iterations of the Newton-Raphson algorithm to be used for each transformed position. The default value is 4. See also attribute TolInverse.

Type:
Integer.

Class Applicability:

PolyMap
All PolyMaps have this attribute.

Nkey Number of unique FITS keywords in a FitsChan Nkey

Description: This attribute gives the total number of unique FITS keywords stored in a FitsChan. It is updated as cards are added or deleted. If no keyword occurrs more than once in the FitsChan, the Ncard and Nkey attributes will be equal. If any keyword occurrs more than once, the Nkey attribute value will be smaller than the Ncard attribute value.

Type:
Integer, read-only.

Class Applicability:

FitsChan
All FitsChans have this attribute.

Nobject Number of Objects in class Nobject

Description: This attribute gives the total number of Objects currently in existence in the same class as the Object whose attribute value is requested. This count does not include Objects which belong to derived (more specialised) classes.

This attribute is mainly intended for debugging. It can be used to detect whether Objects which should have been deleted have, in fact, been deleted.

Type:
Integer, read-only.

Class Applicability:

Object
All Objects have this attribute.

Norm(axis) Specifies the plane upon which a Plot3D draws text and markers Norm(axis)

Description: This attribute controls the appearance of text and markers drawn by a Plot3D. It specifies the orientation of the plane upon which text and markers will be drawn by all subsequent invocations of the astText and astMark functions.

When setting or getting the Norm attribute, the attribute name must be qualified by an axis index in the range 1 to 3. The 3 elements of the Norm attribute are together interpreted as a vector in 3D graphics coordinates that is normal to the plane upon which text and marks should be drawn. When testing or clearing the attribute, the axis index is optional. If no index is supplied, then clearing the Norm attribute will clear all three elements, and testing the Norm attribute will return a non-zero value if any of the three elements are set.

The default value is 1.0 for each of the 3 elements. The length of the vector is insignificant, but an error will be reported when attempting to draw text or markers if the vector has zero length.

Type:

Floating point.

Class Applicability:

Plot

All Plot3Ds have this attribute.

NormUnit(axis) Normalised Axis physical units NormUnit(axis)

Description: The value of this read-only attribute is derived from the current value of the Unit attribute. It will represent an equivalent system of units to the Unit attribute, but will potentially be simplified. For instance, if Unit is set to "s*(m/s)", the NormUnit value will be "m". If no simplification can be performed, the value of the NormUnit attribute will equal that of the Unit attribute.

Type:

String, read-only.

Class Applicability:

Frame

All Frames have this attribute.

Notes:

- When specifying this attribute by name, it should be subscripted with the number of the Frame axis to which it applies.

Nout Number of output coordinates for a Mapping Nout

Description: This attribute gives the number of coordinate values generated by a Mapping to specify each output point (i.e. the number of dimensions of the space in which the Mapping's output points reside).

Type:

Integer, read-only.

Class Applicability:

Mapping
All Mappings have this attribute.

CmpMap
If a CmpMap's component Mappings are joined in series, then its Nout attribute is equal to the Nout attribute of the second component (or to the Nin attribute of the first component if the the CmpMap's Invert attribute is non-zero).

If a CmpMap's component Mappings are joined in parallel, then its Nout attribute is given by the sum of the Nout attributes of each component (or to the sum of their Nin attributes if the CmpMap's Invert attribute is non-zero).

Frame
The Nout attribute for a Frame is always equal to the number of Frame axes (Naxes attribute).

FrameSet
The Nout attribute of a FrameSet is equal to the number of FrameSet axes (Naxes attribute) which, in turn, is equal to the Naxes attribute of the FrameSet's current Frame (as specified by the Current attribute). The Nout attribute value may therefore change if a new current Frame is selected.

Nparameter The number of global parameters in the table Nparameter

Description: This attribute holds the number of global parameters currently in the table. Parameters are added and removed using the astAddParameter and astRemoveParameter functions.

Type:
Integer, read-only.

Class Applicability:

Table
All Tables have this attribute.

Nrow The number of rows in the table Nrow

Description: This attribute holds the index of the last row to which any contents have been added using any of the astMapPut... AST_MAPPUT... functions. The first row has index 1.

Type:
Integer, read-only.

Class Applicability:

Table
All Tables have this attribute.

NumLab(axis) Draw numerical axis labels for a Plot? NumLab(axis)

Description: This attribute controls the appearance of an annotated coordinate grid (drawn with the astGrid function) by determining whether labels should be drawn to represent the numerical values along each axis of a Plot. It takes a separate value for each physical axis of a Plot so that, for instance, the setting "NumLab(2)=1" specifies that numerical labels should be drawn for the second axis.

If the NumLab value of a Plot axis is non-zero (the default), then numerical labels will be drawn for that axis, otherwise they will be omitted.

Type:

Integer (boolean).

Class Applicability:

Plot

All Plots have this attribute.

Notes:

- The drawing of associated descriptive axis labels for a Plot (describing the quantity being plotted along each axis) is controlled by the TextLab(axis) attribute.
- If no axis is specified, (e.g. "NumLab" instead of "NumLab(2)"), then a "set" or "clear" operation will affect the attribute value of all the Plot axes, while a "get" or "test" operation will use just the NumLab(1) value.

NumLabGap(axis) Spacing of numerical labels for a Plot NumLabGap(axis)

Description: This attribute controls the appearance of an annotated coordinate grid (drawn with the astGrid function) by determining where numerical axis labels are placed relative to the axes they describe. It takes a separate value for each physical axis of a Plot so that, for instance, the setting "NumLabGap(2)=-0.01" specifies where the numerical label for the second axis should be drawn.

For each axis, the NumLabGap value gives the spacing between the axis line (or edge of the plotting area, if appropriate) and the nearest edge of the corresponding numerical axis labels. Positive values cause the descriptive label to be placed on the opposite side of the line to the default tick marks, while negative values cause it to be placed on the same side.

The NumLabGap value should be given as a fraction of the minimum dimension of the plotting area, the default value being +0.01.

Type:

Floating point.

Class Applicability:

Plot

All Plots have this attribute.

Notes:

- If no axis is specified, (e.g. "NumLabGap" instead of "NumLabGap(2)"), then a "set" or "clear" operation will affect the attribute value of all the Plot axes, while a "get" or "test" operation will use just the NumLabGap(1) value.

ObjSize The in-memory size of the Object ObjSize

Description: This attribute gives the total number of bytes of memory used by the Object. This includes any Objects which are encapsulated within the supplied Object.

Type:

Integer, read-only.

Class Applicability:

Object

All Objects have this attribute.

ObsAlt The geodetic altitude of the observer **ObsAlt**

Description: This attribute specifies the geodetic altitude of the observer, in metres, relative to the IAU 1976 reference ellipsoid. The basic Frame class makes no use of this attribute, but specialised subclasses of Frame may use it. For instance, the SpecFrame, SkyFrame and TimeFrame classes use it. The default value is zero.

Type:
String.

Class Applicability:

Frame
All Frames have this attribute.

SpecFrame
Together with the ObsLon, Epoch, RefRA and RefDec attributes, it defines the Doppler shift introduced by the observers diurnal motion around the earths axis, which is needed when converting to or from the topocentric standard of rest. The maximum velocity error which can be caused by an incorrect value is 0.5 km/s. The default value for the attribute is zero.

TimeFrame
Together with the ObsLon attribute, it is used when converting between certain time scales (TDB, TCB, LMST, LAST)

ObsLat The geodetic latitude of the observer **ObsLat**

Description: This attribute specifies the geodetic latitude of the observer, in degrees, relative to the IAU 1976 reference ellipsoid. The basic Frame class makes no use of this attribute, but specialised subclasses of Frame may use it. For instance, the SpecFrame, SkyFrame and TimeFrame classes use it. The default value is zero.

The value is stored internally in radians, but is converted to and from a degrees string for access. Some example input formats are: "22:19:23.2", "22 19 23.2", "22:19.387", "22.32311", "N22.32311", "-45.6", "S45.6". As indicated, the sign of the latitude can optionally be indicated using characters "N" and "S" in place of the usual "+" and "-". When converting the stored value to a string, the format "[s]dd:mm:ss.ss" is used, when "[s]" is "N" or "S".

Type:
String.

Class Applicability:

Frame
All Frames have this attribute.

SpecFrame
Together with the ObsLon, Epoch, RefRA and RefDec attributes, it defines the Doppler shift introduced by the observers diurnal motion around the earths axis, which is needed when converting to or from the topocentric standard of rest. The maximum velocity error which can be caused by an incorrect value is 0.5 km/s. The default value for the attribute is zero.

TimeFrame
Together with the ObsLon attribute, it is used when converting between certain time scales (TDB, TCB, LMST, LAST)

ObsLon The geodetic longitude of the observer ObsLon

Description: This attribute specifies the geodetic (or equivalently, geocentric) longitude of the observer, in degrees, measured positive eastwards. See also attribute ObsLat. The basic Frame class makes no use of this attribute, but specialised subclasses of Frame may use it. For instance, the SpecFrame, SkyFrame and TimeFrame classes use it. The default value is zero.

The value is stored internally in radians, but is converted to and from a degrees string for access. Some example input formats are: "155:19:23.2", "155 19 23.2", "155:19.387", "155.32311", "E155.32311", "-204.67689", "W204.67689". As indicated, the sign of the longitude can optionally be indicated using characters "E" and "W" in place of the usual "+" and "-". When converting the stored value to a string, the format "[s]ddd:mm:ss.ss" is used, when "[s]" is "E" or "W" and the numerical value is chosen to be less than 180 degrees.

Type:

String.

Class Applicability:

Frame

All Frames have this attribute.

SpecFrame

Together with the ObsLon, Epoch, RefRA and RefDec attributes, it defines the Doppler shift introduced by the observers diurnal motion around the earths axis, which is needed when converting to or from the topocentric standard of rest. The maximum velocity error which can be caused by an incorrect value is 0.5 km/s. The default value for the attribute is zero.

TimeFrame

Together with the ObsLon attribute, it is used when converting between certain time scales (TDB, TCB, LMST, LAST)

PVMax(i) Maximum number of FITS-WCS projection parameters PVMax(i)

Description: This attribute specifies the largest legal index for a PV projection parameter attached to a specified axis of the WcsMap (i.e. the largest legal value for "m" when accessing the "PVi_m" attribute). The axis index is specified by i, and should be in the range 1 to 99. The value for each axis is determined by the projection type specified when the WcsMap is first created using astWcsMap and cannot subsequently be changed.

Type:

Integer, read-only.

Class Applicability:

WcsMap

All WcsMaps have this attribute.

PVi_m FITS-WCS projection parameters PVi_m

Description: This attribute specifies the projection parameter values to be used by a WcsMap when implementing a FITS-WCS sky projection. Each PV attribute name should include two integers, i and m, separated by an underscore. The axis index is specified by i, and should be in the range 1 to 99. The parameter number is specified by m, and should be in the range 0 to 99. For example, "PV2_1=45.0" would specify a value for projection parameter 1 of axis 2 in a WcsMap.

These projection parameters correspond exactly to the values stored using the FITS-WCS keywords "PV1_1", "PV1_2", etc. This means that projection parameters which correspond to angles must be given in degrees (despite the fact that the angular coordinates and other attributes used by a WcsMap are in radians).

The set of projection parameters used by a WcsMap depends on the type of projection, which is determined by its WcsType parameter. Most projections either do not require projection parameters, or use parameters 1 and 2 associated with the latitude axis. You should consult the FITS-WCS paper for details.

Some projection parameters have default values (as defined in the FITS-WCS paper) which apply if no explicit value is given. You may omit setting a value for these "optional" parameters and the default will apply. Some projection parameters, however, have no default and a value must be explicitly supplied. This is most conveniently done using the "options" argument of astWcsMap (q.v.) when a WcsMap is first created. An error will result when a WcsMap is used to transform coordinates if any of its required projection parameters has not been set and lacks a default value.

A "get" operation for a parameter which has not been assigned a value will return the default value defined in the FITS-WCS paper, or AST__BAD if the paper indicates that the parameter has no default. A default value of zero is returned for parameters which are not accessed by the projection.

Note, the FITS-WCS paper reserves parameters 1 and 2 on the longitude axis to hold the native longitude and latitude of the fiducial point of the projection, in degrees. The default values for these parameters are determined by the projection type. The AST-specific TPN projection does not use this convention - all projection parameters for both axes are used to represent polynomical correction terms, and the native longitude and latitude at the fiducial point may not be changed from the default values of zero and 90 degrees.

Type:

Floating point.

Class Applicability:

WcsMap

All WcsMaps have this attribute.

Notes:

- If the projection parameter values given for a WcsMap do not satisfy all the required constraints (as defined in the FITS-WCS paper), then an error will result when the WcsMap is used to transform coordinates.

PcdCen(axis) Centre coordinates of pincushion/barrel distortion PcdCen(axis)

Description: This attribute specifies the centre of the pincushion/barrel distortion implemented by a PcdMap. It takes a separate value for each axis of the PcdMap so that, for instance, the settings "PcdCen(1)=345.0,PcdCen(2)=-104.4" specify that the pincushion distortion is centred at positions of 345.0 and -104.4 on axes 1 and 2 respectively. This attribute is set when a PcdMap is created, but may later be modified. If the attribute is cleared, the default value for both axes is zero.

Type:

Floating point.

Class Applicability:

PcdMap

All PcdMaps have this attribute.

Notes:

- If no axis is specified, (e.g. "PcdCen" instead of "PcdCen(2)"), then a "set" or "clear" operation will affect the attribute value of both axes, while a "get" or "test" operation will use just the PcdCen(1) value.

Permute Permute axis order? Permute

Description: This attribute is a boolean value which controls how a Frame behaves when it is used (by astFindFrame) as a template to match another (target) Frame. It specifies whether the axis order of the target Frame may be permuted in order to obtain a match.

If the template's Permute value is zero, it will match a target only if it can do so without changing the order of its axes. Otherwise, it will attempt to permute the target's axes as necessary.

The default value is 1, so that axis permutation will be attempted.

Type:
String.

Class Applicability:

Frame
All Frames have this attribute. However, the Frame class effectively ignores this attribute and behaves as if it has the value 1. This is because the axes of a basic Frame are not distinguishable and will always match any other Frame whatever their order.

SkyFrame
Unlike a basic Frame, the SkyFrame class makes use of this attribute.

FrameSet
The Permute attribute of a FrameSet is the same as that of its current Frame (as specified by the Current attribute).

PolarLong The longitude value to assign to either pole PolarLong

Description: This attribute holds the longitude value, in radians, to be returned when a Cartesian position corresponding to either the north or south pole is transformed into spherical coordinates. The default value is zero.

Type:
Double precision.

Class Applicability:

SphMap
All SphMaps have this attribute.

PolyTan Use PVi_m keywords to define distorted TAN projection? PolyTan

Description: This attribute is a boolean value which specifies how FITS "TAN" projections should be treated when reading a FrameSet from a foreign encoded FITS header. If zero, the projection is assumed to conform to the published FITS-WCS standard. If positive, the convention for a distorted TAN projection included in an early draft version of FITS-WCS paper II are assumed. In this convention the coefficients of a polynomial distortion to be applied to intermediate world coordinates are specified by the PVi_m keywords. This convention was removed from the paper before publication and so does not form part of the standard. Indeed, it is incompatible with the published standard because it re-defines the meaning of the first five PVi_m keywords on the longitude axis, which are reserved by the published standard for other purposes. However, headers that use this convention are still to be found, for instance the SCAMP utility (http://www.astromatic.net/software/scamp) creates them.

The default value for the PolyTan attribute is -1. A negative values causes the used convention to depend on the contents of the FitsChan. If the FitsChan contains any PVi_m keywords for the latitude axis, or if it contains PVi_m keywords for the longitude axis with "m" greater than 4, then the distorted TAN convention is used. Otherwise, the standard convention is used.

Type:

Integer.

Class Applicability:

FitsChan

All FitsChans have this attribute.

PreserveAxes Preserve axes? PreserveAxes

Description: This attribute controls how a Frame behaves when it is used (by astFindFrame) as a template to match another (target) Frame. It determines which axes appear (and in what order) in the "result" Frame produced.

If PreserveAxes is zero in the template Frame, then the result Frame will have the same number (and order) of axes as the template. If it is non-zero, however, the axes of the target Frame will be preserved, so that the result Frame will have the same number (and order) of axes as the target.

The default value is zero, so that target axes are not preserved and the result Frame resembles the template.

Type:

Integer (boolean).

Class Applicability:

Frame

All Frames have this attribute.

FrameSet

The PreserveAxes attribute of a FrameSet is the same as that of its current Frame (as specified by the Current attribute).

ProjP(m) FITS-WCS projection parameters ProjP(m)

Description: This attribute provides aliases for the PV attributes, which specifies the projection parameter values to be used by a WcsMap when implementing a FITS-WCS sky projection. ProjP is retained for compatibility with previous versions of FITS-WCS and AST. New applications should use the PV attibute instead.

Attributes ProjP(0) to ProjP(9) correspond to attributes PV<axlat>_0 to PV<axlat>_9, where <axlat> is replaced by the index of the latitude axis (given by attribute WcsAxis(2)). See PV for further details.

Type:

Floating point.

Class Applicability:

WcsMap

All WcsMaps have this attribute.

Projection Sky projection description Projection

Description: This attribute provides a place to store a description of the type of sky projection used when a SkyFrame is attached to a 2-dimensional object, such as an image or plotting surface. For example, typical values might be "orthographic", "Hammer-Aitoff" or "cylindrical equal area".

The Projection value is purely descriptive and does not affect the celestial coordinate system represented by the SkyFrame in any way. If it is set to a non-blank string, the description provided may be used when forming the default value for the SkyFrame's Title attribute (so that typically it will appear in graphical output, for instance). The default value is an empty string.

Type:

String.

Class Applicability:

SkyFrame

All SkyFrames have this attribute.

RefCount Count of active Object pointers RefCount

Description: This attribute gives the number of active pointers associated with an Object. It is modified whenever pointers are created or annulled (by astClone, astAnnul or astEnd for example). The count includes the initial pointer issued when the Object was created.

If the reference count for an Object falls to zero as the result of annulling a pointer to it, then the Object will be deleted.

Type:

Integer, read-only.

Class Applicability:

Object

All Objects have this attribute.

RefDec The declination of the reference point RefDec

Description: This attribute specifies the FK5 J2000.0 declination of a reference point on the sky. See the description of attribute RefRA for details. The default RefDec is "0:0:0".

Type:

String.

Class Applicability:

SpecFrame

All SpecFrames have this attribute.

RefRA The right ascension of the reference point RefRA

Description: This attribute, together with the RefDec attribute, specifies the FK5 J2000.0 coordinates of a reference point on the sky. For 1-dimensional spectra, this should normally be the position of the source. For spectral data with spatial coverage (spectral cubes, etc), this should be close to centre of the spatial coverage. It is used to define the correction for Doppler shift to be applied when using the astFindFrame or astConvert method to convert between different standards of rest.

The SpecFrame class assumes this velocity correction is spatially invariant. If a single SpecFrame is used (for instance, as a component of a CmpFrame) to describe spectral values at different points on the sky, then it is assumes that the doppler shift at any spatial position is the same as at the reference position. The maximum velocity error introduced by this assumption is of the order of V*SIN(FOV), where FOV is the angular field of view, and V is the relative velocity of the two standards of rest. As an example, when correcting from the observers rest frame (i.e. the topocentric rest frame) to the kinematic local standard of rest the maximum value of V is about 20 km/s, so for 5 arc-minute field of view the maximum velocity error introduced by the correction will be about 0.03 km/s. As another example, the maximum error when correcting from the observers rest frame to the local group is about 5 km/s over a 1 degree field of view.

The RefRA and RefDec attributes are stored internally in radians, but are converted to and from a string for access. The format "hh:mm:ss.ss" is used for RefRA, and "dd:mm:ss.s" is used for RefDec. The methods astSetRefPos and astGetRefPos may be used to access the values of these attributes directly as unformatted values in radians.

The default for RefRA is "0:0:0".

Type:

String.

Class Applicability:

SpecFrame

All SpecFrames have this attribute.

RegionClass The AST class name of the Region encapsulated within an Stc RegionClass

Description: This is a read-only attribute giving the AST class name of the Region encapsulated within an Stc (that is, the class of the Region which was supplied when the Stc was created).

Type:

String, read-only.

Class Applicability:

Stc

All Stc objects this attribute.

Report Report transformed coordinates? Report

Description: This attribute controls whether coordinate values are reported whenever a Mapping is used to transform a set of points. If its value is zero (the default), no report is made. However, if it is non-zero, the coordinates of each point are reported (both before and after transformation) by writing them to standard output.

This attribute is provided as an aid to debugging, and to avoid having to report values explicitly in simple programs.

Type:
Integer (boolean).

Class Applicability:

Mapping
All Mappings have this attribute.

CmpMap
When applied to a compound Mapping (CmpMap), only the Report attribute of the CmpMap, and not those of its component Mappings, is used. Coordinate information is never reported for the component Mappings individually, only for the complete CmpMap.

Frame
When applied to any Frame, the formatting capabilities of the Frame (as provided by the astFormat function) will be used to format the reported coordinates.

FrameSet
When applied to any FrameSet, the formatting capabilities of the base and current Frames will be used (as above) to individually format the input and output coordinates, as appropriate. The Report attribute of a FrameSet is not itself affected by selecting a new base or current Frame, but the resulting formatting capabilities may be.

Notes:

- Unlike most other attributes, the value of the Report attribute is not transferred when a Mapping is copied. Instead, its value is undefined (and therefore defaults to zero) in any copy. Similarly, it becomes undefined in any external representation of a Mapping produced by the astWrite function.

ReportLevel — Determines which read/write conditions are reported — ReportLevel

Description: This attribute determines which, if any, of the conditions that occur whilst reading or writing an Object should be reported. These conditions will generate either a fatal error or a warning, as determined by attribute Strict. ReportLevel can take any of the following values:

0 - Do not report any conditions.

1 - Report only conditions where significant information content has been changed. For instance, an unsupported time-scale has been replaced by a supported near-equivalent time-scale. Another example is if a basic Channel unexpected encounters data items that may have been introduced by later versions of AST.

2 - Report the above, and in addition report significant default values. For instance, if no time-scale was specified when reading an Object from an external data source, report the default time-scale that is being used.

3 - Report the above, and in addition report any other potentially interesting conditions that have no significant effect on the conversion. For instance, report if a time-scale of "TT" (terrestrial time) is used in place of "ET" (ephemeris time). This change has no signficiant effect because ET is the predecessor of, and is continuous with, TT. Synonyms such as "IAT" and "TAI" are another example.

The default value is 1. Note, there are many other conditions that can occur whilst reading or writing an Object that completely prevent the conversion taking place. Such conditions will always generate errors, irrespective of the ReportLevel and Strict attributes.

Type:
Integer (boolean).

Class Applicability:

Channel
All Channels have this attribute.

FitsChan
All the conditions selected by the FitsChan Warnings attribute are reported at level 1.

RestFreq The rest frequency RestFreq

Description: This attribute specifies the frequency corresponding to zero velocity. It is used when converting between between velocity-based coordinate systems and and other coordinate systems (such as frequency, wavelength, energy, etc). The default value is 1.0E5 GHz.

When setting a new value for this attribute, the new value can be supplied either directly as a frequency, or indirectly as a wavelength or energy, in which case the supplied value is converted to a frequency before being stored. The nature of the supplied value is indicated by appending text to the end of the numerical value indicating the units in which the value is supplied. If the units are not specified, then the supplied value is assumed to be a frequency in units of GHz. If the supplied unit is a unit of frequency, the supplied value is assumed to be a frequency in the given units. If the supplied unit is a unit of length, the supplied value is assumed to be a (vacuum) wavelength. If the supplied unit is a unit of energy, the supplied value is assumed to be an energy. For instance, the following strings all result in a rest frequency of around 1.4E14 Hz being used: "1.4E5", "1.4E14 Hz", "1.4E14 s**-1", "1.4E5 GHz", "2.14E-6 m", "21400 Angstrom", "9.28E-20 J", "9.28E-13 erg", "0.58 eV", etc.

When getting the value of this attribute, the returned value is always a frequency in units of GHz.

Type:
Floating point.

Class Applicability:

SpecFrame
All SpecFrames have this attribute.

RootCorner Specifies which edges of the 3D box should be annotated RootCorner

Description: This attribute controls the appearance of an annotated coordinate grid (drawn with the astGrid function) by determining which edges of the cube enclosing the 3D graphics space are used for displaying numerical and descriptive axis labels. The attribute value identifies one of the eight corners of the cube within which graphics are being drawn (i.e. the cube specified by the "graphbox" parameter when astPlot3D was called tp create the Plot3D). Axis labels and tick marks will be placed on the three cube edges that meet at the given corner.

The attribute value should consist of three character, each of which must be either "U" or "L". The first character in the string specifies the position of the corner on the first graphics axis. If the character is "U" then the corner is at the upper bound on the first graphics axis. If it is "L", then the corner is at the lower bound on the first axis. Likewise, the second and third characters in the string specify the location of the corner on the second and third graphics axes.

For instance, corner "LLL" is the corner that is at the lower bound on all three graphics axes, and corner "ULU" is at the upper bound on axes 1 and 3 but at the lower bound on axis 2.

The default value is "LLL".

Type:
String.

Class Applicability:

Plot3D
All Plot3Ds have this attribute.

Seed Random number seed for a MathMap Seed

Description: This attribute, which may take any integer value, determines the sequence of random numbers produced by the random number functions in MathMap expressions. It is set to an unpredictable default value when a MathMap is created, so that by default each MathMap uses a different set of random numbers.

If required, you may set this Seed attribute to a value of your choosing in order to produce repeatable behaviour from the random number functions. You may also enquire the Seed value (e.g. if an initially unpredictable value has been used) and then use it to reproduce the resulting sequence of random numbers, either from the same MathMap or from another one.

Clearing the Seed attribute gives it a new unpredictable default value.

Type:
Integer.

Class Applicability:

MathMap
All MathMaps have this attribute.

SideBand Indicates which sideband a dual sideband spectrum represents SideBand

Description: This attribute indicates whether the DSBSpecFrame currently represents its lower or upper sideband, or an offset from the local oscillator frequency. When querying the current value, the returned string is always one of "usb" (for upper sideband), "lsb" (for lower sideband), or "lo" (for offset from the local oscillator frequency). When setting a new value, any of the strings "lsb", "usb", "observed", "image" or "lo" may be supplied (case insensitive). The "observed" sideband is which ever sideband (upper or lower) contains the central spectral position given by attribute DSBCentre, and the "image" sideband is the other sideband. It is the sign of the IF attribute which determines if the observed sideband is the upper or lower sideband. The default value for SideBand is the observed sideband.

Type:
String.

Class Applicability:

DSBSpecFrame
All DSBSpecFrames have this attribute.

SimpFI Forward-inverse MathMap pairs simplify? SimpFI

Description: This attribute should be set to a non-zero value if applying a MathMap's forward transformation, followed immediately by the matching inverse transformation will always restore the original set of coordinates. It indicates that AST may replace such a sequence of operations by an identity Mapping (a UnitMap) if it is encountered while simplifying a compound Mapping (e.g. using astSimplify).

By default, the SimpFI attribute is zero, so that AST will not perform this simplification unless you have set SimpFI to indicate that it is safe to do so.

Type:

Integer (boolean).

Class Applicability:

MathMap

All MathMaps have this attribute.

Notes:

- For simplification to occur, the two MathMaps must be in series and be identical (with textually identical transformation functions). Functional equivalence is not sufficient.
- The consent of both MathMaps is required before simplification can take place. If either has a SimpFI value of zero, then simplification will not occur.
- The SimpFI attribute controls simplification only in the case where a MathMap's forward transformation is followed by the matching inverse transformation. It does not apply if an inverse transformation is followed by a forward transformation. This latter case is controlled by the SimpIF attribute.
- The `"forward"` and `"inverse"` transformations referred to are those defined when the MathMap is created (corresponding to the `"fwd"` and `"inv"` parameters of its constructor function). If the MathMap is inverted (i.e. its Invert attribute is non-zero), then the role of the SimpFI and SimpIF attributes will be interchanged.

SimpIF Inverse-forward MathMap pairs simplify? SimpIF

Description: This attribute should be set to a non-zero value if applying a MathMap's inverse transformation, followed immediately by the matching forward transformation will always restore the original set of coordinates. It indicates that AST may replace such a sequence of operations by an identity Mapping (a UnitMap) if it is encountered while simplifying a compound Mapping (e.g. using astSimplify).

By default, the SimpIF attribute is zero, so that AST will not perform this simplification unless you have set SimpIF to indicate that it is safe to do so.

Type:

Integer (boolean).

Class Applicability:

MathMap

All MathMaps have this attribute.

Notes:

- For simplification to occur, the two MathMaps must be in series and be identical (with textually identical transformation functions). Functional equivalence is not sufficient.
- The consent of both MathMaps is required before simplification can take place. If either has a SimpIF value of zero, then simplification will not occur.
- The SimpIF attribute controls simplification only in the case where a MathMap's inverse transformation is followed by the matching forward transformation. It does not apply if a forward transformation is followed by an inverse transformation. This latter case is controlled by the SimpFI attribute.
- The `"forward"` and `"inverse"` transformations referred to are those defined when the MathMap is created (corresponding to the `"fwd"` and `"inv"` parameters of its constructor function). If the MathMap is inverted (i.e. its Invert attribute is non-zero), then the role of the SimpFI and SimpIF attributes will be interchanged.

SinkFile Output file to which to data should be written SinkFile

Description: This attribute specifies the name of a file to which the Channel should write data. If specified it is used in preference to any sink function specified when the Channel was created.

Assigning a new value to this attribute will cause any previously opened SinkFile to be closed. The first subsequent call to astWrite will attempt to open the new file (an error will be reported if the file cannot be opened), and write data to it. All subsequent call to astWrite will write data to the new file, until the SinkFile attribute is cleared or changed.

Clearing the attribute causes any open SinkFile to be closed. All subsequent data writes will use the sink function specified when the Channel was created, or will write to standard output if no sink function was specified.

If no value has been assigned to SinkFile, a null string will be returned if an attempt is made to get the attribute value.

Type:

String.

Class Applicability:

FitsChan

When the FitsChan is destroyed, any headers in the FitsChan will be written out to the sink file, if one is specified (if not, the sink function used when the FitsChan was created is used). The sink file is a text file (not a FITS file) containing one header per line.

Notes:

- A new SinkFile will over-write any existing file with the same name unless the existing file is write protected, in which case an error will be reported.
- Any open SinkFile is closed when the Channel is deleted.
- If the Channel is copied or dumped (using astCopy or astShow) the SinkFile attribute is left in a cleared state in the output Channel (i.e. the value of the SinkFile attribute is not copied).

Size(element) Character size for a Plot element Size(element)

Description: This attribute determines the character size used when drawing each element of graphical output produced by a Plot. It takes a separate value for each graphical element so that, for instance, the setting "Size(title)=2.0" causes the Plot title to be drawn using twice the default character size.

The range of character sizes available and the appearance of the resulting text is determined by the underlying graphics system. The default behaviour is for all graphical elements to be drawn using the default character size supplied by this graphics system.

Type:

Floating Point.

Class Applicability:

Plot

All Plots have this attribute.

Notes:

- For a list of the graphical elements available, see the description of the Plot class.

- If no graphical element is specified, (e.g. "Size" instead of "Size(title)"), then a "set" or "clear" operation will affect the attribute value of all graphical elements, while a "get" or "test" operation will use just the Size(TextLab) value.

SizeGuess The expected size of the KeyMap SizeGuess

Description: This is attribute gives an estimate of the number of entries that will be stored in the KeyMap. It is used to tune the internal properties of the KeyMap for speed and efficiency. A larger value will result in faster access at the expense of increased memory requirements. However if the SizeGuess value is much larger than the actual size of the KeyMap, then there will be little, if any, speed gained by making the SizeGuess even larger. The default value is 300.

The value of this attribute can only be changed if the KeyMap is empty. Its value can be set conveniently when creating the KeyMap. An error will be reported if an attempt is made to set or clear the attribute when the KeyMap contains any entries.

Type:

Integer.

Class Applicability:

KeyMap

All KeyMaps have this attribute.

Skip Skip irrelevant data? Skip

Description: This is a boolean attribute which indicates whether the Object data being read through a Channel are inter-mixed with other, irrelevant, external data.

If Skip is zero (the default), then the source of input data is expected to contain descriptions of AST Objects and comments and nothing else (if anything else is read, an error will result). If Skip is non-zero, then any non-Object data encountered between Objects will be ignored and simply skipped over in order to reach the next Object.

Type:

Integer (boolean).

Class Applicability:

Channel

All Channels have this attribute.

FitsChan

The FitsChan class sets the default value of this attribute to 1, so that all irrelevant FITS headers will normally be ignored.

SkyRef(axis) Position defining the offset coordinate system SkyRef(axis)

Description: This attribute allows a SkyFrame to represent offsets, rather than absolute axis values, within the coordinate system specified by the System attribute. If supplied, SkyRef should be set to hold the longitude and latitude of a point within the coordinate system specified by the System attribute. The coordinate system represented by the SkyFrame will then be rotated in order to put the specified position at either the pole or the origin of the new coordinate system (as indicated by the SkyRefIs attribute). The orientation of the modified coordinate system is then controlled using the SkyRefP attribute.

If an integer axis index is included in the attribute name (e.g. "SkyRef(1)") then the attribute value should be supplied as a single floating point axis value, in radians, when setting a value for the attribute, and will be returned in the same form when getting the value of the attribute. In this case the integer axis index should be "1" or "2" (the values to use for longitude and latitude axes are given by the LonAxis and LatAxis attributes).

If no axis index is included in the attribute name (e.g. "SkyRef") then the attribute value should be supplied as a character string containing two formatted axis values (an axis 1 value followed by a comma, followed by an axis 2 value). The same form will be used when getting the value of the attribute.

The default values for SkyRef are zero longitude and zero latitude.

Type:

Floating point.

Class Applicability:

SkyFrame

All SkyFrames have this attribute.

Notes:

- If the System attribute of the SkyFrame is changed, any position given for SkyRef is transformed into the new System.
- If a value has been assigned to SkyRef attribute, then the default values for certain attributes are changed as follows: the default axis Labels for the SkyFrame are modified by appending " offset" to the end, the default axis Symbols for the SkyFrame are modified by prepending the character "D" to the start, and the default title is modified by replacing the projection information by the origin information.

Aligning SkyFrames with Offset Coordinate Systems:

The offset coordinate system within a SkyFrame should normally be considered as a superficial "re-badging" of the axes of the coordinate system specified by the System attribute - it merely provides an alternative numerical "label" for each position in the System coordinate system. The SkyFrame retains full knowledge of the celestial coordinate system on which the offset coordinate system is based (given by the System attribute). For instance, the SkyFrame retains knowledge of the way that one celestial coordinate system may "drift" with respect to another over time. Normally, if you attempt to align two SkyFrames (e.g. using the astConvert or astFindFrame routine), the effect of any offset coordinate system defined in either SkyFrame will be removed, resulting in alignment being performed in the celestial coordinate system given by the AlignSystem attribute. However, by setting the AlignOffset attribute ot a non-zero value, it is possible to change this behaviour so that the effect of the offset coordinate system is not removed when aligning two SkyFrames.

SkyRefIs Selects the nature of the offset coordinate system SkyRefIs

Description: This attribute controls how the values supplied for the SkyRef and SkyRefP attributes are used. These three attributes together allow a SkyFrame to represent offsets relative to some specified origin or pole within the coordinate system specified by the System attribute, rather than absolute axis values. SkyRefIs can take one of the case-insensitive values "Origin", "Pole" or "Ignored".

If SkyRefIs is set to "Origin", then the coordinate system represented by the SkyFrame is modified to put the origin of longitude and latitude at the position specified by the SkyRef attribute.

If SkyRefIs is set to "Pole", then the coordinate system represented by the SkyFrame is modified to put the north pole at the position specified by the SkyRef attribute.

If SkyRefIs is set to "Ignored" (the default), then any value set for the SkyRef attribute is ignored, and the SkyFrame represents the coordinate system specified by the System attribute directly without any rotation.

Type:
String.

Class Applicability:

SkyFrame
All SkyFrames have this attribute.

SkyRefP(axis) Position on primary meridian of offset coordinate system SkyRefP(axis)

Description: This attribute is used to control the orientation of the offset coordinate system defined by attributes SkyRef and SkyRefIs. If used, it should be set to hold the longitude and latitude of a point within the coordinate system specified by the System attribute. The offset coordinate system represented by the SkyFrame will then be rotated in order to put the position supplied for SkyRefP on the zero longitude meridian. This rotation is about an axis from the centre of the celestial sphere to the point specified by the SkyRef attribute. The default value for SkyRefP is usually the north pole (that is, a latitude of +90 degrees in the coordinate system specified by the System attribute). The exception to this is if the SkyRef attribute is itself set to either the north or south pole. In these cases the default for SkyRefP is the origin (that is, a (0,0) in the coordinate system specified by the System attribute).

If an integer axis index is included in the attribute name (e.g. "SkyRefP(1)") then the attribute value should be supplied as a single floating point axis value, in radians, when setting a value for the attribute, and will be returned in the same form when getting the value of the attribute. In this case the integer axis index should be "1" or "2" (the values to use for longitude and latitude axes are given by the LonAxis and LatAxis attributes).

If no axis index is included in the attribute name (e.g. "SkyRefP") then the attribute value should be supplied as a character string containing two formatted axis values (an axis 1 value followed by a comma, followed by an axis 2 value). The same form will be used when getting the value of the attribute.

Type:
Floating point.

Class Applicability:

SkyFrame
All SkyFrames have this attribute.

Notes:

- If the position given by the SkyRef attribute defines the origin of the offset coordinate system (that is, if the SkyRefIs attribute is set to "origin"), then there will in general be two orientations which will put the supplied SkyRefP position on the zero longitude meridian. The orientation which is actually used is the one which gives the SkyRefP position a positive latitude in the offset coordinate system (the other possible orientation would give the SkyRefP position a negative latitude).

- An error will be reported if an attempt is made to use a SkyRefP value which is co-incident with SkyRef or with the point diametrically opposite to SkyRef on the celestial sphere. The reporting of this error is deferred until the SkyRef and SkyRefP attribute values are used within a calculation.
- If the System attribute of the SkyFrame is changed, any position given for SkyRefP is transformed into the new System.

SortBy Determines how keys are sorted in a KeyMap SortBy

Description: This attribute determines the order in which keys are returned by the astMapKey function. It may take the following values (the default is "None"):

- "None": The keys are returned in an arbitrary order. This is the fastest method as it avoids the need for a sorted list of keys to be maintained and used.
- "AgeDown": The keys are returned in the order in which values were stored in the KeyMap, with the key for the most recent value being returned last. If the value of an existing entry is changed, it goes to the end of the list.
- "AgeUp": The keys are returned in the order in which values were stored in the KeyMap, with the key for the most recent value being returned first. If the value of an existing entry is changed, it goes to the top of the list.
- "KeyAgeDown": The keys are returned in the order in which they were originally stored in the KeyMap, with the most recent key being returned last. If the value of an existing entry is changed, its position in the list does not change.
- "KeyAgeUp": The keys are returned in the order in which they were originally stored in the KeyMap, with the most recent key being returned first. If the value of an existing entry is changed, its position in the list does not change.
- "KeyDown": The keys are returned in alphabetical order, with "A..." being returned last.
- "KeyUp": The keys are returned in alphabetical order, with "A..." being returned first.

Type:

String.

Class Applicability:

KeyMap

All KeyMaps have this attribute.

Notes:

- If a new value is assigned to SortBy (or if SortBy is cleared), all entries currently in the KeyMap are re-sorted according to the new SortBy value.

SourceFile Input file from which to read data SourceFile

Description: This attribute specifies the name of a file from which the Channel should read data. If specified it is used in preference to any source function specified when the Channel was created.

Assigning a new value to this attribute will cause any previously opened SourceFile to be closed. The first subsequent call to astRead will attempt to open the new file (an error will be reported if the file cannot be opened), and read data from it. All subsequent call to astRead will read data from the new file, until the SourceFile attribute is cleared or changed.

Clearing the attribute causes any open SourceFile to be closed. All subsequent data reads will use the source function specified when the Channel was created, or will read from standard input if no source function was specified.

If no value has been assigned to SourceFile, a null string will be returned if an attempt is made to get the attribute value.

Type:

String.

Class Applicability:

FitsChan

In the case of a FitsChan, the specified SourceFile supplements the source function specified when the FitsChan was created, rather than replacing the source function. The source file should be a text file (not a FITS file) containing one header per line. When a value is assigned to SourceFile, the file is opened and read immediately, and all headers read from the file are appended to the end of any header already in the FitsChan. The file is then closed. Clearing the SourceFile attribute has no further effect, other than nullifying the string (i.e. the file name) associated with the attribute.

Notes:

- Any open SourceFile is closed when the Channel is deleted.
- If the Channel is copied or dumped (using astCopy or astShow) the SourceFile attribute is left in a cleared state in the output Channel (i.e. the value of the SourceFile attribute is not copied).

SourceSys Spectral system in which the source velocity is stored SourceSys

Description: This attribute identifies the spectral system in which the SourceVel attribute value (the source velocity) is supplied and returned. It can be one of the following:

- "VRAD" or "VRADIO": Radio velocity (km/s)
- "VOPT" or "VOPTICAL": Optical velocity (km/s)
- "ZOPT" or "REDSHIFT": Redshift (dimensionless)
- "BETA": Beta factor (dimensionless)
- "VELO" or "VREL": Apparent radial ("relativistic") velocity (km/s)

When setting a new value for the SourceVel attribute, the source velocity should be supplied in the spectral system indicated by this attribute. Likewise, when getting the value of the SourceVel attribute, the velocity will be returned in this spectral system.

If the value of SourceSys is changed, the value stored for SourceVel will be converted from the old to the new spectral systems.

The default value is "VELO" (apparent radial velocity).

Type:

String.

Class Applicability:

SpecFrame

All SpecFrames have this attribute.

SourceVRF Rest frame in which the source velocity is stored SourceVRF

Description: This attribute identifies the rest frame in which the source velocity or redshift is stored (the source velocity or redshift is accessed using attribute SourceVel). When setting a new value for the SourceVel attribute, the source velocity or redshift should be supplied in the rest frame indicated by this attribute. Likewise, when getting the value of the SourceVel attribute, the velocity or redshift will be returned in this rest frame.

If the value of SourceVRF is changed, the value stored for SourceVel will be converted from the old to the new rest frame.

The values which can be supplied are the same as for the StdOfRest attribute (except that SourceVRF cannot be set to "Source"). The default value is "Helio".

Type:

String.

Class Applicability:

SpecFrame

All SpecFrames have this attribute.

SourceVel The source velocity SourceVel

Description: This attribute (together with SourceSys, SourceVRF, RefRA and RefDec) defines the "Source" standard of rest (see attribute StdOfRest). This is a rest frame which is moving towards the position given by RefRA and RefDec at a velocity given by SourceVel. A positive value means the source is moving away from the observer. When a new value is assigned to this attribute, the supplied value is assumed to refer to the spectral system specified by the SourceSys attribute. For instance, the SourceVel value may be supplied as a radio velocity, a redshift, a beta factor, etc. Similarly, when the current value of the SourceVel attribute is obtained, the returned value will refer to the spectral system specified by the SourceSys value. If the SourceSys value is changed, any value previously stored for the SourceVel attribute will be changed automatically from the old spectral system to the new spectral system.

When setting a value for SourceVel, the value should be supplied in the rest frame specified by the SourceVRF attribute. Likewise, when getting the value of SourceVel, it will be returned in the rest frame specified by the SourceVRF attribute.

The default SourceVel value is zero.

Type:

Floating point.

Class Applicability:

SpecFrame

All SpecFrames have this attribute.

Notes:

- It is important to set an appropriate value for SourceVRF and SourceSys before setting a value for SourceVel. If a new value is later set for SourceVRF or SourceSys, the value stored for SourceVel will simultaneously be changed to the new standard of rest or spectral system.

SpecOrigin The zero point for SpecFrame axis values SpecOrigin

Description: This specifies the origin from which all spectral values are measured. The default value (zero) results in the SpecFrame describing absolute spectral values in the system given by the System attribute (e.g. frequency, velocity, etc). If a SpecFrame is to be used to describe offset from some origin, the SpecOrigin attribute should be set to hold the required origin value. The SpecOrigin value stored inside the SpecFrame structure is modified whenever SpecFrame attribute values are changed so that it refers to the original spectral position.

When setting a new value for this attribute, the supplied value is assumed to be in the system, units and standard of rest described by the SpecFrame. Likewise, when getting the value of this attribute, the value is returned in the system, units and standard of rest described by the SpecFrame. If any of these attributes are changed, then any previously stored SpecOrigin value will also be changed so that refers to the new system, units or standard of rest.

Type:

Floating point.

Class Applicability:

SpecFrame

All SpecFrames have this attribute.

SpecVal The spectral position at which flux values are measured SpecVal

Description: This attribute specifies the spectral position (frequency, wavelength, etc.), at which the values described by the FluxFrame are measured. It is used when determining the Mapping between between FluxFrames.

The default value and spectral system used for this attribute are both specified when the FluxFrame is created.

Type:

Floating point.

Class Applicability:

FluxFrame

All FluxFrames have this attribute.

StcsArea Return the CoordinateArea component when reading an STC-S document? StcsArea

Description: This is a boolean attribute which controls what is returned by the astRead function when it is used to read from an StcsChan. If StcsArea is set non-zero (the default), then a Region representing the STC CoordinateArea will be returned by astRead. If StcsArea is set to zero, then the STC CoordinateArea will not be returned.

Type:

Integer (boolean).

Class Applicability:

StcsChan

All StcsChans have this attribute.

Notes:

- Other attributes such as StcsCoords and StcsProps can be used to specify other Objects to be returned by astRead. If more than one of these attributes is set non-zero, then the actual Object returned by astRead will be a KeyMap, containing the requested Objects. In this case, the Region representing the STC CoordinateArea will be stored in the returned KeyMap using the key "AREA". If StcsArea is the only attribute to be set non-zero, then the Object returned by astRead will be the CoordinateArea Region itself.
- The class of Region used to represent the CoordinateArea for each STC-S sub-phrase is determined by the first word in the sub-phrase (the "sub-phrase identifier"). The individual sub-phrase Regions are combined into a single Prism, which is then simplified using astSimplify to form the returned region.
- Sub-phrases that represent a single value (that is, have identifiers "Time", "Position", "Spectral" or "Redshift") are considered to be be part of the STC CoordinateArea component.
- The TimeFrame used to represent a time STC-S sub-phrase will have its TimeOrigin attribute set to the sub-phrase start time. If no start time is specified by the sub-phrase, then the stop time will be used instead. If no stop time is specified by the sub-phrase, then the single time value specified in the sub-phrase will be used instead. Subsequently clearing the TimeOrigin attribute (or setting its value to zero) will cause the TimeFrame to reprsent absolute times.
- The Epoch attribute for the returned Region is set in the same way as the TimeOrigin attribute (see above).

StcsCoords Return the Coordinates component when reading an STC-S document? StcsCoords

Description: This is a boolean attribute which controls what is returned by the astRead function when it is used to read from an StcsChan. If StcsCoords is set non-zero, then a PointList representing the STC Coordinates will be returned by astRead. If StcsCoords is set to zero (the default), then the STC Coordinates will not be returned.

Type:
Integer (boolean).

Class Applicability:

StcsChan
All StcsChans have this attribute.

Notes:

- Other attributes such as StcsArea and StcsProps can be used to specify other Objects to be returned by astRead. If more than one of these attributes is set non-zero, then the actual Object returned by astRead will be a KeyMap, containing the requested Objects. In this case, the PointList representing the STC Coordinates will be stored in the returned KeyMap using the key "COORDS". If StcsCoords is the only attribute to be set non-zero, then the Object returned by astRead will be the Coordinates PointList itself.
- The Coordinates component is specified by the additional axis values embedded within the body of each STC-S sub-phrase that represents an extended area. Sub-phrases that represent a single value (that is, have identifiers "Time", "Position", "Spectral" or "Redshift") are not considered to be be part of the STC Coordinates component.

- If the STC-S documents does not contain a Coordinates component, then a NULL object pointer will be returned by astRead if the Coordinates component is the only object being returned. If other objects are also being returned (see attributes StcsProps and StcsArea), then the returned KeyMap will contain a "COORDS" key only if the Coordinates component is read succesfully.
- The TimeFrame used to represent a time STC-S sub-phrase will have its TimeOrigin attribute set to the sub-phrase start time. If no start time is specified by the sub-phrase, then the stop time will be used instead. If no stop time is specified by the sub-phrase, then the single time value specified in the sub-phrase will be used instead. Subsequently clearing the TimeOrigin attribute (or setting its value to zero) will cause the TimeFrame to reprsent absolute times.
- The Epoch attribute for the returned Region is set in the same way as the TimeOrigin attribute (see above).

StcsLength Controls output line length StcsLength

Description: This attribute specifies the maximum length to use when writing out text through the sink function supplied when the StcsChan was created. It is ignored if the Indent attribute is zero (in which case the text supplied to the sink function can be of any length). The default value is 70.

The number of characters in each string written out through the sink function will not usually be greater than the value of this attribute (but may be less). However, if any single word in the STC-S description exceeds the specified length, then the word will be written out as a single line.

Type:

Integer.

Class Applicability:

StcsChan

All StcsChans have this attribute.

StcsProps Return all properties when reading an STC-S document? StcsProps

Description: This is a boolean attribute which controls what is returned by the astRead function when it is used to read from an StcsChan. If StcsProps is set non-zero, then a KeyMap containing all the properties read from the STC-S document will be returned by astRead. If StcsProps is set to zero (the default), then the properties will not be returned.

Type:

Integer (boolean).

Class Applicability:

StcsChan

All StcsChans have this attribute.

Notes:

- Other attributes such as StcsCoords and StcsArea can be used to specify other Objects to be returned by astRead. If more than one of these attributes is set non-zero, then the actual Object returned by astRead will be a KeyMap containing the requested Objects. In this case, the properties KeyMap will be stored in the returned KeyMap using the key "PROPS". If StcsProps is the only attribute to be set non-zero, then the Object returned by astRead will be the properties KeyMap itself.

- The KeyMap containing the properties will have entries for one or more of the following keys: `"TIME_PROPS"`, `"SPACE_PROPS"`, `"SPECTRAL_PROPS"` and `"REDSHIFT_PROPS"`. Each of these entries will be another KeyMap containing the properties of the corresponding STC-S sub-phrase.

StdOfRest Standard of rest StdOfRest

Description: This attribute identifies the standard of rest to which the spectral axis values of a SpecFrame refer, and may take any of the values listed in the `"Standards of Rest"` section (below).

The default StdOfRest value is `"Helio"`.

Type:

String.

Class Applicability:

SpecFrame

All SpecFrames have this attribute.

Standards of Rest:

The SpecFrame class supports the following StdOfRest values (all are case-insensitive):

- `"Topocentric"`, `"Topocent"` or `"Topo"`: The observers rest-frame (assumed to be on the surface of the earth). Spectra recorded in this standard of rest suffer a Doppler shift which varies over the course of a day because of the rotation of the observer around the axis of the earth. This standard of rest must be qualified using the ObsLat, ObsLon, ObsAlt, Epoch, RefRA and RefDec attributes.
- `"Geocentric"`, `"Geocentr"` or `"Geo"`: The rest-frame of the earth centre. Spectra recorded in this standard of rest suffer a Doppler shift which varies over the course of a year because of the rotation of the earth around the Sun. This standard of rest must be qualified using the Epoch, RefRA and RefDec attributes.
- `"Barycentric"`, `"Barycent"` or `"Bary"`: The rest-frame of the solar-system barycentre. Spectra recorded in this standard of rest suffer a Doppler shift which depends both on the velocity of the Sun through the Local Standard of Rest, and on the movement of the planets through the solar system. This standard of rest must be qualified using the Epoch, RefRA and RefDec attributes.
- `"Heliocentric"`, `"Heliocen"` or `"Helio"`: The rest-frame of the Sun. Spectra recorded in this standard of rest suffer a Doppler shift which depends on the velocity of the Sun through the Local Standard of Rest. This standard of rest must be qualified using the RefRA and RefDec attributes.
- `"LSRK"`, `"LSR"`: The rest-frame of the kinematical Local Standard of Rest. Spectra recorded in this standard of rest suffer a Doppler shift which depends on the velocity of the kinematical Local Standard of Rest through the galaxy. This standard of rest must be qualified using the RefRA and RefDec attributes.
- `"LSRD"`: The rest-frame of the dynamical Local Standard of Rest. Spectra recorded in this standard of rest suffer a Doppler shift which depends on the velocity of the dynamical Local Standard of Rest through the galaxy. This standard of rest must be qualified using the RefRA and RefDec attributes.
- `"Galactic"`, `"Galactoc"` or `"Gal"`: The rest-frame of the galactic centre. Spectra recorded in this standard of rest suffer a Doppler shift which depends on the velocity of the galactic centre through the local group. This standard of rest must be qualified using the RefRA and RefDec attributes.

- "Local_group", "Localgrp" or "LG": The rest-frame of the local group. This standard of rest must be qualified using the RefRA and RefDec attributes.
- "Source", or "src": The rest-frame of the source. This standard of rest must be qualified using the RefRA, RefDec and SourceVel attributes.

Where more than one alternative System value is shown above, the first of these will be returned when an enquiry is made.

Strict Report an error if any unexpeted data items are found? Strict

Description: This is a boolean attribute which indicates whether a warning rather than an error should be issed for insignificant conversion problems. If it is set non-zero, then fatal errors are issued instead of warnings, resulting in the AST error status being set. If Strict is zero (the default), then execution continues after minor conversion problems, and a warning message is added to the Channel structure. Such messages can be retrieved using the astWarnings function.

Type:

Integer (boolean).

Class Applicability:

Channel

All Channels have this attribute.

Notes:

- This attribute was introduced in AST version 5.0. Prior to this version of AST unexpected data items read by a basic Channel always caused an error to be reported. So applications linked against versions of AST prior to version 5.0 may not be able to read Object descriptions created by later versions of AST, if the Object's class description has changed.

Style(element) Line style for a Plot element Style(element)

Description: This attribute determines the line style used when drawing each element of graphical output produced by a Plot. It takes a separate value for each graphical element so that, for instance, the setting "Style(border)=2" causes the Plot border to be drawn using line style 2 (which might result in, say, a dashed line).

The range of integer line styles available and their appearance is determined by the underlying graphics system. The default behaviour is for all graphical elements to be drawn using the default line style supplied by this graphics system (normally, this is likely to give a solid line).

Type:

Integer.

Class Applicability:

Plot

All Plots have this attribute.

Notes:

- For a list of the graphical elements available, see the description of the Plot class.
- If no graphical element is specified, (e.g. "Style" instead of "Style(border)"), then a "set" or "clear" operation will affect the attribute value of all graphical elements, while a "get" or "test" operation will use just the Style(Border) value.

Symbol(axis) Axis symbol Symbol(axis)

Description: This attribute specifies a short-form symbol to be used to represent coordinate values for a particular axis of a Frame. This might be used (e.g.) in algebraic expressions where a full description of the axis would be inappropriate. Examples include "RA" and "Dec" (for Right Ascension and Declination).

If a Symbol value has not been set for a Frame axis, then a suitable default is supplied.

Type:
String.

Class Applicability:

Frame
The default Symbol value supplied by the Frame class is the string "<Domain><n>", where <n> is 1, 2, etc. for successive axes, and <Domain> is the value of the Frame's Domain attribute (truncated if necessary so that the final string does not exceed 15 characters). If no Domain value has been set, "x" is used as the <Domain> value in constructing this default string.

SkyFrame
The SkyFrame class re-defines the default Symbol value (e.g. to "RA" or "Dec") as appropriate for the particular celestial coordinate system being represented.

TimeFrame
The TimeFrame class re-defines the default Symbol value as appropriate for the particular time system being represented.

FrameSet
The Symbol attribute of a FrameSet axis is the same as that of its current Frame (as specified by the Current attribute).

Notes:

- When specifying this attribute by name, it should be subscripted with the number of the Frame axis to which it applies.

System Coordinate system used to describe positions within the domain System

Description: In general it is possible for positions within a given physical domain to be described using one of several different coordinate systems. For instance, the SkyFrame class can use galactic coordinates, equatorial coordinates, etc, to describe positions on the sky. As another example, the SpecFrame class can use frequency, wavelength, velocity, etc, to describe a position within an electromagnetic spectrum. The System attribute identifies the particular coordinate system represented by a Frame. Each class of Frame defines a set of acceptable values for this attribute, as listed below (all are case insensitive). Where more than one alternative System value is shown, the first of will be returned when an enquiry is made.

Type:
String.

Class Applicability:

Frame
The System attribute for a basic Frame always equals "Cartesian", and may not be altered.

CmpFrame

The System attribute for a CmpFrame always equals "Compound", and may not be altered. In addition, the CmpFrame class allows the System attribute to be referenced for a component Frame by including the index of an axis within the required component Frame. For instance, "System(3)" refers to the System attribute of the component Frame which includes axis 3 of the CmpFrame.

FrameSet

The System attribute of a FrameSet is the same as that of its current Frame (as specified by the Current attribute).

SkyFrame

The SkyFrame class supports the following System values and associated celestial coordinate systems:

- "AZEL": Horizon coordinates. The longitude axis is azimuth such that geographic north has an azimuth of zero and geographic east has an azimuth of +PI/2 radians. The zenith has elevation +PI/2. When converting to and from other celestial coordinate systems, no corrections are applied for atmospheric refraction or polar motion (however, a correction for diurnal aberattion is applied). Note, unlike most other celestial coordinate systems, this system is right handed. Also, unlike other SkyFrame systems, the AzEl system is sensitive to the timescale in which the Epoch value is supplied. This is because of the gross diurnal rotation which this system undergoes, causing a small change in time to translate to a large rotation. When converting to or from an AzEl system, the Epoch value for both source and destination SkyFrames should be supplied in the TDB timescale. The difference between TDB and TT is between 1 and 2 milliseconds, and so a TT value can usually be supplied in place of a TDB value. The TT timescale is related to TAI via TT = TAI + 32.184 seconds.
- "ECLIPTIC": Ecliptic coordinates (IAU 1980), referred to the ecliptic and mean equinox specified by the qualifying Equinox value.
- "FK4": The old FK4 (barycentric) equatorial coordinate system, which should be qualified by an Equinox value. The underlying model on which this is based is non-inertial and rotates slowly with time, so for accurate work FK4 coordinate systems should also be qualified by an Epoch value.
- "FK4-NO-E" or "FK4_NO_E": The old FK4 (barycentric) equatorial system but without the "E-terms of aberration" (e.g. some radio catalogues). This coordinate system should also be qualified by both an Equinox and an Epoch value.
- "FK5" or "EQUATORIAL": The modern FK5 (barycentric) equatorial coordinate system. This should be qualified by an Equinox value.
- "GALACTIC": Galactic coordinates (IAU 1958).
- "GAPPT", "GEOCENTRIC" or "APPARENT": The geocentric apparent equatorial coordinate system, which gives the apparent positions of sources relative to the true plane of the Earth's equator and the equinox (the coordinate origin) at a time specified by the qualifying Epoch value. (Note that no Equinox is needed to qualify this coordinate system because no model "mean equinox" is involved.) These coordinates give the apparent right ascension and declination of a source for a specified date of observation, and therefore form an approximate basis for pointing a telescope. Note, however, that they are applicable to a fictitious observer at the Earth's centre, and therefore ignore such effects as atmospheric refraction and the (normally much smaller) aberration of light due to the rotational velocity of the Earth's surface. Geocentric apparent coordinates are derived from the standard FK5 (J2000.0) barycentric coordinates by taking account of the gravitational deflection of light by the Sun (usually small), the aberration of light caused by the motion of the Earth's centre with respect to the barycentre (larger), and the precession and nutation of the Earth's spin axis (normally larger still).

- "HELIOECLIPTIC": Ecliptic coordinates (IAU 1980), referred to the ecliptic and mean equinox of J2000.0, in which an offset is added to the longitude value which results in the centre of the sun being at zero longitude at the date given by the Epoch attribute. Attempts to set a value for the Equinox attribute will be ignored, since this system is always referred to J2000.0.
- "ICRS": The Internation Celestial Reference System, realised through the Hipparcos catalogue. Whilst not an equatorial system by definition, the ICRS is very close to the FK5 (J2000) system and is usually treated as an equatorial system. The distinction between ICRS and FK5 (J2000) only becomes important when accuracies of 50 milli-arcseconds or better are required. ICRS need not be qualified by an Equinox value.
- "J2000": An equatorial coordinate system based on the mean dynamical equator and equinox of the J2000 epoch. The dynamical equator and equinox differ slightly from those used by the FK5 model, and so a "J2000" SkyFrame will differ slightly from an "FK5(Equinox=J2000)" SkyFrame. The J2000 System need not be qualified by an Equinox value
- "SUPERGALACTIC": De Vaucouleurs Supergalactic coordinates.
- "UNKNOWN": Any other general spherical coordinate system. No Mapping can be created between a pair of SkyFrames if either of the SkyFrames has System set to "Unknown".

Currently, the default System value is "ICRS". However, this default may change in future as new astrometric standards evolve. The intention is to track the most modern appropriate standard. For this reason, you should use the default only if this is what you intend (and can tolerate any associated slight change in future). If you intend to use the ICRS system indefinitely, then you should specify it explicitly.

SpecFrame

The SpecFrame class supports the following System values and associated spectral coordinate systems (the default is "WAVE" - wavelength). They are all defined in FITS-WCS paper III:

- "FREQ": Frequency (GHz)
- "ENER" or "ENERGY": Energy (J)
- "WAVN" or "WAVENUM": Wave-number (1/m)
- "WAVE" or "WAVELEN": Vacuum wave-length (Angstrom)
- "AWAV" or "AIRWAVE": Wave-length in air (Angstrom)
- "VRAD" or "VRADIO": Radio velocity (km/s)
- "VOPT" or "VOPTICAL": Optical velocity (km/s)
- "ZOPT" or "REDSHIFT": Redshift (dimensionless)
- "BETA": Beta factor (dimensionless)
- "VELO" or "VREL": Apparent radial ("relativistic") velocity (km/s)

The default value for the Unit attribute for each system is shown in parentheses. Note that the default value for the ActiveUnit flag is non-zero for a SpecFrame, meaning that changes to the Unit attribute for a SpecFrame will result in the SpecFrame being re-mapped within its enclosing FrameSet in order to reflect the change in units (see astSetActiveUnit function for further information).

TimeFrame

The TimeFrame class supports the following System values and associated coordinate systems (the default is "MJD"):

- "MJD": Modified Julian Date (d)
- "JD": Julian Date (d)
- "JEPOCH": Julian epoch (yr)

- "BEPOCH": Besselian (yr)

The default value for the Unit attribute for each system is shown in parentheses. Strictly, these systems should not allow changes to be made to the units. For instance, the usual definition of "MJD" and "JD" include the statement that the values will be in units of days. However, AST does allow the use of other units with all the above supported systems (except BEPOCH), on the understanding that conversion to the "correct" units involves nothing more than a simple scaling (1 yr = 365.25 d, 1 d = 24 h, 1 h = 60 min, 1 min = 60 s). Besselian epoch values are defined in terms of tropical years of 365.2422 days, rather than the usual Julian year of 365.25 days. Therefore, to avoid any confusion, the Unit attribute is automatically cleared to "yr" when a System value of BEPOCH System is selected, and an error is reported if any attempt is subsequently made to change the Unit attribute.

Note that the default value for the ActiveUnit flag is non-zero for a TimeFrame, meaning that changes to the Unit attribute for a TimeFrame will result in the TimeFrame being re-mapped within its enclosing FrameSet in order to reflect the change in units (see astSetActiveUnit function for further information).

FluxFrame

The FluxFrame class supports the following System values and associated systems for measuring observed value:

- "FLXDN": Flux per unit frequency (W/m^2/Hz)
- "FLXDNW": Flux per unit wavelength (W/m^2/Angstrom)
- "SFCBR": Surface brightness in frequency units (W/m^2/Hz/arcmin**2)
- "SFCBRW": Surface brightness in wavelength units (W/m^2/Angstrom/arcmin**2)

The above lists specified the default units for each System. If an explicit value is set for the Unit attribute but no value is set for System, then the default System value is determined by the Unit string (if the units are not appropriate for describing any of the supported Systems then an error will be reported when an attempt is made to access the System value). If no value has been specified for either Unit or System, then System=FLXDN and Unit=W/m^2/Hz are used.

TabOK Should the FITS-WCS -TAB algorithm be recognised? TabOK

Description: This attribute is an integer value which indicates if the "-TAB" algorithm, defined in FITS-WCS paper III, should be supported by the FitsChan. The default value is zero. A zero or negative value results in no support for -TAB axes (i.e. axes that have "-TAB" in their CTYPE keyword value). In this case, the astWrite method will return zero if the write operation would required the use of the -TAB algorithm, and the astRead method will return a NULL pointer if any axis in the supplied header uses the -TAB algorithm.

If TabOK is set to a non-zero positive integer, these methods will recognise and convert axes described by the -TAB algorithm, as follows:

The astWrite method will generate headers that use the -TAB algorithm (if possible) if no other known FITS-WCS algorithm can be used to describe the supplied FrameSet. This will result in a table of coordinate values and index vectors being stored in the FitsChan. After the write operation, the calling application should check to see if such a table has been stored in the FitsChan. If so, the table should be retrived from the FitsChan using the astGetTables method, and the data (and headers) within it copied into a new FITS binary table extension. See astGetTables for more information. The FitsChan uses a FitsTable object to store the table data and headers. This FitsTable will contain the required columns and headers as described by FITS-WCS paper III - the coordinates array will be in a column named "COORDS", and the index vector(s) will be in columns named "INDEX<i>" (where <i> is the index of the corresponding FITS WCS axis).

Note, index vectors are only created if required. The EXTNAME value will be set to the value of the AST__TABEXTNAME constant (currently "WCS-TAB"). The EXTVER header will be set to the positive integer value assigned to the TabOK attribute. No value will be stored for the EXTLEVEL header, and should therefore be considered to default to 1.

The astRead method will generate a FrameSet from headers that use the -TAB algorithm so long as the necessary FITS binary tables are made available. There are two ways to do this: firstly, if the application knows which FITS binary tables will be needed, then it can create a Fitstable describing each such table and store it in the FitsChan (using method astPutTables or astPutTable) before invoking the astRead method. Secondly, if the application does not know which FITS binary tables will be needed by astRead, then it can register a call-back function with the FitsChan using method astTableSource. This call-back function will be called from within astRead if and when a -TAB header is encountered. When called, its arguments will give the name, version and level of the FITS extension containing a required table. The call-back function should read this table from an external FITS file, and create a corresponding FitsTable which it should then return to astRead. Note, currently astRead can only handle -TAB headers that describe 1-dimensional (i.e. separable) axes.

Type:

Integer.

Class Applicability:

FitsChan

All FitsChans have this attribute.

TextLab(axis) Draw descriptive axis labels for a Plot? TextLab(axis)

Description: This attribute controls the appearance of an annotated coordinate grid (drawn with the astGrid function) by determining whether textual labels should be drawn to describe the quantity being represented on each axis of a Plot. It takes a separate value for each physical axis of a Plot so that, for instance, the setting "TextLab(2)=1" specifies that descriptive labels should be drawn for the second axis.

If the TextLab value of a Plot axis is non-zero, then descriptive labels will be drawn for that axis, otherwise they will be omitted. The default behaviour is to draw descriptive labels if tick marks and numerical labels are being drawn around the edges of the plotting area (see the Labelling attribute), but to omit them otherwise.

Type:

Integer (boolean).

Class Applicability:

Plot

All Plots have this attribute.

Notes:

- The text used for the descriptive labels is derived from the Plot's Label(axis) attribute, together with its Unit(axis) attribute if appropriate (see the LabelUnits(axis) attribute).
- The drawing of numerical axis labels for a Plot (which indicate values on the axis) is controlled by the NumLab(axis) attribute.
- If no axis is specified, (e.g. "TextLab" instead of "TextLab(2)"), then a "set" or "clear" operation will affect the attribute value of all the Plot axes, while a "get" or "test" operation will use just the TextLab(1) value.

TextLabGap(axis) Spacing of descriptive axis labels for a Plot TextLabGap(axis)

Description: This attribute controls the appearance of an annotated coordinate grid (drawn with the astGrid function) by determining where descriptive axis labels are placed relative to the axes they describe. It takes a separate value for each physical axis of a Plot so that, for instance, the setting "TextLabGap(2)=0.01" specifies where the descriptive label for the second axis should be drawn.

For each axis, the TextLabGap value gives the spacing between the descriptive label and the edge of a box enclosing all other parts of the annotated grid (excluding other descriptive labels). The gap is measured to the nearest edge of the label (i.e. the top or the bottom). Positive values cause the descriptive label to be placed outside the bounding box, while negative values cause it to be placed inside.

The TextLabGap value should be given as a fraction of the minimum dimension of the plotting area, the default value being +0.01.

Type:

Floating point.

Class Applicability:

Plot

All Plots have this attribute.

Notes:

- If drawn, descriptive labels are always placed at the edges of the plotting area, even although the corresponding numerical labels may be drawn along axis lines in the interior of the plotting area (see the Labelling attribute).
- If no axis is specified, (e.g. "TextLabGap" instead of "TextLabGap(2)"), then a "set" or "clear" operation will affect the attribute value of all the Plot axes, while a "get" or "test" operation will use just the TextLabGap(1) value.

TickAll Draw tick marks on all edges of a Plot? TickAll

Description: This attribute controls the appearance of an annotated coordinate grid (drawn with the astGrid function) by determining whether tick marks should be drawn on all edges of a Plot.

If the TickAll value of a Plot is non-zero (the default), then tick marks will be drawn on all edges of the Plot. Otherwise, they will be drawn only on those edges where the numerical and descriptive axis labels are drawn (see the Edge(axis) attribute).

Type:

Integer (boolean).

Class Applicability:

Plot

All Plots have this attribute.

Notes:

- In some circumstances, numerical labels and tick marks are drawn along grid lines inside the plotting area, rather than around its edges (see the Labelling attribute). In this case, the value of the TickAll attribute is ignored.

TimeOrigin The zero point for TimeFrame axis values TimeOrigin

Description: This specifies the origin from which all time values are measured. The default value (zero) results in the TimeFrame describing absolute time values in the system given by the System attribute (e.g. MJD, Julian epoch, etc). If a TimeFrame is to be used to describe elapsed time since some origin, the TimeOrigin attribute should be set to hold the required origin value. The TimeOrigin value stored inside the TimeFrame structure is modified whenever TimeFrame attribute values are changed so that it refers to the original moment in time.

Type:

Floating point.

Class Applicability:

TimeFrame

All TimeFrames have this attribute.

Input Formats:

The formats accepted when setting a TimeOrigin value are listed below. They are all case-insensitive and are generally tolerant of extra white space and alternative field delimiters:

- Besselian Epoch: Expressed in decimal years, with or without decimal places ("B1950" or "B1976.13" for example).
- Julian Epoch: Expressed in decimal years, with or without decimal places ("J2000" or "J2100.9" for example).
- Units: An unqualified decimal value is interpreted as a value in the system specified by the TimeFrame's System attribute, in the units given by the TimeFrame's Unit attribute. Alternatively, an appropriate unit string can be appended to the end of the floating point value ("123.4 d" for example), in which case the supplied value is scaled into the units specified by the Unit attribute.
- Julian Date: With or without decimal places ("JD 2454321.9" for example).
- Modified Julian Date: With or without decimal places ("MJD 54321.4" for example).
- Gregorian Calendar Date: With the month expressed either as an integer or a 3-character abbreviation, and with optional decimal places to represent a fraction of a day ("1996-10-2" or "1996-Oct-2.6" for example). If no fractional part of a day is given, the time refers to the start of the day (zero hours).
- Gregorian Date and Time: Any calendar date (as above) but with a fraction of a day expressed as hours, minutes and seconds ("1996-Oct-2 12:13:56.985" for example). The date and time can be separated by a space or by a "T" (as used by ISO8601 format).

Output Format:

When enquiring TimeOrigin values, the returned formatted floating point value represents a value in the TimeFrame's System, in the unit specified by the TimeFrame's Unit attribute.

TimeScale Time scale TimeScale

Description: This attribute identifies the time scale to which the time axis values of a TimeFrame refer, and may take any of the values listed in the "Time Scales" section (below).

The default TimeScale value depends on the current System value; if the current TimeFrame system is "Besselian epoch" the default is "TT", otherwise it is "TAI". Note, if the System attribute is set so that the TimeFrame represents Besselian Epoch, then an error will be reported if an attempt is made to set the TimeScale to anything other than TT.

Note, the supported time scales fall into two groups. The first group containing UT1, GMST, LAST and LMST define time in terms of the orientation of the earth. The second group (containing all the remaining time scales) define time in terms of an atomic process. Since the rate of rotation of the earth varies in an unpredictable way, conversion between two timescales in different groups relies on a value being supplied for the Dut1 attribute (defined by the parent Frame class). This attribute specifies the difference between the UT1 and UTC time scales, in seconds, and defaults to zero. See the documentation for the Dut1 attribute for further details.

Type:

String.

Class Applicability:

TimeFrame

All TimeFrames have this attribute.

Time Scales:

The TimeFrame class supports the following TimeScale values (all are case-insensitive):

- "TAI" - International Atomic Time
- "UTC" - Coordinated Universal Time
- "UT1" - Universal Time
- "GMST" - Greenwich Mean Sidereal Time
- "LAST" - Local Apparent Sidereal Time
- "LMST" - Local Mean Sidereal Time
- "TT" - Terrestrial Time
- "TDB" - Barycentric Dynamical Time
- "TCB" - Barycentric Coordinate Time
- "TCG" - Geocentric Coordinate Time
- "LT" - Local Time (the offset from UTC is given by attribute LTOffset)

An very informative description of these and other time scales is available at http://www.ucolick.org/∼sla/leapsecs/time

UTC Warnings:

UTC should ideally be expressed using separate hours, minutes and seconds fields (or at least in seconds for a given date) if leap seconds are to be taken into account. Since the TimeFrame class represents each moment in time using a single floating point number (the axis value) there will be an ambiguity during a leap second. Thus an error of up to 1 second can result when using AST to convert a UTC time to another time scale if the time occurs within a leap second. Leap seconds occur at most twice a year, and are introduced to take account of variation in the rotation of the earth. The most recent leap second occurred on 1st January 1999. Although in the vast majority of cases leap second ambiguities won't matter, there are potential problems in on-line data acquisition systems and in critical applications involving taking the difference between two times.

Title Frame title Title

Description: This attribute holds a string which is used as a title in (e.g.) graphical output to describe the coordinate system which a Frame represents. Examples might be "Detector Coordinates" or "Galactic Coordinates".

If a Title value has not been set for a Frame, then a suitable default is supplied, depending on the class of the Frame.

Type:
String.

Class Applicability:

Frame
The default supplied by the Frame class is "<n>-d coordinate system", where <n> is the number of Frame axes (Naxes attribute).

CmpFrame
The CmpFrame class re-defines the default Title value to be "<n>-d compound coordinate system", where <n> is the number of CmpFrame axes (Naxes attribute).

FrameSet
The Title attribute of a FrameSet is the same as that of its current Frame (as specified by the Current attribute).

Notes:

- A Frame's Title is intended purely for interpretation by human readers and not by software.

TitleGap Vertical spacing for a Plot title TitleGap

Description: This attribute controls the appearance of an annotated coordinate grid (drawn with the astGrid function) by determining where the title of a Plot is drawn.

Its value gives the spacing between the bottom edge of the title and the top edge of a bounding box containing all the other parts of the annotated grid. Positive values cause the title to be drawn outside the box, while negative values cause it to be drawn inside.

The TitleGap value should be given as a fraction of the minimum dimension of the plotting area, the default value being +0.05.

Type:
Floating point.

Class Applicability:

Plot
All Plots have this attribute.

Plot3D
The Plot3D class ignores this attributes since it does not draw a Title.

Notes:

- The text used for the title is obtained from the Plot's Title attribute.

Tol Plotting tolerance Tol

Description: This attribute specifies the plotting tolerance (or resolution) to be used for the graphical output produced by a Plot. Smaller values will result in smoother and more accurate curves being drawn, but may slow down the plotting process. Conversely, larger values may speed up the plotting process in cases where high resolution is not required.

The Tol value should be given as a fraction of the minimum dimension of the plotting area, and should lie in the range from 1.0e-7 to 1.0. By default, a value of 0.01 is used.

Type:

Floating point.

Class Applicability:

Plot

All Plots have this attribute.

TolInverse Target relative error for the iterative inverse transformation TolInverse

Description: This attribute controls the iterative inverse transformation used if the IterInverse attribute is non-zero.

Its value gives the target relative error in teh axis values of each transformed position. Further iterations will be performed until the target relative error is reached, or the maximum number of iterations given by attribute NiterInverse is reached.

The default value is 1.0E-6.

Type:

Floating point.

Class Applicability:

PolyMap

All PolyMaps have this attribute.

Top(axis) Highest axis value to display Top(axis)

Description: This attribute gives the highest axis value to be displayed (for instance, by the astGrid method).

Type:

Floating point.

Class Applicability:

Frame

The default supplied by the Frame class is to display all axis values, without any limit.

SkyFrame

The SkyFrame class re-defines the default Top value to +90 degrees for latitude axes, and 180 degrees for co-latitude axes. The default for longitude axes is to display all axis values.

Notes:

- When specifying this attribute by name, it should be subscripted with the number of the Frame axis to which it applies.

TranForward Forward transformation defined? TranForward

Description: This attribute indicates whether a Mapping is able to transform coordinates in the "forward" direction (i.e. converting input coordinates into output coordinates). If this attribute is non-zero, the forward transformation is available. Otherwise, it is not.

Type:

Integer (boolean), read-only.

Class Applicability:

Mapping

All Mappings have this attribute.

CmpMap

The TranForward attribute value for a CmpMap is given by the boolean AND of the value for each component Mapping.

FrameSet

The TranForward attribute of a FrameSet applies to the transformation which converts between the FrameSet's base Frame and its current Frame (as specified by the Base and Current attributes). This value is given by the boolean AND of the TranForward values which apply to each of the individual sub-Mappings required to perform this conversion. The TranForward attribute value for a FrameSet may therefore change if a new Base or Current Frame is selected.

Notes:

- An error will result if a Mapping with a TranForward value of zero is used to transform coordinates in the forward direction.

TranInverse Inverse transformation defined? TranInverse

Description: This attribute indicates whether a Mapping is able to transform coordinates in the "inverse" direction (i.e. converting output coordinates back into input coordinates). If this attribute is non-zero, the inverse transformation is available. Otherwise, it is not.

Type:

Integer (boolean), readonly.

Class Applicability:

Mapping

All Mappings have this attribute.

CmpMap

The TranInverse attribute value for a CmpMap is given by the boolean AND of the value for each component Mapping.

FrameSet

The TranInverse attribute of a FrameSet applies to the transformation which converts between the FrameSet's current Frame and its base Frame (as specified by the Current and Base attributes). This value is given by the boolean AND of the TranInverse values which apply to each of the individual sub-Mappings required to perform this conversion. The TranInverse attribute value for a FrameSet may therefore change if a new Base or Current Frame is selected.

Notes:

- An error will result if a Mapping with a TranInverse value of zero is used to transform coordinates in the inverse direction.

Unit(axis) Axis physical units Unit(axis)

Description: This attribute contains a textual representation of the physical units used to represent coordinate values on a particular axis of a Frame. The astSetActiveUnit function controls how the Unit values are used.

Type:
String.

Class Applicability:

Frame
The default supplied by the Frame class is an empty string.

SkyFrame
The SkyFrame class re-defines the default Unit value (e.g. to "hh:mm:ss.sss") to describe the character string returned by the astFormat function when formatting coordinate values.

SpecFrame
The SpecFrame class re-defines the default Unit value so that it is appropriate for the current System value. See the System attribute for details. An error will be reported if an attempt is made to use an inappropriate Unit.

TimeFrame
The TimeFrame class re-defines the default Unit value so that it is appropriate for the current System value. See the System attribute for details. An error will be reported if an attempt is made to use an inappropriate Unit (e.g. "km").

FrameSet
The Unit attribute of a FrameSet axis is the same as that of its current Frame (as specified by the Current attribute).

Notes:

- When specifying this attribute by name, it should be subscripted with the number of the Frame axis to which it applies.

UnitRadius SphMap input vectors lie on a unit sphere? UnitRadius

Description: This is a boolean attribute which indicates whether the 3-dimensional vectors which are supplied as input to a SphMap are known to always have unit length, so that they lie on a unit sphere centred on the origin.

If this condition is true (indicated by setting UnitRadius non-zero), it implies that a CmpMap which is composed of a SphMap applied in the forward direction followed by a similar SphMap applied in the inverse direction may be simplified (e.g. by astSimplify) to become a UnitMap. This is because the input and output vectors will both have unit length and will therefore have the same coordinate values.

If UnitRadius is zero (the default), then although the output vector produced by the CmpMap (above) will still have unit length, the input vector may not have. This will, in general, change the coordinate values, so it prevents the pair of SphMaps being simplified.

Type:

Integer (boolean).

Class Applicability:

SphMap

All SphMaps have this attribute.

Notes:

- This attribute is intended mainly for use when SphMaps are involved in a sequence of Mappings which project (e.g.) a dataset on to the celestial sphere. By regarding the celestial sphere as a unit sphere (and setting UnitRadius to be non-zero) it becomes possible to cancel the SphMaps present, along with associated sky projections, when two datasets are aligned using celestial coordinates. This often considerably improves performance.
- Such a situations often arises when interpreting FITS data and is handled automatically by the FitsChan class.
- The value of the UnitRadius attribute is used only to control the simplification of Mappings and has no effect on the value of the coordinates transformed by a SphMap. The lengths of the input 3-dimensional Cartesian vectors supplied are always ignored, even if UnitRadius is non-zero.

UseDefs Use default values for unspecified attributes? UseDefs

Description: This attribute specifies whether default values should be used internally for object attributes which have not been assigned a value explicitly. If a non-zero value (the default) is supplied for UseDefs, then default values will be used for attributes which have not explicitly been assigned a value. If zero is supplied for UseDefs, then an error will be reported if an attribute for which no explicit value has been supplied is needed internally within AST.

Many attributes (including the UseDefs attribute itself) are unaffected by the setting of the UseDefs attribute, and default values will always be used without error for such attributes. The "Applicability:" section below lists the attributes which are affected by the setting of UseDefs.

Note, UseDefs only affects access to attributes internally within AST. The public accessor functions such as astGetC is unaffected by the UseDefs attribute - default values will always be returned if no value has been set. Application code should use the astTest function if required to determine if a value has been set for an attribute.

Type:

Integer (boolean).

Class Applicability:

Object

All Objects have this attribute, but ignore its setting except as described below for individual classes.

FrameSet

The default value of UseDefs for a FrameSet is redefined to be the UseDefs value of its current Frame.

CmpFrame

The default value of UseDefs for a CmpFrame is redefined to be the UseDefs value of its first component Frame.

Region

The default value of UseDefs for a Region is redefined to be the UseDefs value of its encapsulated Frame.

Frame

If UseDefs is zero, an error is reported when aligning Frames if the Epoch, ObsLat or ObsLon attribute is required but has not been assigned a value explicitly.

SkyFrame

If UseDefs is zero, an error is reported when aligning SkyFrames if any of the following attributes are required but have not been assigned a value explicitly: Epoch, Equinox.

SpecFrame

If UseDefs is zero, an error is reported when aligning SpecFrames if any of the following attributes are required but have not been assigned a value explicitly: Epoch, RefRA, RefDec, RestFreq, SourceVel, StdOfRest.

DSBSpecFrame

If UseDefs is zero, an error is reported when aligning DSBSpecFrames or when accessing the ImagFreq attribute if any of the following attributes are required but have not been assigned a value explicitly: Epoch, DSBCentre, IF.

Variant Indicates which variant of the current Frame is to be used Variant

Description: This attribute can be used to change the Mapping that connects the current Frame to the other Frames in the FrameSet. By default, each Frame in a FrameSet is connected to the other Frames by a single Mapping that can only be changed by using the astRemapFrame method. However, it is also possible to associate multiple Mappings with a Frame, each Mapping having an identifying name. If this is done, the "Variant" attribute can be set to indicate the name of the Mapping that is to be used with the current Frame.

A possible (if unlikely) use-case is to create a FrameSet that can be used to describe the WCS of an image formed by co-adding images of two different parts of the sky. In such an image, each pixel contains flux from two points on the sky.and so the WCS for the image should ideally contain one pixel Frame and two SkyFrames - one describing each of the two co-added images. There is nothing to prevent a FrameSet containing two explicit SkyFrames, but the problem then arises of how to distinguish between them. The two primary characteristics of a Frame that distinguishes it from other Frames ar eits class and its Domain attribute value. The class of a Frame cannot be changed, but we could in principle use two different Domain values to distinguish the two SkyFrames. However, in practice it is not uncommon for application software to assume that SkyFrames will have the default Domain value of "SKY". That is, instead of searching for Frames that have a class of "SkyFrame", such software searches for Frames that have a Domain of "SKY". To alleviate this problem, it is possible to add a single SkyFrame to the FrameSet, but specifying two alternate Mappings to use with the SkyFrame. Setting the "Variant" attribute to the name of one or the other of these alternate Mappings will cause the SkyFrame to be remapped within the FrameSet so that it uses the specified Mapping. The same facility can be used with any class of Frame, not just SkyFrames.

To use this facility, the Frame should first be added to the FrameSet in the usual manner using the astAddFrame method. By default, the Mapping supplied to astAddFrame is assigned a name equal to the Domain name of the Frame. To assign a different name to it, the astAddVariant method should then be called specifying the required name and a NULL Mapping. The astAddFrame method should then be called repeatedly to add each required extra Mapping to the current Frame, supplying a unique name for each one.

Each Frame in a FrameSet can have its own set of variant Mappings. To control the Mappings in use with a specific Frame, you need first to make it the current Frame in the FrameSet.

The astMirrorVariants function allows the effects of variant Mappings associated with a nominated Frame to be propagated to other Frames in the FrameSet.

Once this has been done, setting a new value for the "Variant" attribute of a FrameSet will cause the current Frame in the FrameSet to be remapped to use the specified variant Mapping. An error will be reported if the current Frame has no variant Mapping with the supplied name.

Getting the value of the "Variant" attribute will return the name of the variant Mapping currently in use with the current Frame. If the Frame has no variant Mappings, the value will default to the Domain name of the current Frame.

Clearing the "Variant" attribute will have the effect of removing all variant Mappings (except for the currently selected Mapping) from the current Frame.

Testing the "Variant" attribute will return a non-zero value if the current Frame contains any variant Mappings, and zero otherwise.

A complete list of the names associated with all the available variant Mappings in the current Frame can be obtained from the AllVariants attribute.

If a Frame with variant Mappings is remapped using the astRemapFrame method, the currently selected variant Mapping is used by astRemapFrame and the other variant Mappings are removed from the Frame.

Type:
String.

Class Applicability:

FrameSet
All FrameSets have this attribute.

Warnings Controls the issuing of warnings about various conditions Warnings

Description: This attribute controls the issuing of warnings about selected conditions when an Object or keyword is read from or written to a FitsChan. The value supplied for the Warnings attribute should consist of a space separated list of condition names (see the AllWarnings attribute for a list of the currently defined names). Each name indicates a condition which should be reported. The default value for Warnings is the string "Tnx Zpx BadCel BadMat BadPV BadCTYPE".

The text of any warning will be stored within the FitsChan in the form of one or more new header cards with keyword ASTWARN. If required, applications can check the FitsChan for ASTWARN cards (using astFindFits) after the call to astRead or astWrite has been performed, and report the text of any such cards to the user. ASTWARN cards will be propagated to any output header unless they are deleted from the FitsChan using astDelFits.

Type:
String

Class Applicability:

FitsChan
All FitsChans have this attribute.

Notes:

This attribute only controls the warnings that are to be stored as a set of header cards in the FitsChan as described above. It has no effect on the storage of warnings in the parent Channel structure. All warnings are stored in the parent Channel structure, from where they can be retrieved using the astWarnings function.

WcsAxis(lonlat) FITS-WCS projection axes WcsAxis(lonlat)

Description: This attribute gives the indices of the longitude and latitude coordinates of the FITS-WCS projection within the coordinate space used by a WcsMap. These indices are defined when the WcsMap is first created using astWcsMap and cannot subsequently be altered.

If "lonlat" is 1, the index of the longitude axis is returned. Otherwise, if it is 2, the index of the latitude axis is returned.

Type:
Integer, read-only.

Class Applicability:

WcsMap
All WcsMaps have this attribute.

WcsType FITS-WCS projection type WcsType

Description: This attribute specifies which type of FITS-WCS projection will be performed by a WcsMap. The value is specified when a WcsMap is first created using astWcsMap and cannot subsequently be changed.

The values used are represented by macros with names of the form "AST__XXX", where "XXX" is the (upper case) 3-character code used by the FITS-WCS "CTYPEi" keyword to identify the projection. For example, possible values are AST__TAN (for the tangent plane or gnomonic projection) and AST__AIT (for the Hammer-Aitoff projection). AST__TPN is an exception in that it is not part of the FITS-WCS standard (it represents a TAN projection with polynomial correction terms as defined in an early draft of the FITS-WCS paper).

Type:
Integer, read-only.

Class Applicability:

WcsMap
All WcsMaps have this attribute.

Notes:

- For a list of available projections, see the FITS-WCS paper.

Width(element) Line width for a Plot element Width(element)

Description: This attribute determines the line width used when drawing each element of graphical output produced by a Plot. It takes a separate value for each graphical element so that, for instance, the setting "Width(border)=2.0" causes the Plot border to be drawn using a line width of 2.0. A value of 1.0 results in a line thickness which is approximately 0.0005 times the length of the diagonal of the entire display surface.

The actual appearance of lines drawn with any particular width, and the range of available widths, is determined by the underlying graphics system. The default behaviour is for all graphical elements to be drawn using the default line width supplied by this graphics system. This will not necessarily correspond to a Width value of 1.0.

Type:
Floating point.

Class Applicability:

Plot
All Plots have this attribute.

Notes:

- For a list of the graphical elements available, see the description of the Plot class.
- If no graphical element is specified, (e.g. "Width" instead of "Width(border)"), then a "set" or "clear" operation will affect the attribute value of all graphical elements, while a "get" or "test" operation will use just the Width(Border) value.

XmlFormat System for formatting Objects as XML XmlFormat

Description: This attribute specifies the formatting system to use when AST Objects are written out as XML through an XmlChan. It affects the behaviour of the astWrite function when they are used to transfer any AST Object to or from an external XML representation.

The XmlChan class allows AST objects to be represented in the form of XML in several ways (conventions) and the XmlFormat attribute is used to specify which of these should be used. The formatting options available are outlined in the "Formats Available" section below.

By default, an XmlChan will attempt to determine which format system is already in use, and will set the default XmlFormat value accordingly (so that subsequent I/O operations adopt the same conventions). It does this by looking for certain critical items which only occur in particular formats. For details of how this works, see the "Choice of Default Format" section below. If you wish to ensure that a particular format system is used, independently of any XML already read, you should set an explicit XmlFormat value yourself.

Type:
String.

Class Applicability:

XmlChan
All XmlChans have this attribute.

Formats Available:

The XmlFormat attribute can take any of the following (case insensitive) string values to select the corresponding formatting system:

- "NATIVE": This is a direct conversion to XML of the heirarchical format used by a standard XML channel (and also by the NATIVE encoding of a FitsChan).
- "QUOTED": This is the same as NATIVE format except that extra information is included which allows client code to convert the XML into a form which can be read by a standard AST Channel. This extra information indicates which AST attribute values should be enclosed in quotes before being passed to a Channel.
- "IVOA": This is a format that uses an early draft of the STC-X schema developed by the International Virtual Observatory Alliance (IVOA - see "http://www.ivoa.net/") to describe coordinate systems, regions, mappings, etc. Support is limited to V1.20 described at "http://www.ivoa.net/Documents/WD/STC/STC-20050225.html". Since the version of STC-X finally adopted by the IVOA differs in several significant respects from V1.20, this format is now mainly of historical interest. Note, the alternative "STC-S" format (a simpler non-XML encoding of the STC metadata) is supported by the StcsChan class.

Choice of Default Format;:

If the XmlFormat attribute of an XmlChan is not set, the default value it takes is determined by the presence of certain critical items within the document most recently read using astRead. The sequence of decision used to arrive at the default value is as follows:

- If the previous document read contained any elements in any of the STC namespaces ("urn:nvo-stc", "urn:nvo-coords" or "urn:nvo-region"), then the default value is IVOA.
- If the previous document read contained any elements in the AST namespace which had an associated XML attribute called "quoted", then the default value is QUOTED.
- Otherwise, if none of these conditions is met (as would be the case if no document had yet been read), then NATIVE format is used.

Setting an explicit value for the XmlFormat attribute always over-rides this default behaviour.

The IVOA Format:

The IVOA support caters only for certain parts of V1.20 of the draft Space-Time Coordinate (STC) schema (see http://www.ivoa.net/Documents/WD/STC/STC-20050225.html). Note, this draft has now been superceded by an officially adopted version that differs in several significant respects from V1.20. Consequently, the "IVOA" XmlChan format is of historical interest only.

The following points should be noted when using an XmlChan to read or write STC information (note, this list is currently incomplete):

- Objects can currently only be read using this format, not written.
- The AST object generated by reading an <STCMetadata> element will be an instance of one of the AST "Stc" classes: StcResourceProfile, StcSearchLocation, StcCatalogEntryLocation, StcObsDataLocation.
- When reading an <STCMetadata> element, the axes in the returned AST Object will be in the order space, time, spectral, redshift, irrespective of the order in which the axes occur in the <STCMetadata> element. If the supplied <STCMetadata> element does not contain all of these axes, the returned AST Object will also omit them, but the ordering of those axes which are present will be as stated above. If the spatial frame represents a celestial coordinate system the spatial axes will be in the order (longitude, latitude).
- Until such time as the AST TimeFrame is complete, a simple 1-dimensional Frame (with Domain set to TIME) will be used to represent the STC <TimeFrame> element. Consequently, most of the information within a <TimeFrame> element is currently ignored.
- <SpaceFrame> elements can only be read if they describe a celestial longitude and latitude axes supported by the AST SkyFrame class. The space axes will be returned in the order (longitude, latitude).
- Velocities associated with SpaceFrames cannot be read.
- Any <GenericCoordFrame> elements within an <AstroCoordSystem> element are currently ignored.
- Any second or subsequent <AstroCoordSystem> found within an STCMetaData element is ignored.
- Any second or subsequent <AstroCoordArea> found within an STCMetaData element is ignored.
- Any <OffsetCenter> found within a <SpaceFrame> is ignored.
- Any CoordFlavor element found within a <SpaceFrame> is ignored.
- <SpaceFrame> elements can only be read if they refer to one of the following space reference frames: ICRS, GALACTIC_II, SUPER_GALACTIC, HEE, FK4, FK5, ECLIPTIC.

- <SpaceFrame> elements can only be read if the reference position is TOPOCENTER. Also, any planetary ephemeris is ignored.
- Regions: there is currently no support for STC regions of type Sector, ConvexHull or SkyIndex.
- The AST Region read from a CoordInterval element is considered to be open if either the lo_include or the hi_include attribute is set to false.
- <RegionFile> elements are not supported.
- Vertices within <Polygon> elements are always considered to be joined using great circles (that is, <SmallCircle> elements are ignored).

XmlLength Controls output buffer length XmlLength

Description: This attribute specifies the maximum length to use when writing out text through the sink function supplied when the XmlChan was created.

The number of characters in each string written out through the sink function will not be greater than the value of this attribute (but may be less). A value of zero (the default) means there is no limit - each string can be of any length.

Type:
Integer.

Class Applicability:

XmlChan
All XmlChans have this attribute.

XmlPrefix The namespace prefix to use when writing XmlPrefix

Description: This attribute is a string which is to be used as the namespace prefix for all XML elements created as a result of writing an AST Object out through an XmlChan. The URI associated with the namespace prefix is given by the symbolic constant AST__XMLNS defined in ast.h. A definition of the namespace prefix is included in each top-level element produced by the XmlChan.

The default value is a blank string which causes no prefix to be used. In this case each top-level element will set the default namespace to be the value of AST__XMLNS.

Type:
String.

Class Applicability:

Object
All Objects have this attribute.

Zoom ZoomMap scale factor Zoom

Description: This attribute holds the ZoomMap scale factor, by which coordinate values are multiplied (by the forward transformation) or divided (by the inverse transformation). This factor is set when a ZoomMap is created, but may later be modified. The default value is unity.

Note that if a ZoomMap is inverted (e.g. by using astInvert), then the reciprocal of this zoom factor will, in effect, be used.

Type:
Double precision.

Class Applicability:

ZoomMap

All ZoomMaps have this attribute.

Notes:

- The Zoom attribute may not be set to zero.

D AST Class Descriptions

Axis Store axis information Axis

Description: The Axis class is used to store information associated with a particular axis of a Frame. It is used internally by the AST library and has no constructor function. You should encounter it only within textual output (e.g. from astWrite).

Constructor Function:
None.

Inheritance:

The Axis class inherits from the Object class.

Box A box region with sides parallel to the axes of a Frame Box

Description: The Box class implements a Region which represents a box with sides parallel to the axes of a Frame (i.e. an area which encloses a given range of values on each axis). A Box is similar to an Interval, the only real difference being that the Interval class allows some axis limits to be unspecified. Note, a Box will only look like a box if the Frame geometry is approximately flat. For instance, a Box centred close to a pole in a SkyFrame will look more like a fan than a box (the Polygon class can be used to create a box-like region close to a pole).

Constructor Function:
astBox

Inheritance:

The Box class inherits from the Region class.

Attributes:

The Box class does not define any new attributes beyond those which are applicable to all Regions.

Functions:

The Box class does not define any new functions beyond those which are applicable to all Regions.

Channel Basic (textual) I/O channel Channel

Description: The Channel class implements low-level input/output for the AST library. Writing an Object to a Channel will generate a textual representation of that Object, and reading from a Channel will create a new Object from its textual representation.

Normally, when you use a Channel, you should provide "source" and "sink" functions which connect it to an external data store by reading and writing the resulting text. By default, however, a Channel will read from standard input and write to standard output. Alternatively, a Channel can be told to read or write from specific text files using the SinkFile and SourceFile attributes, in which case no sink or source function need be supplied.

Constructor Function:
astChannel

Inheritance:

The Channel class inherits from the Object class.

Attributes:

In addition to those attributes common to all Objects, every Channel also has the following attributes:

- Comment: Include textual comments in output?
- Full: Set level of output detail
- Indent: Indentation increment between objects
- ReportLevel: Selects the level of error reporting
- SinkFile: The path to a file to which the Channel should write
- Skip: Skip irrelevant data?
- SourceFile: The path to a file from which the Channel should read
- Strict: Generate errors instead of warnings?

Functions:

In addition to those functions applicable to all Objects, the following functions may also be applied to all Channels:

- astWarnings: Return warnings from the previous read or write
- astPutChannelData: Store data to pass to source or sink functions
- astRead: Read an Object from a Channel
- astWrite: Write an Object to a Channel

Circle A circular or spherical region within a Frame Circle

Description: The Circle class implements a Region which represents a circle or sphere within a Frame.

Constructor Function:
astCircle

Inheritance:

The Circle class inherits from the Region class.

Attributes:

The Circle class does not define any new attributes beyond those which are applicable to all Regions.

Functions:

In addition to those functions applicable to all Regions, the following functions may also be applied to all Circles:

- astCirclePars: Get the geometric parameters of the Circle
- AST_CIRCLEPARS: Get the geometric parameters of the Circle

CmpFrame Compound Frame **CmpFrame**

Description: A CmpFrame is a compound Frame which allows two component Frames (of any class) to be merged together to form a more complex Frame. The axes of the two component Frames then appear together in the resulting CmpFrame (those of the first Frame, followed by those of the second Frame).

Since a CmpFrame is itself a Frame, it can be used as a component in forming further CmpFrames. Frames of arbitrary complexity may be built from simple individual Frames in this way.

Also since a Frame is a Mapping, a CmpFrame can also be used as a Mapping. Normally, a CmpFrame is simply equivalent to a UnitMap, but if either of the component Frames within a CmpFrame is a Region (a sub-class of Frame), then the CmpFrame will use the Region as a Mapping when transforming values for axes described by the Region. Thus input axis values corresponding to positions which are outside the Region will result in bad output axis values.

Constructor Function:
astCmpFrame

Inheritance:

The CmpFrame class inherits from the Frame class.

Attributes:

The CmpFrame class does not define any new attributes beyond those which are applicable to all Frames. However, the attributes of the component Frames can be accessed as if they were attributes of the CmpFrame. For instance, if a CmpFrame contains a SpecFrame and a SkyFrame, then the CmpFrame will recognise the "Equinox" attribute and forward access requests to the component SkyFrame. Likewise, it will recognise the "RestFreq" attribute and forward access requests to the component SpecFrame. An axis index can optionally be appended to the end of any attribute name, in which case the request to access the attribute will be forwarded to the primary Frame defining the specified axis.

Functions:

The CmpFrame class does not define any new functions beyond those which are applicable to all Frames.

CmpMap Compound Mapping **CmpMap**

Description: A CmpMap is a compound Mapping which allows two component Mappings (of any class) to be connected together to form a more complex Mapping. This connection may either be "in series" (where the first Mapping is used to transform the coordinates of each point and the second mapping is then applied to the result), or "in parallel" (where one Mapping transforms the earlier coordinates for each point and the second Mapping simultaneously transforms the later coordinates).

Since a CmpMap is itself a Mapping, it can be used as a component in forming further CmpMaps. Mappings of arbitrary complexity may be built from simple individual Mappings in this way.

Constructor Function:
astCmpMap

Inheritance:

The CmpMap class inherits from the Mapping class.

Attributes:

The CmpMap class does not define any new attributes beyond those which are applicable to all Mappings.

Functions:

The CmpMap class does not define any new functions beyond those which are applicable to all Mappings.

CmpRegion A combination of two regions within a single Frame CmpRegion

Description: A CmpRegion is a Region which allows two component Regions (of any class) to be combined to form a more complex Region. This combination may be performed a boolean AND, OR or XOR (exclusive OR) operator. If the AND operator is used, then a position is inside the CmpRegion only if it is inside both of its two component Regions. If the OR operator is used, then a position is inside the CmpRegion if it is inside either (or both) of its two component Regions. If the XOR operator is used, then a position is inside the CmpRegion if it is inside one but not both of its two component Regions. Other operators can be formed by negating one or both component Regions before using them to construct a new CmpRegion.

The two component Region need not refer to the same coordinate Frame, but it must be possible for the astConvert function to determine a Mapping between them (an error will be reported otherwise when the CmpRegion is created). For instance, a CmpRegion may combine a Region defined within an ICRS SkyFrame with a Region defined within a Galactic SkyFrame. This is acceptable because the SkyFrame class knows how to convert between these two systems, and consequently the astConvert function will also be able to convert between them. In such cases, the second component Region will be mapped into the coordinate Frame of the first component Region, and the Frame represented by the CmpRegion as a whole will be the Frame of the first component Region.

Since a CmpRegion is itself a Region, it can be used as a component in forming further CmpRegions. Regions of arbitrary complexity may be built from simple individual Regions in this way.

Constructor Function:
astCmpRegion

Inheritance:

The CmpRegion class inherits from the Region class.

Attributes:

The CmpRegion class does not define any new attributes beyond those which are applicable to all Regions.

Functions:

The CmpRegion class does not define any new functions beyond those which are applicable to all Regions.

DSBSpecFrame Dual sideband spectral coordinate system description DSBSpecFrame

Description: A DSBSpecFrame is a specialised form of SpecFrame which represents positions in a spectrum obtained using a dual sideband instrument. Such an instrument produces a spectrum in which each point contains contributions from two distinctly different frequencies, one from the "lower side band" (LSB) and one from the "upper side band" (USB). Corresponding LSB and USB frequencies are connected by the fact that they are an equal distance on either side of a fixed central frequency known as the "Local Oscillator" (LO) frequency.

When quoting a position within such a spectrum, it is necessary to indicate whether the quoted position is the USB position or the corresponding LSB position. The SideBand attribute provides this indication. Another option that the SideBand attribute provides is to represent a spectral position by its topocentric offset from the LO frequency.

In practice, the LO frequency is specified by giving the distance from the LO frequency to some "central" spectral position. Typically this central position is that of some interesting spectral feature. The distance from this central position to the LO frequency is known as the "intermediate frequency" (IF). The value supplied for IF can be a signed value in order to indicate whether the LO frequency is above or below the central position.

Constructor Function:

astDSBSpecFrame

Inheritance:

The DSBSpecFrame class inherits from the SpecFrame class.

Attributes:

In addition to those attributes common to all SpecFrames, every DSBSpecFrame also has the following attributes:

- AlignSideBand: Should alignment occur between sidebands?
- DSBCentre: The central position of interest.
- IF: The intermediate frequency used to define the LO frequency.
- ImagFreq: The image sideband equivalent of the rest frequency.
- SideBand: Indicates which sideband the DSBSpecFrame represents.

Functions:

The DSBSpecFrame class does not define any new functions beyond those which are applicable to all SpecFrames.

DssMap Map points using a Digitised Sky Survey plate solution DssMap

Description: The DssMap class implements a Mapping which transforms between 2-dimensional pixel coordinates and an equatorial sky coordinate system (right ascension and declination) using a Digitised Sky Survey (DSS) astrometric plate solution.

The input coordinates are pixel numbers along the first and second dimensions of an image, where the centre of the first pixel is located at (1,1) and the spacing between pixel centres is unity.

The output coordinates are right ascension and declination in radians. The celestial coordinate system used (FK4, FK5, etc.) is unspecified, and will usually be indicated by appropriate keywords in a FITS header.

Constructor Function:

The DssMap class does not have a constructor function. A DssMap is created only as a by-product of reading a FrameSet (using astRead) from a FitsChan which contains FITS header cards describing a DSS plate solution, and whose Encoding attribute is set to "DSS". The result of such a read, if successful, is a FrameSet whose base and current Frames are related by a DssMap.

Inheritance:

The DssMap class inherits from the Mapping class.

Attributes:

The DssMap class does not define any new attributes beyond those which are applicable to all Mappings.

Functions:

The DssMap class does not define any new functions beyond those which are applicable to all Mappings.

Ellipse An elliptical region within a 2-dimensional Frame Ellipse

Description: The Ellipse class implements a Region which represents a ellipse within a 2-dimensional Frame.

Constructor Function:
astEllipse

Inheritance:

The Ellipse class inherits from the Region class.

Attributes:

The Ellipse class does not define any new attributes beyond those which are applicable to all Regions.

Functions:

In addition to those functions applicable to all Regions, the following functions may also be applied to all Ellipses:

- astEllipsePars: Get the geometric parameters of the Ellipse
- AST_ELLIPSEPARS: Get the geometric parameters of the Ellipse

---

FitsChan I/O Channel using FITS header cards to represent Objects FitsChan

Description: A FitsChan is a specialised form of Channel which supports I/O operations involving the use of FITS (Flexible Image Transport System) header cards. Writing an Object to a FitsChan (using astWrite) will, if the Object is suitable, generate a description of that Object composed of FITS header cards, and reading from a FitsChan will create a new Object from its FITS header card description.

While a FitsChan is active, it represents a buffer which may contain zero or more 80-character "header cards" conforming to FITS conventions. Any sequence of FITS-conforming header cards may be stored, apart from the "END" card whose existence is merely implied. The cards may be accessed in any order by using the FitsChan's integer Card attribute, which identifies a "current" card, to which subsequent operations apply. Searches based on keyword may be performed (using astFindFits), new cards may be inserted (astPutFits, astPutCards, astSetFits<X>) and existing ones may be deleted (astDelFits), extracted (astGetFits<X>), or changed (astSetFits<X>).

When you create a FitsChan, you have the option of specifying "source" and "sink" functions which connect it to external data stores by reading and writing FITS header cards. If you provide a source function, it is used to fill the FitsChan with header cards when it is accessed for the first time. If you do not provide a source function, the FitsChan remains empty until you explicitly enter data into it (e.g. using astPutFits, astPutCards, astWrite or by using the SourceFile attribute to specifying a text file from which headers should be read). When the FitsChan is deleted, any remaining header cards in the FitsChan can be saved in either of two ways: 1) by specifying a value for the SinkFile attribute (the name of a text file to which header cards should be written), or 2) by providing a sink function (used to to deliver header cards to an external data store). If you do not provide a sink function or a value for SinkFile, any header cards remaining when the FitsChan is deleted will be lost, so you should arrange to extract them first if necessary (e.g. using astFindFits or astRead).

Coordinate system information may be described using FITS header cards using several different conventions, termed "encodings". When an AST Object is written to (or read from) a FitsChan, the value of the FitsChan's Encoding attribute determines how the Object is converted to (or from) a description involving FITS header cards. In general, different encodings will result in different sets of header cards to describe the same Object. Examples of encodings include the DSS encoding (based on conventions used by the STScI Digitised Sky Survey data), the FITS-WCS encoding (based on a proposed FITS standard) and the NATIVE encoding (a near loss-less way of storing AST Objects in FITS headers).

The available encodings differ in the range of Objects they can represent, in the number of Object descriptions that can coexist in the same FitsChan, and in their accessibility to other (external) astronomy applications (see the Encoding attribute for details). Encodings are not necessarily mutually exclusive and it may sometimes be possible to describe the same Object in several ways within a particular set of FITS header cards by using several different encodings.

The detailed behaviour of astRead and astWrite, when used with a FitsChan, depends on the encoding in use. In general, however, all successful use of astRead is destructive, so that FITS header cards are consumed in the process of reading an Object, and are removed from the FitsChan (this deletion can be prevented for specific cards by calling the astRetainFits function). An unsuccessful call of astRead (for instance, caused by the FitsChan not containing the necessary FITS headers cards needed to create an Object) results in the contents of the FitsChan being left unchanged.

If the encoding in use allows only a single Object description to be stored in a FitsChan (e.g. the DSS, FITS-WCS and FITS-IRAF encodings), then write operations using astWrite will over-write any existing Object description using that encoding. Otherwise (e.g. the NATIVE encoding), multiple Object descriptions are written sequentially and may later be read back in the same sequence.

Constructor Function:

astFitsChan

Inheritance:

The FitsChan class inherits from the Channel class.

Attributes:

In addition to those attributes common to all Channels, every

FitsChan also has the following attributes:

- AllWarnings: A list of the available conditions
- Card: Index of current FITS card in a FitsChan
- CardType: The data type of the current FITS card in a FitsChan
- CarLin: Ignore spherical rotations on CAR projections?
- CDMatrix: Use a CD matrix instead of a PC matrix?
- Clean: Remove cards used whilst reading even if an error occurs?
- DefB1950: Use FK4 B1950 as default equatorial coordinates?
- Encoding: System for encoding Objects as FITS headers
- FitsDigits: Digits of precision for floating-point FITS values
- Iwc: Add a Frame describing Intermediate World Coords?
- Ncard: Number of FITS header cards in a FitsChan
- Nkey: Number of unique keywords in a FitsChan

- TabOK: Should the FITS "-TAB" algorithm be recognised?
- PolyTan: Use PVi_m keywords to define distorted TAN projection?
- Warnings: Produces warnings about selected conditions

Functions:

In addition to those functions applicable to all Channels, the following functions may also be applied to all FitsChans:

- astDelFits: Delete the current FITS card in a FitsChan
- astEmptyFits: Delete all cards in a FitsChan
- astFindFits: Find a FITS card in a FitsChan by keyword
- astGetFits<X>: Get a keyword value from a FitsChan
- astGetTables: Retrieve any FitsTables from a FitsChan
- astPurgeWCS: Delete all WCS-related cards in a FitsChan
- astPutCards: Stores a set of FITS header card in a FitsChan
- astPutFits: Store a FITS header card in a FitsChan
- astPutTable: Store a single FitsTable in a FitsChan
- astPutTables: Store multiple FitsTables in a FitsChan
- astReadFits: Read cards in through the source function
- astRemoveTables: Remove one or more FitsTables from a FitsChan
- astRetainFits: Ensure current card is retained in a FitsChan
- astSetFits<X>: Store a new keyword value in a FitsChan
- astTableSource: Register a source function for FITS table access
- astTestFits: Test if a keyword has a defined value in a FitsChan
- astWriteFits: Write all cards out to the sink function

FitsTable A representation of a FITS binary table FitsTable

Description: The FitsTable class is a representation of a FITS binary table. It inherits from the Table class. The parent Table is used to hold the binary data of the main table, and a FitsChan (encapsulated within the FitsTable) is used to hold the FITS header.

Note - it is not recommended to use the FitsTable class to store very large tables.

FitsTables are primarily geared towards the needs of the "-TAB" algorithm defined in FITS-WCS paper 2, and so do not support all features of FITS binary tables. In particularly, they do not provide any equivalent to the following features of FITS binary tables: "heap" data (i.e. binary data following the main table), columns holding complex values, columns holding variable length arrays, scaled columns, column formats, columns holding bit values, 8-byte integer values or logical values.

Constructor Function:
astFitsTable

Inheritance:

The FitsTable class inherits from the Table class.

Attributes:

The FitsTable class does not define any new attributes beyond those which are applicable to all Tables.

Functions:

In addition to those functions applicable to all Tables, the following functions may also be applied to all FitsTables:

- astColumnNull: Get/set the null value for a column of a FitsTable
- astColumnSize: Get number of bytes needed to hold a full column of data
- astGetColumnData: Retrieve all the data values stored in a column
- astGetTableHeader: Get the FITS headers from a FitsTable
- astPutColumnData: Store data values in a column
- astPutTableHeader: Store FITS headers within a FitsTable

FluxFrame Measured flux description FluxFrame

Description: A FluxFrame is a specialised form of one-dimensional Frame which represents various systems used to represent the signal level in an observation. The particular coordinate system to be used is specified by setting the FluxFrame's System attribute qualified, as necessary, by other attributes such as the units, etc (see the description of the System attribute for details).

All flux values are assumed to be measured at the same frequency or wavelength (as given by the SpecVal attribute). Thus this class is more appropriate for use with images rather than spectra.

Constructor Function:
astFluxFrame

Inheritance:

The FluxFrame class inherits from the Frame class.

Attributes:

In addition to those attributes common to all Frames, every FluxFrame also has the following attributes:

- SpecVal: The spectral position at which the flux values are measured.

Functions:

The FluxFrame class does not define any new functions beyond those which are applicable to all Frames.

Frame Coordinate system description Frame

Description: This class is used to represent coordinate systems. It does this in rather the same way that a frame around a graph describes the coordinate space in which data are plotted. Consequently, a Frame has a Title (string) attribute, which describes the coordinate space, and contains axes which in turn hold information such as Label and Units strings which are used for labelling (e.g.) graphical output. In general, however, the number of axes is not restricted to two.

Functions are available for converting Frame coordinate values into a form suitable for display, and also for calculating distances and offsets between positions within the Frame.

Frames may also contain knowledge of how to transform to and from related coordinate systems.

Constructor Function:
astFrame

Notes:

- When used as a Mapping, a Frame implements a unit (null) transformation in both the forward and inverse directions (equivalent to a UnitMap). The Nin and Nout attribute values are both equal to the number of Frame axes.

Inheritance:

The Frame class inherits from the Mapping class.

Attributes:

In addition to those attributes common to all Mappings, every Frame also has the following attributes (if the Frame has only one axis, the axis specifier can be omited from the following attribute names):

- AlignSystem: Coordinate system used to align Frames
- Bottom(axis): Lowest axis value to display
- Digits/Digits(axis): Number of digits of precision
- Direction(axis): Display axis in conventional direction?
- Domain: Coordinate system domain
- Dut1: Difference between the UT1 and UTC timescale
- Epoch: Epoch of observation
- Format(axis): Format specification for axis values
- Label(axis): Axis label
- MatchEnd: Match trailing axes?
- MaxAxes: Maximum number of Frame axes to match
- MinAxes: Minimum number of Frame axes to match
- Naxes: Number of Frame axes
- NormUnit(axis): Normalised axis physical units
- ObsAlt: Geodetic altitude of observer
- ObsLat: Geodetic latitude of observer
- ObsLon: Geodetic longitude of observer
- Permute: Permute axis order?
- PreserveAxes: Preserve axes?
- Symbol(axis): Axis symbol
- System: Coordinate system used to describe the domain
- Title: Frame title
- Top(axis): Highest axis value to display
- Unit(axis): Axis physical units

Functions:

In addition to those functions applicable to all Mappings, the following functions may also be applied to all Frames:

- astAngle: Calculate the angle subtended by two points at a third point
- astAxAngle: Find the angle from an axis, to a line through two points

- astAxDistance: Calculate the distance between two axis values
- astAxOffset: Calculate an offset along an axis
- astConvert: Determine how to convert between two coordinate systems
- astDistance: Calculate the distance between two points in a Frame
- astFindFrame: Find a coordinate system with specified characteristics
- astFormat: Format a coordinate value for a Frame axis
- astGetActiveUnit: Determines how the Unit attribute will be used
- astIntersect: Find the intersection between two geodesic curves
- astMatchAxes: Find any corresponding axes in two Frames
- astNorm: Normalise a set of Frame coordinates
- astOffset: Calculate an offset along a geodesic curve
- astOffset2: Calculate an offset along a geodesic curve in a 2D Frame
- astPermAxes: Permute the order of a Frame's axes
- astPickAxes: Create a new Frame by picking axes from an existing one
- astResolve: Resolve a vector into two orthogonal components
- astSetActiveUnit: Specify how the Unit attribute should be used
- astUnformat: Read a formatted coordinate value for a Frame axis

---

FrameSet Set of inter-related coordinate systems FrameSet

Description: A FrameSet consists of a set of one or more Frames (which describe coordinate systems), connected together by Mappings (which describe how the coordinate systems are inter-related). A FrameSet makes it possible to obtain a Mapping between any pair of these Frames (i.e. to convert between any of the coordinate systems which it describes). The individual Frames are identified within the FrameSet by an integer index, with Frames being numbered consecutively from one as they are added to the FrameSet.

Every FrameSet has a "base" Frame and a "current" Frame (which are allowed to be the same). Any of the Frames may be nominated to hold these positions, and the choice is determined by the values of the FrameSet's Base and Current attributes, which hold the indices of the relevant Frames. By default, the first Frame added to a FrameSet is its base Frame, and the last one added is its current Frame.

The base Frame describes the "native" coordinate system of whatever the FrameSet is used to calibrate (e.g. the pixel coordinates of an image) and the current Frame describes the "apparent" coordinate system in which it should be viewed (e.g. displayed, etc.). Any further Frames represent a library of alternative coordinate systems, which may be selected by making them current.

When a FrameSet is used in a context that requires a Frame, (e.g. obtaining its Title value, or number of axes), the current Frame is used. A FrameSet may therefore be used in place of its current Frame in most situations.

When a FrameSet is used in a context that requires a Mapping, the Mapping used is the one between its base Frame and its current Frame. Thus, a FrameSet may be used to convert "native" coordinates into "apparent" ones, and vice versa. Like any Mapping, a FrameSet may also be inverted (see astInvert), which has the effect of interchanging its base and current Frames and hence of reversing the Mapping between them.

Regions may be added into a FrameSet (since a Region is a type of Frame), either explicitly or as components within CmpFrames. In this case the Mapping between a pair of Frames within a FrameSet will include the effects of the clipping produced by any Regions included in the path between the Frames.

Constructor Function:
astFrameSet

Inheritance:

The FrameSet class inherits from the Frame class.

Attributes:

In addition to those attributes common to all Frames, every FrameSet also has the following attributes:

- AllVariants: List of all variant mappings store with current Frame
- Base: FrameSet base Frame index
- Current: FrameSet current Frame index
- Nframe: Number of Frames in a FrameSet
- Variant: Name of variant mapping in use by current Frame

Every FrameSet also inherits any further attributes that belong to its current Frame, regardless of that Frame's class. (For example, the Equinox attribute, defined by the SkyFrame class, is inherited by any FrameSet which has a SkyFrame as its current Frame.) The set of attributes belonging to a FrameSet may therefore change when a new current Frame is selected.

Functions:

In addition to those functions applicable to all Frames, the following functions may also be applied to all FrameSets:

- astAddFrame: Add a Frame to a FrameSet to define a new coordinate system
- astAddVariant: Add a variant Mapping to the current Frame
- astGetFrame: Obtain a pointer to a specified Frame in a FrameSet
- astGetMapping: Obtain a Mapping between two Frames in a FrameSet
- astMirrorVariants: Make the current Frame mirror variant Mappings in another Frame
- astRemapFrame: Modify a Frame's relationship to the other Frames in a FrameSet
- astRemoveFrame: Remove a Frame from a FrameSet

GrismMap Transform 1-dimensional coordinates using a grism dispersion equation GrismMap

Description: A GrismMap is a specialised form of Mapping which transforms 1-dimensional coordinates using the spectral dispersion equation described in FITS-WCS paper III "Representation of spectral coordinates in FITS". This describes the dispersion produced by gratings, prisms and grisms.

When initially created, the forward transformation of a GrismMap transforms input "grism parameter" values into output wavelength values. The "grism parameter" is a dimensionless value which is linearly related to position on the detector. It is defined in FITS-WCS paper III as "the offset on the detector from the point of intersection of the camera axis, measured in units of the effective local length". The units in which wavelength values are expected or returned is determined by the values supplied for the GrismWaveR, GrismNRP and GrismG attribute: whatever units are used for these attributes will also be used for the wavelength values.

Constructor Function:
astGrismMap

Inheritance:

The GrismMap class inherits from the Mapping class.

Attributes:

In addition to those attributes common to all Mappings, every GrismMap also has the following attributes:

- GrismNR: The refractive index at the reference wavelength
- GrismNRP: Rate of change of refractive index with wavelength
- GrismWaveR: The reference wavelength
- GrismAlpha: The angle of incidence of the incoming light
- GrismG: The grating ruling density
- GrismM: The interference order
- GrismEps: The angle between the normal and the dispersion plane
- GrismTheta: Angle between normal to detector plane and reference ray

Functions:

The GrismMap class does not define any new functions beyond those which are applicable to all Mappings.

Interval A region representing an interval on one or more axes of a Frame Interval

Description: The Interval class implements a Region which represents upper and/or lower limits on one or more axes of a Frame. For a point to be within the region represented by the Interval, the point must satisfy all the restrictions placed on all the axes. The point is outside the region if it fails to satisfy any one of the restrictions. Each axis may have either an upper limit, a lower limit, both or neither. If both limits are supplied but are in reverse order (so that the lower limit is greater than the upper limit), then the interval is an excluded interval, rather than an included interval.

Note, The Interval class makes no allowances for cyclic nature of some coordinate systems (such as SkyFrame coordinates). A Box should usually be used in these cases since this requires the user to think about suitable upper and lower limits,

Constructor Function:
astInterval

Inheritance:

The Interval class inherits from the Region class.

Attributes:

The Interval class does not define any new attributes beyond those which are applicable to all Regions.

Functions:

The Interval class does not define any new functions beyond those which are applicable to all Regions.

IntraMap Map points using a private transformation function IntraMap

Description: The IntraMap class provides a specialised form of Mapping which encapsulates a privately-defined coordinate transformation other AST Mapping. This allows you to create Mappings that perform any conceivable coordinate transformation.

However, an IntraMap is intended for use within a single program or a private suite of software, where all programs have access to the same coordinate transformation functions (i.e. can be linked against them). IntraMaps should not normally be stored in datasets which may be exported for processing by other software, since that software will not have the necessary transformation functions available, resulting in an error.

You must register any coordinate transformation functions to be used using astIntraReg before creating an IntraMap.

Constructor Function:
astIntraMap (also see astIntraReg)

Inheritance:

The IntraMap class inherits from the Mapping class.

Attributes:

In addition to those attributes common to all Mappings, every IntraMap also has the following attributes:

- IntraFlag: IntraMap identification string

Functions:

The IntraMap class does not define any new functions beyond those which are applicable to all Mappings.

KeyMap Store a set of key/value pairs KeyMap

Description: The KeyMap class is used to store a set of values with associated keys which identify the values. The keys are strings. These may be case sensitive or insensitive as selected by the KeyCase attribute, and trailing spaces are ignored. The value associated with a key can be integer (signed 4 and 2 byte, or unsigned 1 byte), floating point (single or double precision), void pointer, character string or AST Object pointer. Each value can be a scalar or a one-dimensional vector. A KeyMap is conceptually similar to a Mapping in that a KeyMap transforms an input into an output - the input is the key, and the output is the value associated with the key. However, this is only a conceptual similarity, and it should be noted that the KeyMap class inherits from the Object class rather than the Mapping class. The methods of the Mapping class cannot be used with a KeyMap.

Constructor Function:
astKeyMap

Inheritance:

The KeyMap class inherits from the Object class.

Attributes:

In addition to those attributes common to all Objects, every KeyMap also has the following attributes:

- KeyCase: Sets the case in which keys are stored
- KeyError: Report an error if the requested key does not exist?
- SizeGuess: The expected size of the KeyMap.
- SortBy: Determines how keys are sorted in a KeyMap.
- MapLocked: Prevent new entries being added to the KeyMap?

Functions:

In addition to those functions applicable to all Objects, the following functions may also be applied to all KeyMaps:

- astMapDefined: Does a KeyMap contain a defined value for a key?
- astMapGet0<X>: Get a named scalar entry from a KeyMap
- astMapGet1<X>: Get a named vector entry from a KeyMap
- astMapGetElem<X>: Get an element of a named vector entry from a KeyMap
- astMapHasKey: Does the KeyMap contain a named entry?
- astMapKey: Return the key name at a given index in the KeyMap
- astMapLenC: Get the length of a named character entry in a KeyMap
- astMapLength: Get the length of a named entry in a KeyMap
- astMapCopy: Copy entries from one KeyMap into another
- astMapPut0<X>: Add a new scalar entry to a KeyMap
- astMapPut1<X>: Add a new vector entry to a KeyMap
- astMapPutElem<X>: Puts a value into a vector entry in a KeyMap
- astMapPutU: Add a new entry to a KeyMap with an undefined value
- astMapRemove: Removed a named entry from a KeyMap
- astMapRename: Rename an existing entry in a KeyMap
- astMapSize: Get the number of entries in a KeyMap
- astMapType: Return the data type of a named entry in a map

LutMap Transform 1-dimensional coordinates using a lookup table LutMap

Description: A LutMap is a specialised form of Mapping which transforms 1-dimensional coordinates by using linear interpolation in a lookup table.

Each input coordinate value is first scaled to give the index of an entry in the table by subtracting a starting value (the input coordinate corresponding to the first table entry) and dividing by an increment (the difference in input coordinate value between adjacent table entries).

The resulting index will usually contain a fractional part, so the output coordinate value is then generated by interpolating linearly between the appropriate entries in the table. If the index lies outside the range of the table, linear extrapolation is used based on the two nearest entries (i.e. the two entries at the start or end of the table, as appropriate). If either of the entries used for the interplation has a value of AST__BAD, then the interpolated value is returned as AST__BAD.

If the lookup table entries increase or decrease monotonically (ignoring any flat sections), then the inverse transformation may also be performed.

Constructor Function:

astLutMap

Inheritance:

The LutMap class inherits from the Mapping class.

Attributes:

In addition to those attributes common to all Mappings, every LutMap also has the following attributes:

- LutInterp: The interpolation method to use between table entries.

Functions:

The LutMap class does not define any new functions beyond those which are applicable to all Mappings.

Mapping Inter-relate two coordinate systems Mapping

Description: This class provides the basic facilities for transforming a set of coordinates (representing "input" points) to give a new set of coordinates (representing "output" points). It is used to describe the relationship which exists between two different coordinate systems and to implement operations which make use of this (such as transforming coordinates and resampling grids of data). However, the Mapping class does not have a constructor function of its own, as it is simply a container class for a family of specialised Mappings which implement particular types of coordinate transformation.

Constructor Function:
None.

Inheritance:

The Mapping class inherits from the Object class.

Attributes:

In addition to those attributes common to all Objects, every Mapping also has the following attributes:

- Invert: Mapping inversion flag
- IsLinear: Is the Mapping linear?
- IsSimple: Has the Mapping been simplified?
- Nin: Number of input coordinates for a Mapping
- Nout: Number of output coordinates for a Mapping
- Report: Report transformed coordinates?
- TranForward: Forward transformation defined?
- TranInverse: Inverse transformation defined?

Functions:

In addition to those functions applicable to all Objects, the following functions may also be applied to all Mappings:

- astDecompose: Decompose a Mapping into two component Mappings
- astTranGrid: Transform a grid of positions
- astInvert: Invert a Mapping

- astLinearApprox: Calculate a linear approximation to a Mapping
- astMapBox: Find a bounding box for a Mapping
- astMapSplit: Split a Mapping up into parallel component Mappings
- astQuadApprox: Calculate a quadratic approximation to a 2D Mapping
- astRate: Calculate the rate of change of a Mapping output
- astRebin<X>: Rebin a region of a data grid
- astResample<X>: Resample a region of a data grid
- astRemoveRegions: Remove any Regions from a Mapping
- astSimplify: Simplify a Mapping
- astTran1: Transform 1-dimensional coordinates
- astTran2: Transform 2-dimensional coordinates
- astTranN: Transform N-dimensional coordinates
- astTranP: Transform N-dimensional coordinates held in separate arrays

MathMap Transform coordinates using mathematical expressions MathMap

Description: A MathMap is a Mapping which allows you to specify a set of forward and/or inverse transformation functions using arithmetic operations and mathematical functions similar to those available in C. The MathMap interprets these functions at run-time, whenever its forward or inverse transformation is required. Because the functions are not compiled in the normal sense (unlike an IntraMap), they may be used to describe coordinate transformations in a transportable manner. A MathMap therefore provides a flexible way of defining new types of Mapping whose descriptions may be stored as part of a dataset and interpreted by other programs.

Constructor Function:
astMathMap

Inheritance:

The MathMap class inherits from the Mapping class.

Attributes:

In addition to those attributes common to all Mappings, every MathMap also has the following attributes:

- Seed: Random number seed
- SimpFI: Forward-inverse MathMap pairs simplify?
- SimpIF: Inverse-forward MathMap pairs simplify?

Functions:

The MathMap class does not define any new functions beyond those which are applicable to all Mappings.

MatrixMap Map coordinates by multiplying by a matrix MatrixMap

Description: A MatrixMap is form of Mapping which performs a general linear transformation. Each set of input coordinates, regarded as a column-vector, are pre-multiplied by a matrix (whose elements are specified when the MatrixMap is created) to give a new column-vector containing the output coordinates. If appropriate, the inverse transformation may also be performed.

Constructor Function:
astMatrixMap

Inheritance:

The MatrixMap class inherits from the Mapping class.

Attributes:

The MatrixMap class does not define any new attributes beyond those which are applicable to all Mappings.

Functions:

The MatrixMap class does not define any new functions beyond those which are applicable to all Mappings.

NormMap Normalise coordinates using a supplied Frame NormMap

Description: The NormMap class implements a Mapping which normalises coordinate values using the astNorm function of a supplied Frame. The number of inputs and outputs of a NormMap are both equal to the number of axes in the supplied Frame.

The forward and inverse transformation of a NormMap are both defined but are identical (that is, they do not form a real inverse pair in that the inverse transformation does not undo the normalisation, instead it reapplies it). However, the astSimplify function will replace neighbouring pairs of forward and inverse NormMaps by a single UnitMap.

Constructor Function:
astNormMap

Inheritance:

The NormMap class inherits from the Mapping class.

Attributes:

The NormMap class does not define any new attributes beyond those which are applicable to all Mappings.

Functions:

The NormMap class does not define any new functions beyond those which are applicable to all Mappings.

NullRegion A boundless region within a Frame NullRegion

Description: The NullRegion class implements a Region with no bounds within a Frame. If the Negated attribute of a NullRegion is false, the NullRegion represents a Region containing no points. If the Negated attribute of a NullRegion is true, the NullRegion represents an infinite Region (that is, all points in the coordinate system are inside the NullRegion).

Constructor Function:
astNullRegion

Inheritance:

The NullRegion class inherits from the Region class.

Attributes:

The NullRegion class does not define any new attributes beyond those which are applicable to all Regions.

Functions:

The NullRegion class does not define any new functions beyond those which are applicable to all Regions.

Object Base class for all AST Objects Object

Description: This class is the base class from which all other classes in the AST library are derived. It provides all the basic Object behaviour and Object manipulation facilities required throughout the library. There is no Object constructor, however, as Objects on their own are not useful.

Constructor Function:
None.

Inheritance:

The Object base class does not inherit from any other class.

Attributes:

All Objects have the following attributes:

- Class: Object class name
- ID: Object identification string
- Ident: Permanent Object identification string
- Nobject: Number of Objects in class
- ObjSize: The in-memory size of the Object in bytes
- RefCount: Count of active Object pointers
- UseDefs: Allow use of default values for Object attributes?

Functions:

The following functions may be applied to all Objects:

- astAnnul: Annul a pointer to an Object
- astBegin: Begin a new AST context
- astClear: Clear attribute values for an Object
- astClone: Clone a pointer to an Object
- astCopy: Copy an Object
- astDelete: Delete an Object
- astEnd: End an AST context
- astEscapes: Control whether graphical escape sequences are removed

- astExempt: Exempt an Object pointer from AST context handling
- astExport: Export an Object pointer to an outer context
- astGet<X>: Get an attribute value for an Object
- astHasAttribute: Test if an Object has a named attribute
- astImport: Import an Object pointer to the current context
- astIsA<Class>: Test class membership
- astLock: Lock an Object for use by the calling thread
- astToString: Create an in-memory serialisation of an Object
- astSame: Do two AST pointers refer to the same Object?
- astSet: Set attribute values for an Object
- astSet<X>: Set an attribute value for an Object
- astShow: Display a textual representation of an Object on standard output
- astTest: Test if an attribute value is set for an Object
- astTune: Set or get an integer AST tuning parameter
- astTuneC: Set or get a character AST tuning parameter
- astUnlock: Unlock an Object for use by other threads
- astFromString: Re-create an Object from an in-memory serialisation
- astVersion: Return the verson of the AST library being used.

PcdMap Apply 2-dimensional pincushion/barrel distortion PcdMap

Description: A PcdMap is a non-linear Mapping which transforms 2-dimensional positions to correct for the radial distortion introduced by some cameras and telescopes. This can take the form either of pincushion or barrel distortion, and is characterized by a single distortion coefficient.

A PcdMap is specified by giving this distortion coefficient and the coordinates of the centre of the radial distortion. The forward transformation of a PcdMap applies the distortion:

RD = R ∗ (1 + C ∗ R ∗ R)

where R is the undistorted radial distance from the distortion centre (specified by attribute PcdCen), RD is the radial distance from the same centre in the presence of distortion, and C is the distortion coefficient (given by attribute Disco).

The inverse transformation of a PcdMap removes the distortion produced by the forward transformation. The expression used to derive R from RD is an approximate inverse of the expression above.

Constructor Function:

astPcdMap

Inheritance:

The PcdMap class inherits from the Mapping class.

Attributes:

In addition to those attributes common to all Mappings, every PcdMap also has the following attributes:

- Disco: PcdMap pincushion/barrel distortion coefficient
- PcdCen(axis): Centre coordinates of pincushion/barrel distortion

Functions:

The PcdMap class does not define any new functions beyond those which are applicable to all Mappings.

PermMap Coordinate permutation Mapping PermMap

Description: A PermMap is a Mapping which permutes the order of coordinates, and possibly also changes the number of coordinates, between its input and output.

In addition to permuting the coordinate order, a PermMap may also assign constant values to coordinates. This is useful when the number of coordinates is being increased as it allows fixed values to be assigned to any new ones.

Constructor Function:
astPermMap

Inheritance:

The PermMap class inherits from the Mapping class.

Attributes:

The PermMap class does not define any new attributes beyond those which are applicable to all Mappings.

Functions:

The PermMap class does not define any new functions beyond those which are applicable to all Mappings.

Plot Provide facilities for 2D graphical output Plot

Description: This class provides facilities for producing 2D graphical output. A Plot is a specialised form of FrameSet, in which the base Frame describes a "graphical" coordinate system and is associated with a rectangular plotting area in the underlying graphics system. This plotting area is where graphical output appears. It is defined when the Plot is created.

The current Frame of a Plot describes a "physical" coordinate system, which is the coordinate system in which plotting operations are specified. The results of each plotting operation are automatically transformed into graphical coordinates so as to appear in the plotting area (subject to any clipping which may be in effect).

Because the Mapping between physical and graphical coordinates may often be non-linear, or even discontinuous, most plotting does not result in simple straight lines. The basic plotting element is therefore not a straight line, but a geodesic curve (see astCurve, astGenCurve and astPolyCurve). A Plot also provides facilities for drawing markers or symbols (astMark), text (astText) and grid lines (astGridLine). It is also possible to draw curvilinear axes with optional coordinate grids (astGrid). A range of Plot attributes is available to allow precise control over the appearance of graphical output produced by these functions.

You may select different physical coordinate systems in which to plot (including the native graphical coordinate system itself) by selecting different Frames as the current Frame of a Plot, using its Current attribute. You may also set up clipping (see astClip) to limit the extent of any plotting you perform, and this may be done in any of the coordinate systems associated with the Plot, not necessarily the one you are plotting in.

Like any FrameSet, a Plot may also be used as a Frame. In this case, it behaves like its current Frame, which describes the physical coordinate system.

When used as a Mapping, a Plot describes the inter-relation between graphical coordinates (its base Frame) and physical coordinates (its current Frame). It differs from a normal FrameSet, however, in that an attempt to transform points which lie in clipped areas of the Plot will result in bad coordinate values (AST__BAD).

Constructor Function:

astPlot

Inheritance:

The Plot class inherits from the FrameSet class.

Attributes:

In addition to those attributes common to all FrameSets, every Plot also has the following attributes:

- Abbrev: Abbreviate leading fields?
- Border: Draw a border around valid regions of a Plot?
- Clip: Clip lines and/or markers at the Plot boundary?
- ClipOp: Combine Plot clipping limits using a boolean OR?
- Colour(element): Colour index for a Plot element
- DrawAxes(axis): Draw axes for a Plot?
- DrawTitle: Draw a title for a Plot?
- Escape: Allow changes of character attributes within strings?
- Edge(axis): Which edges to label in a Plot
- Font(element): Character font for a Plot element
- Gap(axis): Interval between linearly spaced major axis values
- Grf: Select the graphics interface to use.
- Grid: Draw grid lines for a Plot?
- Invisible: Draw graphics in invisible ink?
- LabelAt(axis): Where to place numerical labels for a Plot
- LabelUnits(axis): Use axis unit descriptions in a Plot?
- LabelUp(axis): Draw numerical Plot labels upright?
- Labelling: Label and tick placement option for a Plot
- LogGap(axis): Interval between logarithmically spaced major axis values
- LogPlot(axis): Map the plot onto the screen logarithmically?
- LogTicks(axis): Space the major tick marks logarithmically?
- MajTickLen(axis): Length of major tick marks for a Plot
- MinTickLen(axis): Length of minor tick marks for a Plot
- MinTick(axis): Density of minor tick marks for a Plot
- NumLab(axis): Draw numerical axis labels for a Plot?
- NumLabGap(axis): Spacing of numerical axis labels for a Plot
- Size(element): Character size for a Plot element
- Style(element): Line style for a Plot element
- TextLab(axis): Draw descriptive axis labels for a Plot?
- TextLabGap(axis): Spacing of descriptive axis labels for a Plot

- TickAll: Draw tick marks on all edges of a Plot?
- TitleGap: Vertical spacing for a Plot title
- Tol: Plotting tolerance
- Width(element): Line width for a Plot element

Functions:

In addition to those functions applicable to all FrameSets, the following functions may also be applied to all Plots:

- astBBuf: Begin a new graphical buffering context
- astBorder: Draw a border around valid regions of a Plot
- astBoundingBox: Returns a bounding box for previously drawn graphics
- astClip: Set up or remove clipping for a Plot
- astCurve: Draw a geodesic curve
- astEBuf: End the current graphical buffering context
- astGenCurve: Draw a generalized curve
- astGetGrfContext: Get the graphics context for a Plot
- astGrfPop: Retrieve previously saved graphics functions
- astGrfPush: Save the current graphics functions
- astGrfSet: Register a graphics routine for use by a Plot
- astGrid: Draw a set of labelled coordinate axes
- astGridLine: Draw a grid line (or axis) for a Plot
- astMark: Draw a set of markers for a Plot
- astPolyCurve: Draw a series of connected geodesic curves
- astText: Draw a text string for a Plot

Graphical Elements:

The colour index, character font, character size, line style and line width used for plotting can be set independently for various elements of the graphical output produced by a Plot. The different graphical elements are identified by appending the strings listed below as subscripts to the Plot attributes Colour(element), Font(element), Size(element), Style(element) and Width(element). These strings are case-insensitive and unambiguous abbreviations may be used. Elements of the graphical output which relate to individual axes can be referred to either independently (e.g. `"(Grid1)"` and `"(Grid2)"`) or together (e.g. `"(Grid)"`):

- Axes: Axis lines drawn through tick marks using astGrid
- Axis1: Axis line drawn through tick marks on axis 1 using astGrid
- Axis2: Axis line drawn through tick marks on axis 2 using astGrid
- Border: The Plot border drawn using astBorder or astGrid
- Curves: Geodesic curves drawn using astCurve, astGenCurve or astPolyCurve
- Grid: Grid lines drawn using astGridLine or astGrid
- Grid1: Grid lines which cross axis 1, drawn using astGridLine or astGrid
- Grid2: Grid lines which cross axis 2, drawn using astGridLine or astGrid
- Markers: Graphical markers (symbols) drawn using astMark
- NumLab: Numerical axis labels drawn using astGrid

- NumLab1: Numerical labels for axis 1 drawn using astGrid
- NumLab2: Numerical labels for axis 2 drawn using astGrid
- Strings: Text strings drawn using astText
- TextLab: Descriptive axis labels drawn using astGrid
- TextLab1: Descriptive label for axis 1 drawn using astGrid
- TextLab2: Descriptive label for axis 2 drawn using astGrid
- Ticks: Tick marks (both major and minor) drawn using astGrid
- Ticks1: Tick marks (both major and minor) for axis 1 drawn using astGrid
- Ticks2: Tick marks (both major and minor) for axis 2 drawn using astGrid
- Title: The Plot title drawn using astGrid

Plot3D Provide facilities for 3D graphical output Plot3D

Description: A Plot3D is a specialised form of Plot that provides facilities for producing 3D graphical output, including fully annotated 3D coordinate grids. The base Frame in a Plot3D describes a 3-dimensional "graphical" coordinate system. The axes of this coordinate system are assumed to be right-handed (that is, if X appears horizontally to the right and Y vertically upwards, then Z is out of the screen towards the viewer), and are assumed to be equally scaled (that is, the same units are used to measure positions on each of the 3 axes). The upper and lower bounds of a volume within this graphical coordinate system is specified when the Plot3D is created, and all subsequent graphics are "drawn" in this volume.

The Plot3D class does not itself include any ability to draw on a graphics device. Instead it calls upon function in an externally supplied module (the "grf3d" module) to do the required drawing. A module should be written that implements the functions of the grf3d interface using the facilities of a specific graphics system This module should then be linked into the application so that the Plot3D class can use its functions (see the description of the ast_link commands for details of how to do this). The grf3d interface defines a few simple functions for drawing primitives such as straight lines, markers and character strings. These functions all accept positions in the 3D graphics coordinate system (the base Frame of the Plot3D), and so the grf3d module must also manage the projection of these 3D coordinates onto the 2D viewing surface, including the choice of "eye"/"camera" position, direction of viewing, etc. The AST library includes a sample implementation of the grf3d interface based on the PGPLOT graphics system (see file grf3d_pgplot.c). This implementation also serves to document the grf3d interface itself and should be consulted for details before writing a new implementation.

The current Frame of a Plot3D describes a "physical" 3-dimensional coordinate system, which is the coordinate system in which plotting operations are specified when invoking the methods of the Plot3D class. The results of each plotting operation are automatically transformed into 3D graphical coordinates before being plotted using the facilities of the grf3d module linked into the application. Note, at least one of the three axes of the current Frame must be independent of the other two current Frame axes.

You may select different physical coordinate systems in which to plot (including the native graphical coordinate system itself) by selecting different Frames as the current Frame of a Plot3D, using its Current attribute.

Like any FrameSet, a Plot3D may also be used as a Frame. In this case, it behaves like its current Frame, which describes the physical coordinate system.

When used as a Mapping, a Plot3D describes the inter-relation between 3D graphical coordinates (its base Frame) and 3D physical coordinates (its current Frame).

Although the Plot3D class inherits from the Plot class, several of the facilities of the Plot class are not available in the Plot3D class, and an error will be reported if any attempt is made to use them. Specifically, the Plot3D class does not support clipping using the astClip function. Nor does it support the specification of graphics primitive functions at run-time using the astGrfSet, astGrfPop, astGrfPush and astGetGrfContext functions.

Constructor Function:
astPlot3D

Inheritance:

The Plot3D class inherits from the Plot class.

Attributes:

In addition to those attributes common to all Plots, every Plot3D also has the following attributes:

- Norm: Normal vector defining the 2D plane used for text and markers
- RootCorner: Specifies which edges of the 3D box should be annotated.

Some attributes of the Plot class refer to specific physical coordinate axes (e.g. Gap, LabelUp, DrawAxes, etc). For a basic Plot, the axis index must be 1 or 2, but for a Plot3D the axis index can be 1, 2 or 3.

Certain Plot attributes are ignored by the Plot3D class (e.g. Edge, DrawTitle, TitleGap, etc). Consult the Plot attribute documentation for details.

Functions:

The Plot3D class does not define any new functions beyond those which are applicable to all Plots. Note, however, that the following methods inherited from the Plot class cannot be used with a Plot3D and will report an error if called:

- astBoundingBox, astClip, astCurve, astGenCurve, astGetGrfContext, astGrfPop, astGrfPush, astGrfSet, astGridLine, astPolyCurve.

PointList A collection of points in a Frame PointList

Description: The PointList class implements a Region which represents a collection of points in a Frame.

Constructor Function:
astPointList

Inheritance:

The PointList class inherits from the Region class.

Attributes:

In addition to those attributes common to all Regions, every PointList also has the following attributes:

- ListSize: The number of positions stored in the PointList

Functions:

The PointList class does not define any new functions beyond those which are applicable to all Regions.

PolyMap Map coordinates using polynomial functions PolyMap

Description: A PolyMap is a form of Mapping which performs a general polynomial transformation. Each output coordinate is a polynomial function of all the input coordinates. The coefficients are specified separately for each output coordinate. The forward and inverse transformations are defined independantly by separate sets of coefficients. If no inverse transformation is supplied, an iterative method can be used to evaluate the inverse based only on the forward transformation.

Constructor Function:
astPolyMap

Inheritance:

The PolyMap class inherits from the Mapping class.

Attributes:

In addition to those attributes common to all Mappings, every PolyMap also has the following attributes:

- IterInverse: Provide an iterative inverse transformation?
- NiterInverse: Maximum number of iterations for iterative inverse
- TolInverse: Target relative error for iterative inverse

Functions:

In addition to those functions applicable to all Objects, the following functions may also be applied to all Mappings:

- astPolyTran: Fit a PolyMap inverse or forward transformation

---

Polygon A polygonal region within a 2-dimensional Frame Polygon

Description: The Polygon class implements a polygonal area, defined by a collection of vertices, within a 2-dimensional Frame. The vertices are connected together by geodesic curves within the encapsulated Frame. For instance, if the encapsulated Frame is a simple Frame then the geodesics will be straight lines, but if the Frame is a SkyFrame then the geodesics will be great circles. Note, the vertices must be supplied in an order such that the inside of the polygon is to the left of the boundary as the vertices are traversed. Supplying them in the reverse order will effectively negate the polygon.

Within a SkyFrame, neighbouring vertices are always joined using the shortest path. Thus if an edge of 180 degrees or more in length is required, it should be split into section each of which is less than 180 degrees. The closed path joining all the vertices in order will divide the celestial sphere into two disjoint regions. The inside of the polygon is the region which is circled in an anti-clockwise manner (when viewed from the inside of the celestial sphere) when moving through the list of vertices in the order in which they were supplied when the Polygon was created (i.e. the inside is to the left of the boundary when moving through the vertices in the order supplied).

Constructor Function:
astPolygon

Inheritance:

The Polygon class inherits from the Region class.

Attributes:

The Polygon class does not define any new attributes beyond those which are applicable to all Regions.

Functions:

In addition to those functions applicable to all Regions, the following functions may also be applied to all Polygons:

- astDownsize: Reduce the number of vertices in a Polygon.
- astOutline<X>: Create a Polygon outlining values in a pixel array

---

Prism An extrusion of a region into higher dimensions Prism

Description: A Prism is a Region which represents an extrusion of an existing Region into one or more orthogonal dimensions (specified by another Region). If the Region to be extruded has N axes, and the Region defining the extrusion has M axes, then the resulting Prism will have (M+N) axes. A point is inside the Prism if the first N axis values correspond to a point inside the Region being extruded, and the remaining M axis values correspond to a point inside the Region defining the extrusion.

As an example, a cylinder can be represented by extruding an existing Circle, using an Interval to define the extrusion. Ih this case, the Interval would have a single axis and would specify the upper and lower limits of the cylinder along its length.

Constructor Function:
astPrism

Inheritance:

The Prism class inherits from the Region class.

Attributes:

The Prism class does not define any new attributes beyond those which are applicable to all Regions.

Functions:

The Prism class does not define any new functions beyond those which are applicable to all Regions.

RateMap Mapping which represents differentiation RateMap

Description: A RateMap is a Mapping which represents a single element of the Jacobian matrix of another Mapping. The Mapping for which the Jacobian is required is specified when the new RateMap is created, and is referred to as the "encapsulated Mapping" below.

The number of inputs to a RateMap is the same as the number of inputs to its encapsulated Mapping. The number of outputs from a RateMap is always one. This one output equals the rate of change of a specified output of the encapsulated Mapping with respect to a specified input of the encapsulated Mapping (the input and output to use are specified when the RateMap is created).

A RateMap which has not been inverted does not define an inverse transformation. If a RateMap has been inverted then it will define an inverse transformation but not a forward transformation.

Constructor Function:
astRateMap

Inheritance:

The RateMap class inherits from the Mapping class.

Attributes:

The RateMap class does not define any new attributes beyond those which are applicable to all Mappings.

Functions:

The RateMap class does not define any new functions beyond those which are applicable to all Mappings.

Region Represents a region within a coordinate system Region

Description: This class provides the basic facilities for describing a region within a specified coordinate system. However, the Region class does not have a constructor function of its own, as it is simply a container class for a family of specialised sub-classes such as Circle, Box, etc, which implement Regions with particular shapes.

All sub-classes of Region require a Frame to be supplied when the Region is created. This Frame describes the coordinate system in which the Region is defined, and is referred to as the "encapsulated Frame" below. Constructors will also typically required one or more positions to be supplied which define the location and extent of the region. These positions must be supplied within the encapsulated Frame.

The Region class inherits from the Frame class, and so a Region can be supplied where-ever a Frame is expected. In these cases, supplying a Region is equivalent to supplying a reference to its encapsulated Frame. Thus all the methods of the Frame class can be used on the Region class. For instance, the astFormat function may be used on a Region to format an axis value.

In addition, since Frame inherits from Mapping, a Region is also a sort of Mapping. Transforming positions by supplying a Region to one of the astTran<X> functions is the way to determine if a given position is inside or outside the Region. When used as a Mapping, most classes of Frame are equivalent to a UnitMap. However, the Region class modifies this behaviour so that a Region acts like a UnitMap only for input positions which are within the area represented by the Region. Input positions which are outside the area produce bad output values (i.e. the output values are equal to AST__BAD). This behaviour is the same for both the forward and the inverse transformation. In this sense the "inverse transformation" is not a true inverse of the forward transformation, since applying the forward transformation to a point outside the Region, and then applying the inverse transformation results, in a set of AST__BAD axis values rather than the original axis values. If required, the astRemoveRegions function can be used to remove the "masking" effect of any Regions contained within a compound Mapping or FrameSet. It does this by replacing each Region with a UnitMap or equivalent Frame (depending on the context in which the Region is used).

If the coordinate system represented by the Region is changed (by changing the values of one or more of the attribute which the Region inherits from its encapsulated Frame), the area represented by the Region is mapped into the new coordinate system. For instance, let's say a Circle (a subclass of Region) is created, a SkyFrame being supplied to the constructor so that the Circle describes a circular area on the sky in FK4 equatorial coordinates. Since Region inherits from Frame, the Circle will have a System attribute and this attribute will be set to "FK4". If the System attribute of the Region is then changed from FK4 to FK5, the circular area represented by the Region will automatically be mapped from the FK4 system into the FK5 system. In general, changing the coordinate system in this way may result in the region changing shape - for instance, a circle may change into an ellipse if the transformation from the old to the new coordinate system is linear but with different scales on each axis. Thus the specific class of a Region cannot be used as a guarantee of the shape in any particular coordinate system. If the astSimplify function is used on a Region, it will endeavour to return a new Region of a sub-class which accurately describes the shape in the current coordinate system of the Region (but this may not always be possible).

It is possible to negate an existing Region so that it represents all areas of the encapsulated Frame except for the area specified when the Region was created.

Constructor Function:
None.

Inheritance:

The Region class inherits from the Frame class.

Attributes:

In addition to those attributes common to all Frames, every Region also has the following attributes:

- Adaptive: Should the area adapt to changes in the coordinate system?
- Negated: Has the original region been negated?
- Closed: Should the boundary be considered to be inside the region?
- MeshSize: Number of points used to create a mesh covering the Region
- FillFactor: Fraction of the Region which is of interest
- Bounded: Is the Region bounded?

Every Region also inherits any further attributes that belong to the encapsulated Frame, regardless of that Frame's class. (For example, the Equinox attribute, defined by the SkyFrame class, is inherited by any Region which represents a SkyFrame.)

Functions:

In addition to those functions applicable to all Frames, the following functions may also be applied to all Regions:

- astGetRegionBounds: Get the bounds of a Region
- astGetRegionFrame: Get a copy of the Frame represent by a Region
- astGetRegionMesh: Get a mesh of points covering a Region
- astGetRegionPoints: Get the positions that define a Region
- astGetUnc: Obtain uncertainty information from a Region
- astMapRegion: Transform a Region into a new coordinate system
- astNegate: Toggle the value of the Negated attribute
- astOverlap: Determines the nature of the overlap between two Regions
- astMask<X>: Mask a region of a data grid
- astSetUnc: Associate a new uncertainty with a Region
- astShowMesh: Display a mesh of points on the surface of a Region

---

SelectorMap A Mapping that locates positions within one of a set of alternate Regions SelectorMap

Description: A SelectorMap is a Mapping that identifies which Region contains a given input position.

A SelectorMap encapsulates a number of Regions that all have the same number of axes and represent the same coordinate Frame. The number of inputs (Nin attribute) of the SelectorMap equals the number of axes spanned by one of the encapsulated Region. All SelectorMaps have only a single output. SelectorMaps do not define an inverse transformation.

For each input position, the forward transformation of a SelectorMap searches through the encapsulated Regions (in the order supplied when the SelectorMap was created) until a Region is found which contains the input position. The index associated with this Region is returned as the SelectorMap output value (the index value is the position of the Region within the list of Regions supplied when the SelectorMap was created, starting at 1 for the first Region). If an input position is not contained within any Region, a value of zero is returned by the forward transformation.

If a compound Mapping contains a SelectorMap in series with its own inverse, the combination of the two adjacent SelectorMaps will be replaced by a UnitMap when the compound Mapping is simplified using astSimplify.

In practice, SelectorMaps are often used in conjunction with SwitchMaps.

Constructor Function:
astSelectorMap

Inheritance:

The SelectorMap class inherits from the Mapping class.

Attributes:

The SelectorMap class does not define any new attributes beyond those which are applicable to all Mappings.

Functions:

The SelectorMap class does not define any new functions beyond those which are applicable to all Mappings.

ShiftMap Add a constant value to each coordinate ShiftMap

Description: A ShiftMap is a linear Mapping which shifts each axis by a specified constant value.

Constructor Function:
astShiftMap

Inheritance:

The ShiftMap class inherits from the Mapping class.

Attributes:

The ShiftMap class does not define any new attributes beyond those which are applicable to all Mappings.

Functions:

The ShiftMap class does not define any new functions beyond those which are applicable to all Mappings.

SkyAxis Store celestial axis information SkyAxis

Description: The SkyAxis class is used to store information associated with a particular axis of a SkyFrame. It is used internally by the AST library and has no constructor function. You should encounter it only within textual output (e.g. from astWrite).

Constructor Function:
None.

Inheritance:

The SkyAxis class inherits from the Axis class.

SkyFrame Celestial coordinate system description SkyFrame

Description: A SkyFrame is a specialised form of Frame which describes celestial longitude/latitude coordinate systems. The particular celestial coordinate system to be represented is specified by setting the SkyFrame's System attribute (currently, the default is ICRS) qualified, as necessary, by a mean Equinox value and/or an Epoch.

For each of the supported celestial coordinate systems, a SkyFrame can apply an optional shift of origin to create a coordinate system representing offsets within the celestial coordinate system from some specified reference point. This offset coordinate system can also be rotated to define new longitude and latitude axes. See attributes SkyRef, SkyRefIs, SkyRefP and AlignOffset.

All the coordinate values used by a SkyFrame are in radians. These may be formatted in more conventional ways for display by using astFormat.

Constructor Function:
astSkyFrame

Inheritance:

The SkyFrame class inherits from the Frame class.

Attributes:

In addition to those attributes common to all Frames, every SkyFrame also has the following attributes:

- AlignOffset: Align SkyFrames using the offset coordinate system?
- AsTime(axis): Format celestial coordinates as times?
- Equinox: Epoch of the mean equinox
- IsLatAxis: Is the specified axis the latitude axis?
- IsLonAxis: Is the specified axis the longitude axis?
- LatAxis: Index of the latitude axis
- LonAxis: Index of the longitude axis
- NegLon: Display longitude values in the range [-pi,pi]?
- Projection: Sky projection description.
- SkyRef: Position defining location of the offset coordinate system
- SkyRefIs: Selects the nature of the offset coordinate system
- SkyRefP: Position defining orientation of the offset coordinate system

Functions:

In addition to those functions applicable to all Frames, the following functions may also be applied to all SkyFrames:

- astSkyOffsetMap: Obtain a Mapping from absolute to offset coordinates

SlaMap Sequence of celestial coordinate conversions SlaMap

Description: An SlaMap is a specialised form of Mapping which can be used to represent a sequence of conversions between standard celestial (longitude, latitude) coordinate systems.

When an SlaMap is first created, it simply performs a unit (null) Mapping on a pair of coordinates. Using the astSlaAdd function, a series of coordinate conversion steps may then be added, selected from those provided by the SLALIB Positional Astronomy Library (Starlink User Note SUN/67). This allows multi-step conversions between a variety of celestial coordinate systems to be assembled out of the building blocks provided by SLALIB.

For details of the individual coordinate conversions available, see the description of the astSlaAdd function.

Constructor Function:
astSlaMap (also see astSlaAdd)

Inheritance:

The SlaMap class inherits from the Mapping class.

Attributes:

The SlaMap class does not define any new attributes beyond those which are applicable to all Mappings.

Functions:

In addition to those functions applicable to all Mappings, the following function may also be applied to all SlaMaps:

- astSlaAdd: Add a celestial coordinate conversion to an SlaMap

SpecFluxFrame Compound spectrum/flux Frame SpecFluxFrame

Description: A SpecFluxFrame combines a SpecFrame and a FluxFrame into a single 2-dimensional compound Frame. Such a Frame can for instance be used to describe a Plot of a spectrum in which the first axis represents spectral position and the second axis represents flux.

Constructor Function:
astSpecFluxFrame

Inheritance:

The SpecFluxFrame class inherits from the CmpFrame class.

Attributes:

The SpecFluxFrame class does not define any new attributes beyond those which are applicable to all CmpFrames. However, the attributes of the component Frames can be accessed as if they were attributes of the SpecFluxFrame. For instance, the SpecFluxFrame will recognise the "StdOfRest" attribute and forward access requests to the component SpecFrame. An axis index can optionally be appended to the end of any attribute name, in which case the request to access the attribute will be forwarded to the primary Frame defining the specified axis.

Functions:

The SpecFluxFrame class does not define any new functions beyond those which are applicable to all CmpFrames.

SpecFrame Spectral coordinate system description SpecFrame

Description: A SpecFrame is a specialised form of one-dimensional Frame which represents various coordinate systems used to describe positions within an electro-magnetic spectrum. The particular coordinate system to be used is specified by setting the SpecFrame's System attribute (the default is wavelength) qualified, as necessary, by other attributes such as the rest frequency, the standard of rest, the epoch of observation, units, etc (see the description of the System attribute for details).

By setting a value for thr SpecOrigin attribute, a SpecFrame can be made to represent offsets from a given spectral position, rather than absolute spectral values.

Constructor Function:

astSpecFrame

Inheritance:

The SpecFrame class inherits from the Frame class.

Attributes:

In addition to those attributes common to all Frames, every SpecFrame also has the following attributes:

- AlignSpecOffset: Align SpecFrames using the offset coordinate system?
- AlignStdOfRest: Standard of rest in which to align SpecFrames
- RefDec: Declination of the source (FK5 J2000)
- RefRA: Right ascension of the source (FK5 J2000)
- RestFreq: Rest frequency
- SourceSys: Source velocity spectral system
- SourceVel: Source velocity
- SourceVRF: Source velocity rest frame
- SpecOrigin: The zero point for SpecFrame axis values
- StdOfRest: Standard of rest

Several of the Frame attributes inherited by the SpecFrame class refer to a specific axis of the Frame (for instance Unit(axis), Label(axis), etc). Since a SpecFrame is strictly one-dimensional, it allows these attributes to be specified without an axis index. So for instance, "Unit" is allowed in place of "Unit(1)".

Functions:

In addition to those functions applicable to all Frames, the following functions may also be applied to all SpecFrames:

- astSetRefPos: Set reference position in any celestial system
- astGetRefPos: Get reference position in any celestial system

---

SpecMap Sequence of spectral coordinate conversions **SpecMap**

Description: A SpecMap is a specialised form of Mapping which can be used to represent a sequence of conversions between standard spectral coordinate systems.

When an SpecMap is first created, it simply performs a unit (null) Mapping. Using the astSpecAdd function, a series of coordinate conversion steps may then be added. This allows multi-step conversions between a variety of spectral coordinate systems to be assembled out of a set of building blocks.

Conversions are available to transform between standards of rest. Such conversions need to know the source position as an RA and DEC. This information can be supplied in the form of parameters for the relevant conversions, in which case the SpecMap is 1-dimensional, simply transforming the spectral axis values. This means that the same source position will always be used by the SpecMap. However, this may not be appropriate for an accurate description of a 3-D spectral cube, where changes of spatial position can produce significant changes in the Doppler shift introduced when transforming between standards of rest. For this situation, a 3-dimensional SpecMap can be created in which axes 2 and 3 correspond to the source RA and DEC The SpecMap simply copies values for axes 2 and 3 from input to output), but modifies axis 1 values (the spectral axis) appropriately.

For details of the individual coordinate conversions available, see the description of the astSpecAdd function.

Constructor Function:
astSpecMap (also see astSpecAdd)

Inheritance:

The SpecMap class inherits from the Mapping class.

Attributes:

The SpecMap class does not define any new attributes beyond those which are applicable to all Mappings.

Functions:

In addition to those functions applicable to all Mappings, the following function may also be applied to all SpecMaps:

- astSpecAdd: Add a spectral coordinate conversion to an SpecMap

---

SphMap Map 3-d Cartesian to 2-d spherical coordinates **SphMap**

Description: A SphMap is a Mapping which transforms points from a 3-dimensional Cartesian coordinate system into a 2-dimensional spherical coordinate system (longitude and latitude on a unit sphere centred at the origin). It works by regarding the input coordinates as position vectors and finding their intersection with the sphere surface. The inverse transformation always produces points which are a unit distance from the origin (i.e. unit vectors).

Constructor Function:
astSphMap

Inheritance:

The SphMap class inherits from the Mapping class.

Attributes:

In addition to those attributes common to all Mappings, every SphMap also has the following attributes:

- UnitRadius: SphMap input vectors lie on a unit sphere?
- PolarLong: The longitude value to assign to either pole

Functions:

The SphMap class does not define any new functions beyond those which are applicable to all Mappings.

Stc Represents an instance of the IVOA STC class Stc

Description: The Stc class is an implementation of the IVOA STC class which forms part of the IVOA Space-Time Coordinate Metadata system. See:

http://hea-www.harvard.edu/∼arots/nvometa/STC.html

The Stc class does not have a constructor function of its own, as it is simply a container class for a family of specialised sub-classes including StcCatalogEntryLocation, StcResourceProfile, StcSearchLocation and StcObsDataLocation.

Constructor Function:
astStc

Inheritance:

The Stc class inherits from the Region class.

Attributes:

In addition to those attributes common to all Regions, every Stc also has the following attributes:

- RegionClass: The class name of the encapsulated Region.

Functions:

In addition to those functions applicable to all Regions, the following functions may also be applied to all Stc's:

- astGetStcRegion: Get a pointer to the encapsulated Region
- astGetStcCoord: Get information about an AstroCoords element
- astGetStcNCoord: Returns the number of AstroCoords elements in an Stc

---

StcCatalogEntryLocation Correspond to the IVOA STCCatalogEntryLocation class StcCatalogEntryLocation

Description: The StcCatalogEntryLocation class is a sub-class of Stc used to describe the coverage of the datasets contained in some VO resource.

See http://hea-www.harvard.edu/~arots/nvometa/STC.html

Constructor Function:
astStcCatalogEntryLocation

Inheritance:

The StcCatalogEntryLocation class inherits from the Stc class.

Attributes:

The StcCatalogEntryLocation class does not define any new attributes beyond those which are applicable to all Stcs.

Functions:

The StcCatalogEntryLocation class does not define any new functions beyond those which are applicable to all Stcs.

StcObsDataLocation Correspond to the IVOA ObsDataLocation class StcObsDataLocation

Description: The StcObsDataLocation class is a sub-class of Stc used to describe the coordinate space occupied by a particular observational dataset.

See http://hea-www.harvard.edu/~arots/nvometa/STC.html

An STC ObsDataLocation element specifies the extent of the observation within a specified coordinate system, and also specifies the observatory location within a second coordinate system.

The AST StcObsDataLocation class inherits from Stc, and therefore an StcObsDataLocation can be used directly as an Stc. When used in this way, the StcObsDataLocation describes the location of the observation (not the observatory).

Eventually, this class will have a method for returning an Stc describing the observatory location. However, AST currently does not include any classes of Frame for describing terrestrial or solar system positions. Therefore, the provision for returning observatory location as an Stc is not yet available. However, for terrestrial observations, the position of the observatory can still be recorded using the ObsLon and ObsLat attributes of the Frame encapsulated within the Stc representing the observation location (this assumes the observatory is located at sea level).

Constructor Function:
astStcObsDataLocation

Inheritance:

The StcObsDataLocation class inherits from the Stc class.

Attributes:

The StcObsDataLocation class does not define any new attributes beyond those which are applicable to all Stcs.

Functions:

The StcObsDataLocation class does not define any new functions beyond those which are applicable to all Stcs.

StcResourceProfile Correspond to the IVOA STCResourceProfile class StcResourceProfile

Description: The StcResourceProfile class is a sub-class of Stc used to describe the coverage of the datasets contained in some VO resource.

See http://hea-www.harvard.edu/∼arots/nvometa/STC.html

Constructor Function:
astStcResourceProfile

Inheritance:

The StcResourceProfile class inherits from the Stc class.

Attributes:

The StcResourceProfile class does not define any new attributes beyond those which are applicable to all Stcs.

Functions:

The StcResourceProfile class does not define any new functions beyond those which are applicable to all Stcs.

StcSearchLocation Correspond to the IVOA SearchLocation class StcSearchLocation

Description: The StcSearchLocation class is a sub-class of Stc used to describe the coverage of a query.

See http://hea-www.harvard.edu/∼arots/nvometa/STC.html

Constructor Function:
astStcSearchLocation

Inheritance:

The StcSearchLocation class inherits from the Stc class.

Attributes:

The StcSearchLocation class does not define any new attributes beyond those which are applicable to all Stcs.

Functions:

The StcSearchLocation class does not define any new functions beyond those which are applicable to all Stcs.

StcsChan I/O Channel using STC-S to represent Objects StcsChan

Description: A StcsChan is a specialised form of Channel which supports STC-S I/O operations. Writing an Object to an StcsChan (using astWrite) will, if the Object is suitable, generate an STC-S description of that Object, and reading from an StcsChan will create a new Object from its STC-S description.

When an STC-S description is read using astRead, the returned AST Object may be 1) a PointList describing the STC AstroCoords (i.e. a single point of interest within the coordinate frame described by the STC-S description), or 2) a Region describing the STC AstrCoordsArea (i.e. an area or volume of interest within the coordinate frame described by the STC-S description), or 3) a KeyMap containing the uninterpreted property values read form the STC-S description, or 4) a KeyMap containing any combination of the first 3 options. The attributes StcsArea, StcsCoords and StcsProps control which of the above is returned by astRead.

When an STC-S description is created from an AST Object using astWrite, the AST Object must be either a Region or a KeyMap. If it is a Region, it is assumed to define the AstroCoordsArea or (if the Region is a single point) the AstroCoords to write to the STC-S description. If the Object is a KeyMap, it may contain an entry with the key "AREA", holding a Region to be used to define the AstroCoordsArea. It may also contain an entry with the key "COORDS", holding a Region (a PointList) to be used to create the AstroCoords. It may also contain an entry with key "PROPS", holding a KeyMap that contains uninterpreted property values to be used as defaults for any STC-S properties that are not determined by the other supplied Regions. In addition, a KeyMap supplied to astWrite may itself hold the default STC-S properties (rather than defaults being held in a secondary KeyMap, stored as the "PROPS" entry in the supplied KeyMap).

The astRead and astWrite functions work together so that any Object returned by astRead can immediately be re-written using astWrite.

Normally, when you use an StcsChan, you should provide "source" and "sink" functions which connect it to an external data store by reading and writing the resulting text. These functions should perform any conversions needed between external character encodings and the internal ASCII encoding. If no such functions are supplied, a Channel will read from standard input and write to standard output.

Alternatively, an XmlChan can be told to read or write from specific text files using the SinkFile and SourceFile attributes, in which case no sink or source function need be supplied.

Support for STC-S is currently based on the IVOA document "STC-S: Space-Time Coordinate (STC) Metadata Linear String Implementation", version 1.30 (dated 5th December 2007), available at http://www.ivoa.net/Documents/latest/STC-S.html. Note, this document is a recommednation only and does not constitute an accepted IVOA standard.

The full text of version 1.30 is supported by the StcsChan class, with the following exceptions and provisos:

- When reading an STC-S phrase, case is ignored except when reading units strings.
- There is no support for multiple intervals specified within a TimeInterval, PositionInterval, SpectralInterval or RedshiftInterval.
- If the ET timescale is specified, TT is used instead.
- If the TEB timescale is specified, TDB is used instead.
- The LOCAL timescale is not supported.
- The AST TimeFrame and SkyFrame classes do not currently allow a reference position to be specified. Consequently, any <refpos> specified within the Time or Space sub-phrase of an STC-S document is ignored.
- The Convex identifier for the space sub-phrase is not supported.
- The GEO_C and GEO_D space frames are not supported.
- The UNITSPHERE and SPHER3 space flavours are not supported.
- If any Error values are supplied in a space sub-phrase, then the number of values supplied should equal the number of spatial axes, and the values are assumed to specify an error box (i.e. error circles, ellipses, etc, are not supported).
- The spectral and redshift sub-phrases do not support the following <refpos> values: LOCAL_GROUP_CENTER, UNKNOWNRefPos, EMBARYCENTER, MOON, MERCURY, VENUS, MARS, JUPITER, SATURN, URANUS, NEPTUNE, PLUTO.
- Error values are supported but error ranges are not.

- Resolution, PixSize and Size values are ignored.
- Space velocity sub-phrases are ignored.

Constructor Function:
astStcsChan

Inheritance:

The StcsChan class inherits from the Channel class.

Attributes:

In addition to those attributes common to all Channels, every StcsChan also has the following attributes:

- StcsArea: Return the CoordinateArea component after reading an STC-S?
- StcsCoords: Return the Coordinates component after reading an STC-S?
- StcsLength: Controls output buffer length
- StcsProps: Return the STC-S properties after reading an STC-S?

Functions:

The StcsChan class does not define any new functions beyond those which are applicable to all Channels.

SwitchMap A Mapping that encapsulates a set of alternate Mappings SwitchMap

Description: A SwitchMap is a Mapping which represents a set of alternate Mappings, each of which is used to transform positions within a particular region of the input or output coordinate system of the SwitchMap.

A SwitchMap can encapsulate any number of Mappings, but they must all have the same number of inputs (Nin attribute value) and the same number of outputs (Nout attribute value). The SwitchMap itself inherits these same values for its Nin and Nout attributes. Each of these Mappings represents a "route" through the switch, and are referred to as "route" Mappings below. Each route Mapping transforms positions between the input and output coordinate space of the entire SwitchMap, but only one Mapping will be used to transform any given position. The selection of the appropriate route Mapping to use with any given input position is made by another Mapping, called the "selector" Mapping. Each SwitchMap encapsulates two selector Mappings in addition to its route Mappings; one for use with the SwitchMap's forward transformation (called the "forward selector Mapping"), and one for use with the SwitchMap's inverse transformation (called the "inverse selector Mapping"). The forward selector Mapping must have the same number of inputs as the route Mappings, but should have only one output. Likewise, the inverse selector Mapping must have the same number of outputs as the route Mappings, but should have only one input.

When the SwitchMap is used to transform a position in the forward direction (from input to output), each supplied input position is first transformed by the forward transformation of the forward selector Mapping. This produces a single output value for each input position referred to as the selector value. The nearest integer to the selector value is found, and is used to index the array of route Mappings (the first supplied route Mapping has index 1, the second route Mapping has index 2, etc). If the nearest integer to the selector value is less than 1 or greater than the number of route Mappings, then the SwitchMap output position is set to a value of AST__BAD on every axis. Otherwise, the forward transformation of the selected route Mapping is used to transform the supplied input position to produce the SwitchMap output position.

When the SwitchMap is used to transform a position in the inverse direction (from "output" to "input"), each supplied "output" position is first transformed by the inverse transformation of the inverse selector Mapping. This produces a selector value for each "output" position. Again, the nearest integer to the selector value is found, and is used to index the array of route Mappings. If this selector index value is within the bounds of the array of route Mappings, then the inverse transformation of the selected route Mapping is used to transform the supplied "output" position to produce the SwitchMap "input" position. If the selector index value is outside the bounds of the array of route Mappings, then the SwitchMap "input" position is set to a value of AST__BAD on every axis.

In practice, appropriate selector Mappings should be chosen to associate a different route Mapping with each region of coordinate space. Note that the SelectorMap class of Mapping is particularly appropriate for this purpose.

If a compound Mapping contains a SwitchMap in series with its own inverse, the combination of the two adjacent SwitchMaps will be replaced by a UnitMap when the compound Mapping is simplified using astSimplify.

Constructor Function:
astSwitchMap

Inheritance:

The SwitchMap class inherits from the Mapping class.

Attributes:

The SwitchMap class does not define any new attributes beyond those which are applicable to all Mappings.

Functions:

The SwitchMap class does not define any new functions beyond those which are applicable to all Mappings.

Table A 2-dimensional table of values Table

Description: The Table class is a type of KeyMap that represents a two-dimensional table of values. The astMapGet... and astMapPut... methods provided by the KeyMap class should be used for storing and retrieving values from individual cells within a Table. Each entry in the KeyMap represents a single cell of the table and has an associated key of the form "<COL>(i)" where "<COL>" is the upper-case name of a table column and "i" is the row index (the first row is row 1). Keys of this form should always be used when using KeyMap methods to access entries within a Table.

Columns must be declared using the astAddColumn method before values can be stored within them. This also fixes the type and shape of the values that may be stored in any cell of the column. Cells may contain scalar or vector values of any data type supported by the KeyMap class. Multi-dimensional arrays may also be stored, but these must be vectorised when storing and retrieving them within a table cell. All cells within a single column must have the same type and shape, as specified when the column is added to the Table.

Tables may have parameters that describe global properties of the entire table. These are stored as entries in the parent KeyMap and can be access using the get and set method of the KeyMap class. However, parameters must be declared using the astAddParameter method before being accessed.

Note - since accessing entries within a KeyMap is a relatively slow process, it is not recommended to use the Table class to store very large tables.

Constructor Function:
astTable

Inheritance:

The Table class inherits from the KeyMap class.

Attributes:

In addition to those attributes common to all KeyMaps, every Table also has the following attributes:

- ColumnLenC(column): The largest string length of any value in a column
- ColumnLength(column): The number of elements in each value in a column
- ColumnNdim(column): The number of axes spanned by each value in a column
- ColumnType(column): The data type of each value in a column
- ColumnUnit(column): The unit string describing each value in a column
- Ncolumn: The number of columns currently in the Table
- Nrow: The number of rows currently in the Table
- Nparameter: The number of global parameters currently in the Table

Functions:

In addition to those functions applicable to all KeyMaps, the following functions may also be applied to all Tables:

- astAddColumn: Add a new column definition to a Table
- astAddParameter: Add a new global parameter definition to a Table
- astColumnName: Return the name of the column with a given index
- astColumnShape: Return the shape of the values in a named column
- astHasColumn: Checks if a column exists in a Table
- astHasParameter: Checks if a global parameter exists in a Table
- astParameterName: Return the name of the parameter with a given index
- astPurgeRows: Remove all empty rows from a Table
- astRemoveColumn: Remove a column from a Table
- astRemoveParameter: Remove a global parameter from a Table
- astRemoveRow: Remove a row from a Table

TimeFrame Time coordinate system description TimeFrame

Description: A TimeFrame is a specialised form of one-dimensional Frame which represents various coordinate systems used to describe positions in time.

A TimeFrame represents a moment in time as either an Modified Julian Date (MJD), a Julian Date (JD), a Besselian epoch or a Julian epoch, as determined by the System attribute. Optionally, a zero point can be specified (using attribute TimeOrigin) which results in the TimeFrame representing time offsets from the specified zero point.

Even though JD and MJD are defined as being in units of days, the TimeFrame class allows other units to be used (via the Unit attribute) on the basis of simple scalings (60 seconds = 1 minute, 60 minutes = 1 hour, 24 hours = 1 day, 365.25 days = 1 year). Likewise, Julian epochs can be described in units other than the usual years. Besselian epoch are always represented in units of (tropical) years.

The TimeScale attribute allows the time scale to be specified (that is, the physical process used to define the rate of flow of time). MJD, JD and Julian epoch can be used to represent a time in any supported time scale. However, Besselian epoch may only be used with the "TT" (Terrestrial Time) time scale. The list of supported time scales includes universal time and siderial time. Strictly, these represent angles rather than time scales, but are included in the list since they are in common use and are often thought of as time scales.

When a time value is formatted it can be formated either as a simple floating point value, or as a Gregorian date (see the Format attribute).

Constructor Function:
astTimeFrame

Inheritance:

The TimeFrame class inherits from the Frame class.

Attributes:

In addition to those attributes common to all Frames, every TimeFrame also has the following attributes:

- AlignTimeScale: Time scale in which to align TimeFrames
- LTOffset: The offset of Local Time from UTC, in hours.
- TimeOrigin: The zero point for TimeFrame axis values
- TimeScale: The timescale used by the TimeFrame

Several of the Frame attributes inherited by the TimeFrame class refer to a specific axis of the Frame (for instance Unit(axis), Label(axis), etc). Since a TimeFrame is strictly one-dimensional, it allows these attributes to be specified without an axis index. So for instance, "Unit" is allowed in place of "Unit(1)".

Functions:

In addition to those functions applicable to all Frames, the following functions may also be applied to all TimeFrames:

- astCurrentTime: Return the current system time

TimeMap Sequence of time coordinate conversions TimeMap

Description: A TimeMap is a specialised form of 1-dimensional Mapping which can be used to represent a sequence of conversions between standard time coordinate systems.

When a TimeMap is first created, it simply performs a unit (null) Mapping. Using the astTimeAdd function, a series of coordinate conversion steps may then be added. This allows multi-step conversions between a variety of time coordinate systems to be assembled out of a set of building blocks.

For details of the individual coordinate conversions available, see the description of the astTimeAdd function.

Constructor Function:
astTimeMap (also see astTimeAdd)

Inheritance:

The TimeMap class inherits from the Mapping class.

Attributes:

The TimeMap class does not define any new attributes beyond those which are applicable to all Mappings.

Functions:

In addition to those functions applicable to all Mappings, the following function may also be applied to all TimeMaps:

- astTimeAdd: Add a time coordinate conversion to an TimeMap

TranMap Mapping with specified forward and inverse transformations TranMap

Description: A TranMap is a Mapping which combines the forward transformation of a supplied Mapping with the inverse transformation of another supplied Mapping, ignoring the un-used transformation in each Mapping (indeed the un-used transformation need not exist).

When the forward transformation of the TranMap is referred to, the transformation actually used is the forward transformation of the first Mapping supplied when the TranMap was constructed. Likewise, when the inverse transformation of the TranMap is referred to, the transformation actually used is the inverse transformation of the second Mapping supplied when the TranMap was constructed.

Constructor Function:
astTranMap

Inheritance:

The TranMap class inherits from the Mapping class.

Attributes:

The TranMap class does not define any new attributes beyond those which are applicable to all Mappings.

Functions:

The TranMap class does not define any new functions beyond those which are applicable to all Mappings.

UnitMap Unit (null) Mapping UnitMap

Description: A UnitMap is a unit (null) Mapping that has no effect on the coordinates supplied to it. They are simply copied. This can be useful if a Mapping is required (e.g. to pass to another function) but you do not want it to have any effect. The Nin and Nout attributes of a UnitMap are always equal and are specified when it is created.

Constructor Function:
astUnitMap

Inheritance:

The UnitMap class inherits from the Mapping class.

Attributes:

The UnitMap class does not define any new attributes beyond those which are applicable to all Mappings.

Functions:

The UnitMap class does not define any new functions beyond those which are applicable to all Mappings.

WcsMap Implement a FITS-WCS sky projection WcsMap

Description: This class is used to represent sky coordinate projections as described in the FITS world coordinate system (FITS-WCS) paper II "Representations of Celestial Coordinates in FITS" by M. Calabretta and E.W. Griesen. This paper defines a set of functions, or sky projections, which transform longitude-latitude pairs representing spherical celestial coordinates into corresponding pairs of Cartesian coordinates (and vice versa).

A WcsMap is a specialised form of Mapping which implements these sky projections and applies them to a specified pair of coordinates. All the projections in the FITS-WCS paper are supported, plus the now deprecated "TAN with polynomial correction terms" projection which is refered to here by the code "TPN". Using the FITS-WCS terminology, the transformation is between "native spherical" and "projection plane" coordinates (also called "intermediate world coordinates". These coordinates may, optionally, be embedded in a space with more than two dimensions, the remaining coordinates being copied unchanged. Note, however, that for consistency with other AST facilities, a WcsMap handles coordinates that represent angles in radians (rather than the degrees used by FITS-WCS).

The type of FITS-WCS projection to be used and the coordinates (axes) to which it applies are specified when a WcsMap is first created. The projection type may subsequently be determined using the WcsType attribute and the coordinates on which it acts may be determined using the WcsAxis(lonlat) attribute.

Each WcsMap also allows up to 100 "projection parameters" to be associated with each axis. These specify the precise form of the projection, and are accessed using PVi_m attribute, where "i" is the integer axis index (starting at 1), and m is an integer "parameter index" in the range 0 to 99. The number of projection parameters required by each projection, and their meanings, are dependent upon the projection type (most projections either do not use any projection parameters, or use parameters 1 and 2 associated with the latitude axis). Before creating a WcsMap you should consult the FITS-WCS paper for details of which projection parameters are required, and which have defaults. When creating the WcsMap, you must explicitly set values for all those required projection parameters which do not have defaults defined in this paper.

Constructor Function:
astWcsMap

Inheritance:

The WcsMap class inherits from the Mapping class.

Attributes:

In addition to those attributes common to all Mappings, every WcsMap also has the following attributes:

- NatLat: Native latitude of the reference point of a FITS-WCS projection
- NatLon: Native longitude of the reference point of a FITS-WCS projection
- PVi_m: FITS-WCS projection parameters
- PVMax: Maximum number of FITS-WCS projection parameters
- WcsAxis(lonlat): FITS-WCS projection axes
- WcsType: FITS-WCS projection type

Functions:

The WcsMap class does not define any new functions beyond those which are applicable to all Mappings.

WinMap Map one window on to another by scaling and shifting each axis WinMap

Description: A Winmap is a linear Mapping which transforms a rectangular window in one coordinate system into a similar window in another coordinate system by scaling and shifting each axis (the window edges being parallel to the coordinate axes).

A WinMap is specified by giving the coordinates of two opposite corners (A and B) of the window in both the input and output coordinate systems.

Constructor Function:
astWinMap

Inheritance:

The WinMap class inherits from the Mapping class.

Attributes:

The WinMap class does not define any new attributes beyond those which are applicable to all Mappings.

Functions:

The WinMap class does not define any new functions beyond those which are applicable to all Mappings.

XmlChan I/O Channel using XML to represent Objects XmlChan

Description: A XmlChan is a specialised form of Channel which supports XML I/O operations. Writing an Object to an XmlChan (using astWrite) will, if the Object is suitable, generate an XML description of that Object, and reading from an XmlChan will create a new Object from its XML description.

Normally, when you use an XmlChan, you should provide "source" and "sink" functions which connect it to an external data store by reading and writing the resulting XML text. These functions should perform any conversions needed between external character encodings and the internal ASCII encoding. If no such functions are supplied, a Channel will read from standard input and write to standard output.

Alternatively, an XmlChan can be told to read or write from specific text files using the SinkFile and SourceFile attributes, in which case no sink or source function need be supplied.

Constructor Function:
astXmlChan

Inheritance:

The XmlChan class inherits from the Channel class.

Attributes:

In addition to those attributes common to all Channels, every XmlChan also has the following attributes:

- XmlFormat: System for formatting Objects as XML
- XmlLength: Controls output buffer length
- XmlPrefix: The namespace prefix to use when writing

Functions:

The XmlChan class does not define any new functions beyond those which are applicable to all Mappings.

ZoomMap Zoom coordinates about the origin **ZoomMap**

Description: The ZoomMap class implements a Mapping which performs a "zoom" transformation by multiplying all coordinate values by the same scale factor (the inverse transformation is performed by dividing by this scale factor). The number of coordinate values representing each point is unchanged.

Constructor Function:
astZoomMap

Inheritance:

The ZoomMap class inherits from the Mapping class.

Attributes:

In addition to those attributes common to all Mappings, every ZoomMap also has the following attributes:

- Zoom: ZoomMap scale factor

Functions:

The ZoomMap class does not define any new functions beyond those which are applicable to all Mappings.

E UNIX Command Descriptions

The commands described here are provided for use from the UNIX shell to assist with developing software which uses AST. To use these commands, you should ensure that the directory "/star/bin"[38] is on your PATH.

---

ast_link Link a program with the AST library ast_link

Description: This command should be used when building programs which use the AST library, in order to generate the correct arguments to allow the compiler to link your program. The arguments generated are written to standard output but may be substituted into the compiler command line in the standard UNIX way using backward quotes (see below).

By default, it is assumed that you are building a stand-alone program which does not produce graphical output. However, switches are provided for linking other types of program.

Invocation:

```
cc program.c -L/star/lib ‘ast_link [switches]‘ -o program
```

Examples:

```
cc display.c -L/star/lib ‘ast_link -pgplot‘ -o display
```

Compiles and links a C program called "display" which uses the standard version of PGPLOT for graphical output.

```
cc plotit.c -L. -L/star/lib ‘ast_link -grf‘ -lgrf -o plotit
```

Compiles and links a C program "plotit". The "-grf" switch indicates that graphical output will be delivered through a graphical interface which you have implemented yourself, which corresponds to the interface required by the current version of AST. Here, this interface is supplied by means of the "-lgrf" library reference.

```
cc plotit.c -L. -L/star/lib ‘ast_link -grf_v2.0‘ -lgrf -o plotit
```

Compiles and links a C program "plotit". The "-grf_v2.0" switch indicates that graphical output will be delivered through a graphical interface which you have implemented yourself, which corresponds to the interface required by version 2.0 of AST. Here, this interface is supplied by means of the "-lgrf" library reference.

Switches:

The following switches may optionally be given to this command to modify its behaviour:

- "-csla": Ignored. Provided for backward compatibility only.
- "-fsla": Ignored. Provided for backward compatibility only.
- "-ems": Requests that the program be linked so that error messages produced by the AST library are delivered via the Starlink EMS (Error Message Service) library (Starlink System Note SSN/4). By default, error messages are simply written to standard error.
- "-drama": Requests that the program be linked so that error messages produced by the AST library are delivered via the DRAMA Ers (Error Reporting Service) library. By default, error messages are simply written to standard error.
- "-grf": Requests that no arguments be generated to specify which 2D graphics system is used to display output from the AST library. You should use this option only if you have implemented an interface to a new graphics system yourself and wish to provide your own arguments for linking with it. This switch differs from the other "grf" switches in that it assumes that your graphics module implements the complete interface required by the current version of AST. If future versions of AST introduce new functions to the graphics interface,

---

[38] Or the equivalent directory if AST is installed in a non-standard location.

this switch will cause "unresolved symbol" errors to occur during linking, warning you that you need to implement new functions in your graphics module. To avoid such errors, you can use one of the other, version-specific, switches in place of the "-grf" switch, but these will cause run-time errors to be reported if any AST function is invoked which requires facilities not in the implemented interface.

- "-grf_v2.0": This switch is equivalent to the "-mygrf" switch. It indicates that you want to link with your own graphics module which implements the 2D graphics interface required by V2.0 of AST.
- "-grf_v3.2": Indicates that you want to link with your own graphics module which implements the 2D graphics interface required by V3.2 of AST.
- "-grf_v5.6": Indicates that you want to link with your own graphics module which implements the 2D graphics interface required by V5.6 of AST.
- "-myerr": Requests that no arguments be generated to specify how error messages produced by the AST library should be delivered. You should use this option only if you have implemented an interface to a new error delivery system yourself and wish to provide your own arguments for linking with it.
- "-mygrf": This switch has been superceeded by the "-grf" switch, but is retained in order to allow applications to be linked with a graphics module which implements the 2D interface used by AST V2.0. It is equivalent to the "-grf_v2.0" switch.
- "-pgp": Requests that the program be linked so that 2D graphical output from the AST library is displayed via the Starlink version of the PGPLOT graphics package (which uses GKS for its output). By default, no 2D graphics package is linked and this will result in an error at run time if AST routines are invoked that attempt to generate graphical output.
- "-pgplot": Requests that the program be linked so that 2D graphical output from the AST library is displayed via the standard (or "native") version of the PGPLOT graphics package. By default, no 2D graphics package is linked and this will result in an error at run time if AST routines are invoked that attempt to generate graphical output.
- "-grf3d": Requests that no arguments be generated to specify which 3D graphics system is used to display output from the AST library. You should use this option only if you have implemented an interface to a new 3D graphics system yourself and wish to provide your own arguments for linking with it.
- "-pgp3d": Requests that the program be linked so that 3D graphical output from the AST library is displayed via the Starlink version of the PGPLOT graphics package (which uses GKS for its output). By default, no 3D graphics package is linked and this will result in an error at run time if AST routines are invoked that attempt to generate graphical output.
- "-pgplot3d": Requests that the program be linked so that 3D graphical output from the AST library is displayed via the standard (or "native") version of the PGPLOT graphics package. By default, no 3D graphics package is linked and this will result in an error at run time if AST routines are invoked that attempt to generate graphical output.

SOFA & PAL:

The AST distribution includes bundled copies of the IAU SOFA and Starlink PAL libraries. These will be used for fundamental positional astronomy calculations unless the "–with-external_pal" option was used when AST was configured. If "–with-external_pal" is used, this script will include "-lpal" in the returned list of linking options, and the user should then ensure that extrnal copies of the PAL and SOFA libraries are available (SOFA functions are used within PAL).

---

ast_link_adam Link an ADAM program with the AST library ast_link_adam

Description: This command should only be used when building Starlink ADAM programs which use the AST library, in order to generate the correct arguments to allow the ADAM “alink” command to link the program. The arguments generated are written to standard output but may be substituted into the “alink” command line in the standard UNIX way using backward quotes (see below).

By default, it is assumed that you are building an ADAM program which does not produce graphical output. However, switches are provided for linking other types of program. This command should not be used when building stand-alone (non-ADAM) programs. Use the “ast_link” command instead.

Invocation: `alink program.o -L/star/lib ‘ast_link_adam [switches]‘`

Examples:

`alink display.o -L/star/lib ‘ast_link_adam -pgplot‘`

Links an ADAM program “display” which uses the standard version of PGPLOT for graphical output.

`alink plotit.o -L. -L/star/lib ‘ast_link_adam -grf‘ -lgrf`

Links an ADAM program “plotit”, written in C. The “-grf” switch indicates that graphical output will be delivered through a graphical interface which you have implemented yourself, which corresponds to the interface required by the current version of AST. Here, this interface is supplied by means of the “-lgrf” library reference.

`alink plotit.o -L. -L/star/lib ‘ast_link_adam -grf_v2.0‘ -lgrf`

Links an ADAM program “plotit”, written in C. The “-grf_v2.0” switch indicates that graphical output will be delivered through a graphical interface which you have implemented yourself, which corresponds to the interface required by version 2.0 of AST. Here, this interface is supplied by means of the “-lgrf” library reference.

Switches:

The following switches may optionally be given to this command to modify its behaviour:

- “-csla”: Ignored. Provided for backward compatibility only.
- “-fsla”: Ignored. Provided for backward compatibility only.
- “-grf”: Requests that no arguments be generated to specify which 2D graphics system is used to display output from the AST library. You should use this option only if you have implemented an interface to a new graphics system yourself and wish to provide your own arguments for linking with it. This switch differs from the other “grf” switches in that it assumes that your graphics module implements the complete interface required by the current version of AST. If future versions of AST introduce new functions to the graphics interface, this switch will cause “unresolved symbol” errors to occur during linking, warning you that you need to implement new functions in your graphics module. To avoid such errors, you can use one of the other, version-specific, switches in place of the “-grf” switch, but these will cause run-time errors to be reported if any AST function is invoked which requires facilities not in the implemented interface.
- “-grf_v2.0”: This switch is equivalent to the “-mygrf” switch. It indicates that you want to link with your own graphics module which implements the 2D graphics interface required by V2.0 of AST.
- “-grf_v3.2”: Indicates that you want to link with your own graphics module which implements the 2D graphics interface required by V3.2 of AST.
- “-grf_v5.6”: Indicates that you want to link with your own graphics module which implements the 2D graphics interface required by V5.6 of AST.
- “-myerr”: Requests that no arguments be generated to specify how error messages produced by the AST library should be delivered. You should use this option only if you have implemented an interface to a new error delivery system yourself and wish to provide your own

arguments for linking with it. By default, error messages are delivered in the standard ADAM way via the EMS Error Message Service (Starlink System Note SSN/4).

- "-mygrf": This switch has been superceeded by the "-grf" switch, but is retained in order to allow applications to be linked with a graphics module which implements the interface used by AST V2.0. It is equivalent to the "-grf_v2.0" switch.
- "-pgp": Requests that the program be linked so that 2D graphical output from the AST library is displayed via the Starlink version of the PGPLOT graphics package (which uses GKS for its output). By default, no graphics package is linked and this will result in an error at run time if AST routines are invoked that attempt to generate graphical output.
- "-pgplot": Requests that the program be linked so that 2D graphical output from the AST library is displayed via the standard (or "native") version of the PGPLOT graphics package. By default, no graphics package is linked and this will result in an error at run time if AST routines are invoked that attempt to generate graphical output.
- "-grf3d": Requests that no arguments be generated to specify which 3D graphics system is used to display output from the AST library. You should use this option only if you have implemented an interface to a new 3D graphics system yourself and wish to provide your own arguments for linking with it.
- "-pgp3d": Requests that the program be linked so that 3D graphical output from the AST library is displayed via the Starlink version of the PGPLOT graphics package (which uses GKS for its output). By default, no 3D graphics package is linked and this will result in an error at run time if AST routines are invoked that attempt to generate graphical output.
- "-pgplot3d": Requests that the program be linked so that 3D graphical output from the AST library is displayed via the standard (or "native") version of the PGPLOT graphics package. By default, no 3D graphics package is linked and this will result in an error at run time if AST routines are invoked that attempt to generate graphical output.

SLALIB:

The AST distribution includes a cut down subset of the C version of the SLALIB library written by Pat Wallace. This subset contains only the functions needed by the AST library. It is built as part of the process of building AST and is distributed under GPL (and is thus compatible with the AST license). Previous version of this script allowed AST applications to be linked against external SLALIB libraries (either Fortran or C) rather than the internal version. The current version of this script does not provide this option, and always uses the internal SLALIB library. However, for backward compatibility, this script still allows the `"-fsla"` and `"-csla"` flags (previously used for selecting which version of SLALIB to use) to be specified, but they will be ignored.

F AST Memory Management and Utility Functions

AST provides a memory management layer that can be used in place of system functions such as `malloc`, `free`, `realloc`, *etc.* The AST replacements for these functions (`astMalloc`, `astFree` and `astRealloc`) add extra information to each allocated memory block that allows AST to check the validity of supplied pointers. For example, this extra information allows `astFree` to detect if the supplied pointer has already been freed, and if so to issue an appropriate error message. The existence of this extra information is invisible to outside callers, and stored in a header block located just before the returned memory block.

In addition to the standard functions, AST provides other memory management functions, such as:

`astStore` - stores data in dynamically allocated memory, allocating the memory (or adjusting the size of previously allocated memory) to match the amount of data to be stored.

`astGrow` - allocates and expands memory to hold an adjustable-sized array.

`astAppendString` - allocates and expands memory to hold a concatenated string.

Theses are just a few of the available utilities functions in the AST memory management layer. Prototypes for all AST memory management functions are included in the header file "`ast.h`".

An important restriction on these functions is that pointers created by other memory management functions, such as the system version of `malloc` *etc.*, should never supplied to an AST memory management function. Only pointers created by AST should be used by these functions.

In addition to memory management functions, AST provides various other utility functions, such as a basic regular expression facility, and other string manipulation functions. These are also documented in this appendix.

The AST memory management layer is implemented on top of the usual `malloc`, tt free and `realloc` functions. By default these will be the standard functions provided by ¡stdlib.h¿. However, the facilities of the STARMEM package (included in the Starlink Software Collection) can be used to specify alternative functions to use. This requires that AST be configured using the "–with-starmem" option when it is built.

The STARMEM package provides a wrapper for the standard malloc implementation that enables the user to switch malloc schemes at runtime by setting the STARMEM_MALLOC environment variable. Currently allowed values for this variable are:

SYSTEM - standard system malloc/free - the default

DL - Doug Lea's malloc/free

GC - Hans-Boehm Garbage Collection

---

astAppendString Append a string to another string which grows dynamically astAppendString

Description: This function appends one string to another dynamically allocated string, extending the dynamic string as necessary to accommodate the new characters (plus the final null).

Synopsis: `char *astAppendString( char *str1, int *nc, const char *str2 )`

Parameters:

str1

Pointer to the null-terminated dynamic string, whose memory has been allocated using an AST memory allocation function. If no space has yet been allocated for this string, a NULL pointer may be given and fresh space will be allocated by this function.

nc

Pointer to an integer containing the number of characters in the dynamic string (excluding the final null). This is used to save repeated searching of this string to determine its length and it defines the point where the new string will be appended. Its value is updated by this function to include the extra characters appended.

If "str1" is NULL, the initial value supplied for "*nc" will be ignored and zero will be used.

str2

Pointer to a constant null-terminated string, a copy of which is to be appended to "str1".

Returned Value:

astAppendString()

A possibly new pointer to the dynamic string with the new string appended (its location in memory may have to change if it has to be extended, in which case the original memory is automatically freed by this function). When the string is no longer required, its memory should be freed using astFree.

Notes:

- If this function is invoked with the global error status set or if it should fail for any reason, then the returned pointer will be equal to "str1" and the dynamic string contents will be unchanged.

---

astCalloc Allocate and initialise memory astCalloc

Description: This function allocates memory in a similar manner to the standard C "calloc" function, but with improved security (against memory leaks, etc.) and with error reporting. It also fills the allocated memory with zeros.

Like astMalloc, it allows zero-sized memory allocation (without error), resulting in a NULL returned pointer value.

Synopsis: `void *astCalloc( size_t nmemb, size_t size )`

Parameters:

nmemb

The number of array elements for which memory is to be allocated.

size

The size of each array element, in bytes.

Returned Value:

astCalloc()

If successful, the function returns a pointer to the start of the allocated memory region. If the size allocated is zero, this will be a NULL pointer.

Notes:

- A pointer value of NULL is returned if this function is invoked with the global error status set or if it fails for any reason.

---

astChr2Double read a double value from a string astChr2Double

Description: This function reads a double from the supplied null-terminated string, ignoring leading and trailing white space. AST__BAD is ereturned without error if the string is not a numerical value.

Synopsis: `double astChr2Double( const char *str )`

Parameters:

str

Pointer to the string.

Returned Value:

astChr2Double()

The double value, or AST__BAD.

Notes:

- A value of AST__BAD is returned if this function is invoked with the global error status set or if it should fail for any reason.

---

astChrCase Convert a string to upper or lower case astChrCase

Description: This function converts a supplied string to upper or lower case, storing the result in a supplied buffer. The astStringCase function is similar, but stores the result in a dynamically allocated buffer.

Synopsis: `void astChrCase( const char *in, char *out, int upper, int blen, int *status )`

Parameters:

in

Pointer to the null terminated string to be converted. If this is NULL, the supplied contents of the "out" string are used as the input string.

out

Pointer to the buffer to receive the converted string. The length of this buffer is given by "blen". If NULL is supplied for "in", then the supplied contents of "out" are converted and written back into "out" over-writing the supplied contents.

upper

If non-zero, the string is converted to upper case. Otherwise it is converted to lower case.

blen

The length of the output buffer. Ignored if "in" is NULL. No more than "blen - 1" characters will be copied from "in" to "out", and a terminating null character will then be added.

astChrLen Determine the used length of a string astChrLen

Description: This function returns the used length of a string. This excludes any trailing white space or non-printable characters (such as the trailing null character).

Synopsis: `size_t astChrLen( const char *string )`

Parameters:

string
Pointer to the string.

Returned Value:

astChrLen()
The number of characters in the supplied string, not including the trailing newline, and any trailing white-spaces or non-printable characters.

astChrMatch Case insensitive string comparison astChrMatch

Description: This function compares two null terminated strings for equality, discounting differences in case and any trailing white space in either string.

Synopsis: `int astChrMatch( const char *str1, const char *str2 )`

Parameters:

str1
Pointer to the first string.

str2
Pointer to the second string.

Returned Value:

astChrMatch()
Non-zero if the two strings match, otherwise zero.

Notes:

- A value of zero is returned if this function is invoked with the global error status set or if it should fail for any reason.

astChrMatchN Case insensitive string comparison of at most N characters astChrMatchN

Description: This function compares two null terminated strings for equality, discounting differences in case and any trailing white space in either string. No more than `"n"` characters are compared.

Synopsis: `int astChrMatchN( const char *str1, const char *str2, size_t n )`

Parameters:

str1
Pointer to the first string.

str2
Pointer to the second string.

n

Maximum number of characters to compare.

Returned Value:

astChrMatchN()

Non-zero if the two strings match, otherwise zero.

Notes:

- A value of zero is returned if this function is invoked with the global error status set or if it should fail for any reason.

---

astChrSplit Extract words from a supplied string astChrSplit

Description: This function extracts all space-separated words form the supplied string and returns them in an array of dynamically allocated strings.

Synopsis: `char **astChrSplit_( const char *str, int *n )`

Parameters:

str

Pointer to the string to be split.

n

Address of an int in which to return the number of words returned.

Returned Value:

astChrSplit()

A pointer to a dynamically allocated array containing "*n" elements. Each element is a pointer to a dynamically allocated character string containing a word extracted from the supplied string. Each of these words will have no leading or trailing white space.

Notes:

- A NULL pointer is returned if this function is invoked with the global error status set or if it should fail for any reason, or if the supplied string contains no words.

---

astChrSplitC Split a string using a specified character delimiter astChrSplitC

Description: This function extracts all sub-strings separated by a given character from the supplied string and returns them in an array of dynamically allocated strings. The delimiter character itself is not included in the returned strings.

Delimiter characters that are preceded by "\" are not used as delimiters but are included in the returned word instead (without the "\").

Synopsis: `char **astChrSplitC( const char *str, char c, int *n )`

Parameters:

str

Pointer to the string to be split.

c

The delimiter character.

n

Address of an int in which to return the number of words returned.

Returned Value:

astChrSplitC()

A pointer to a dynamically allocated array containing "*n" elements. Each element is a pointer to a dynamically allocated character string containing a word extracted from the supplied string.

Notes:

- A NULL pointer is returned if this function is invoked with the global error status set or if it should fail for any reason, or if the supplied string contains no words.

---

astChrSplitRE Extract sub-strings matching a specified regular expression astChrSplitRE

Description: This function compares the supplied string with the supplied regular expression. If they match, each section of the test string that corresponds to a parenthesised sub-string in the regular expression is copied and stored in the returned array.

Synopsis: `char **astChrSplitRE( const char *str, const char *regexp, int *n, const char **matchend )`

Parameters:

str

Pointer to the string to be split.

regexp

The regular expression. See "Template Syntax:" in the astChrSub prologue. Note, this function differs from astChrSub in that any equals signs (=) in the regular expression are treated literally.

n

Address of an int in which to return the number of sub-strings returned.

matchend

A pointer to a location at which to return a pointer to the character that follows the last character within the supplied test string that matched any parenthesises sub-section of "regexp". A NULL pointer is returned if no matches were found. A NULL pointer may be supplied if the location of the last matching character is not needed.

Returned Value:

astChrSplitRE()

A pointer to a dynamically allocated array containing "*n" elements. Each element is a pointer to a dynamically allocated character string containing a sub-string extracted from the supplied string. The array itself, and the strings within it, should all be freed using astFree when no longer needed.

Notes:

- If a parenthesised sub-string in the regular expression is matched by more than one sub-string within the test string, then only the first is returned. To return multiple matches, the regular expression should include multiple copies of the parenthesised sub-string (for instance, separated by ".+?" if the intervening string is immaterial).
- A NULL pointer is returned if this function is invoked with the global error status set or if it should fail for any reason, or if the supplied string contains no words.

astChrSub Performs substitutions on a supplied string astChrSub

Description: This function checks a supplied test string to see if it matches a supplied template. If it does, specified sub-sections of the test string may optionally be replaced by supplied substitution strings. The resulting string is returned.

Synopsis: `char *astChrSub( const char *test, const char *pattern, const char *subs[], int nsub )`

Parameters:

test
The string to be tested.

pattern
The template string. See "Template Syntax:" below.

subs
An array of strings that are to replace the sections of the test string that match each parenthesised sub-string in "pattern". The first element of "subs" replaces the part of the test string that matches the first parenthesised sub-string in the template, etc.

If "nsub" is zero, then the "subs" pointer is ignored. In this case, substitution strings may be specified by appended them to the end of the "pattern" string, separated by "=" characters. Note, if you need to include a literal "=" character in the pattern, precede it by an escape "\" character.

nsub
The number of substitution strings supplied in array "subs".

Returned Value:

astChrSub()
A pointer to a dynamically allocated string holding the result of the substitutions, or NULL if the test string does not match the template string. This string should be freed using astFree when no longer needed. If no substituions are specified then a copy of the test string is returned if it matches the template.

Notes:

- A NULL pointer is returned if this function is invoked with the global error status set or if it should fail for any reason, or if the supplied test string does not match the template.

Template Syntax:

The template syntax is a minimal form of regular expression, The quantifiers allowed are "*", "?", "+", "{n}", "*?" and "+?" (the last two are non-greedy - they match the minimum length possible that still gives an overall match to the template). The only constraints allowed are "∧" and "$". The following atoms are allowed:

chars : Matches any of the specified characters.

∧chars : Matches anything but the specified characters.

- .: Matches any single character.
- x: Matches the character x so long as x has no other significance.
- \x: Always matches the character x (except for [dDsSwW]).
- \d: Matches a single digit.
- \D: Matches anything but a single digit.

- \w: Matches any alphanumeric character, and `"_"`.
- \W: Matches anything but alphanumeric characters, and `"_"`.
- \s: Matches white space.
- \S: Matches anything but white space.

Note, minus signs (`"-"`) within brackets have no special significance, so ranges of characters must be specified explicitly.

Multiple template strings can be concatenated, using the "|" character to separate them. The test string is compared against each one in turn until a match is found.

Parentheses are used within each template to identify sub-strings that are to be replaced by the strings supplied in `"sub"`.

If `"nsub"` is supplied as zero, then substitution strings may be specified by appended them to the end of the `"pattern"` string, separated by "=" characters. If `"nsub"` is not zero, then any substitution strings appended to the end of `"pattern"` are ignored.

Each element of `"subs"` may contain a reference to a token of the form `"$1"`, `"$2"`, etc. The `"$1"` token will be replaced by the part of the test string that matched the first parenthesised sub-string in `"pattern"`. The `"$2"` token will be replaced by the part of the test string that matched the second parenthesised sub-string in `"pattern"`, etc.

---

astFree Free previously allocated memory astFree

Description: This function frees memory that has previouly been dynamically allocated using one of the AST memory function.

Synopsis: `void *astFree( void *ptr )`

Parameters:

ptr

Pointer to previously allocated memory. An error will result if the memory has not previously been allocated by another function in this module. However, a NULL pointer value is accepted (without error) as indicating that no memory has yet been allocated, so that no action is required.

Returned Value:

astFree()

Always returns a NULL pointer.

---

astFreeDouble Free previously double allocated memory astFreeDouble

Description: This function frees memory that has previouly been dynamically allocated using one of the AST memory function. It assumes that the supplied pointer is a pointer to an array of pointers. Each of these pointers is first freed, and then the supplied pointer is freed.

Synopsis: `void *astFreeDouble( void *ptr )`

Parameters:

ptr

Pointer to previously allocated memory. An error will result if the memory has not previously been allocated by another function in this module. However, a NULL pointer value is accepted (without error) as indicating that no memory has yet been allocated, so that no action is required.

Returned Value:

astFreeDouble()
Always returns a NULL pointer.

---

astGrow Allocate memory for an adjustable array astGrow

Description: This function allocates memory in which to store an array of data whose eventual size is unknown. It should be invoked whenever a new array size is determined and will appropriately increase the amount of memory allocated when necessary. In general, it will over-allocate in anticipation of future growth so that the amount of memory does not need adjusting on every invocation.

Synopsis: `void *astGrow( void *ptr, int n, size_t size )`

Parameters:

ptr
Pointer to previously allocated memory (or NULL if none has yet been allocated).

n
Number of array elements to be stored (may be zero).

size
The size of each array element.

Returned Value:

astGrow()
If the memory was allocated successfully, a pointer to the start of the possibly new memory region is returned (this may be the same as the original pointer).

Notes:

- When new memory is allocated, the existing contents are preserved.
- This function does not free memory once it is allocated, so the size allocated grows to accommodate the maximum size of the array (or "high water mark"). Other memory handling routines may be used to free the memory (or alter its size) if necessary.
- If this function is invoked with the global error status set, or if it fails for any reason, the original pointer value is returned and the memory contents are unchanged.

---

astIsDynamic Returns a flag indicating if memory was allocated dynamically astIsDynamic

Description: This function takes a pointer to a region of memory and tests if the memory has previously been dynamically allocated using other functions from this module. It does this by checking for the presence of a "magic" number in the header which precedes the allocated memory. If the magic number is not present (or the pointer is invalid for any other reason), zero is returned. Otherwise 1 is returned.

Synopsis: `int astIsDynamic_( const void *ptr )`

Parameters:

ptr
Pointer to test.

Returned Value:

astIsDynamic()
Non-zero if the memory was allocated dynamically. Zero is returned if the supplied pointer is NULL.

Notes:

- A value of zero is returned if this function is invoked with the global error status set, or if it fails for any reason.

---

astMalloc Allocate memory astMalloc

Description: This function allocates memory in a similar manner to the standard C "malloc" function, but with improved security (against memory leaks, etc.) and with error reporting. It also allows zero-sized memory allocation (without error), resulting in a NULL returned pointer value.

Synopsis: `void *astMalloc( size_t size )`

Parameters:

size
The size of the memory region required (may be zero).

Returned Value:

astMalloc()
If successful, the function returns a pointer to the start of the allocated memory region. If the size allocated is zero, this will be a NULL pointer.

Notes:

- A pointer value of NULL is returned if this function is invoked with the global error status set or if it fails for any reason.

---

astMemCaching Controls whether allocated but unused memory is cached in this module astMemCaching

Description: This function sets a flag indicating if allocated but unused memory should be cached or not. It also returns the original value of the flag.

If caching is switched on or off as a result of this call, then the current contents of the cache are discarded.

Note, each thread has a separate cache. Calling this function affects only the currently executing thread.

Synopsis: `int astMemCaching( int newval )`

Parameters:

newval
The new value for the MemoryCaching tuning parameter (see astTune in objectc.c). If AST__TUNULL is supplied, the current value is left unchanged.

Returned Value:

astMemCaching()
The original value of the MemoryCaching tuning parameter.

---

astRealloc Change the size of a dynamically allocated region of memory astRealloc

Description: This function changes the size of a dynamically allocated region of memory, preserving its contents up to the minimum of the old and new sizes. This may involve copying the contents to a new location, so a new pointer is returned (and the old memory freed if necessary).

This function is similar to the standard C "realloc" function except that it provides better security against programming errors and also supports the allocation of zero-size memory regions (indicated by a NULL pointer).

Synopsis: `void *astRealloc( void *ptr, size_t size )`

Parameters:

ptr
Pointer to previously allocated memory (or NULL if the previous size of the allocated memory was zero).

size
New size required for the memory region. This may be zero, in which case a NULL pointer is returned (no error results). It should not be negative.

Returned Value:

astRealloc()
If the memory was reallocated successfully, a pointer to the start of the new memory region is returned (this may be the same as the original pointer). If size was given as zero, a NULL pointer is returned.

Notes:

- If this function is invoked with the error status set, or if it fails for any reason, the original pointer value is returned and the memory contents are unchanged. Note that this behaviour differs from that of the standard C "realloc" function which returns NULL if it fails.

---

astRemoveLeadingBlanks Remove any leading white space from a string astRemoveLeadingBlanks

Description: This function moves characters in the supplied string to the left in order to remove any leading white space.

Synopsis: `void astRemoveLeadingBlanks( char *string )`

Parameters:

string
Pointer to the string.

---

astSizeOf Determine the size of a dynamically allocated region of memory astSizeOf

Description: This function returns the size of a region of dynamically allocated memory.

Synopsis: `size_t astSizeOf( const void *ptr )`

Parameters:

ptr

Pointer to dynamically allocated memory (or NULL if the size of the allocated memory was zero).

Returned Value:

astSizeOf()

The allocated size. This will be zero if a NULL pointer was supplied (no error will result).

Notes:

- A value of zero is returned if this function is invoked with the global error status set, or if it fails for any reason.

---

astStore Store data in dynamically allocated memory astStore

Description: This function stores data in dynamically allocated memory, allocating the memory (or adjusting the size of previously allocated memory) to match the amount of data to be stored.

Synopsis: `void *astStore( void *ptr, const void *data, size_t size )`

Parameters:

ptr

Pointer to previously allocated memory (or NULL if none has yet been allocated).

data

Pointer to the start of the data to be stored. This may be given as NULL if there are no data, in which case it will be ignored and this function behaves like astRealloc, preserving the existing memory contents.

size

The total size of the data to be stored and/or the size of memory to be allocated. This may be zero, in which case the data parameter is ignored, any previously-allocated memory is freed and a NULL pointer is returned.

Returned Value:

astStore()

If the data were stored successfully, a pointer to the start of the possibly new memory region is returned (this may be the same as the original pointer). If size was given as zero, a NULL pointer is returned.

Notes:

- This is a convenience function for use when storing data of arbitrary size in memory which is to be allocated dynamically. It is appropriate when the size of the data will not change frequently because the size of the memory region will be adjusted to fit the data on every invocation.
- If this function is invoked with the error status set, or if it fails for any reason, the original pointer value is returned and the memory contents are unchanged.

---

astString Create a C string from an array of characters **astString**

Description: This function allocates memory to hold a C string and fills the string with the sequence of characters supplied. It then terminates the string with a null character and returns a pointer to its start. The memory used for the string may later be de-allocated using astFree.

This function is intended for constructing null terminated C strings from arrays of characters which are not null terminated, such as when importing a character argument from a Fortran 77 program.

Synopsis: `char *astString( const char *chars, int nchars )`

Parameters:

chars
Pointer to the array of characters to be used to fill the string.

nchars
The number of characters in the array (zero or more).

Returned Value:

astString()
If successful, the function returns a pointer to the start of the allocated string. If the number of characters is zero, a zero-length string is still allocated and a pointer to it is returned.

Notes:

- A pointer value of NULL is returned if this function is invoked with the global error status set or if it fails for any reason.

---

astStringArray Create an array of C strings from an array of characters **astStringArray**

Description: This function turns an array of fixed-length character data into a dynamicllay allocated array of null-terminated C strings with an index array that may be used to access them.

The array of character data supplied is assumed to hold "nel" adjacent fixed-length strings (without terminating nulls), each of length "len" characters. This function allocates memory and creates a null-terminated copy of each of these strings. It also creates an array of "nel" pointers which point at the start of each of these new strings. A pointer to this index array is returned.

The memory used is allocated in a single block and should later be de-allocated using astFree.

Synopsis: `char **astStringArray( const char *chars, int nel, int len )`

Parameters:

chars
Pointer to the array of input characters. The number of characters in this array should be at least equal to (nel * len).

nel
The number of fixed-length strings in the input character array. This may be zero but should not be negative.

len
The number of characters in each fixed-length input string. This may be zero but should not be negative.

Returned Value:

astStringArray()

A pointer to the start of the index array, which contains "nel" pointers pointing at the start of each null-terminated output string.

The returned pointer should be passed to astFree to de-allocate the memory used when it is no longer required. This will free both the index array and the memory used by the strings it points at.

Notes:

- A NULL pointer will also be returned if the value of "nel" is zero, in which case no memory is allocated.
- A pointer value of NULL will also be returned if this function is invoked with the global error status set or if it fails for any reason.

---

astStringCase Convert a string to upper or lower case astStringCase

Description: This function converts a supplied string to upper or lower case, storing the result in dynamically allocated memory. The astChrCase function is similar, but stores the result in a supplied buffer.

Synopsis: `char *astStringCase( const char string, int upper )`

Parameters:

string

Pointer to the null terminated string to be converted.

upper

If non-zero, the string is converted to upper case. Otherwise it is converted to lower case.

Returned Value:

astStringCase()

If successful, the function returns a pointer to the start of the allocated string. The returned memory should be freed using astFree when no longer needed.

Notes:

- A pointer value of NULL is returned if this function is invoked with the global error status set or if it fails for any reason.

G FITS-WCS Coverage

This appendix gives details of the FitsChan class implementation of the conventions described in the FITS-WCS papers available at http://fits.gsfc.nasa.gov/fits_wcs.html. These conventions are used only if the Encoding attribute of the FitsChan has the value "FITS-WCS" (whether set explicitly or defaulted). It should always be possible for a FrameSet to be read (using the astRead function) from a FitsChan containing a header which conforms to these conventions. However, only those FrameSets which are compatible with the FITS-WCS model can be *written* to a FitsChan using the astWrite function. For instance, if the current Frame of a FrameSet is re-mapped using, say, an arbitrary MathMap then the FrameSet will no longer be compatible with the FITS-WCS model, and so will not be written out successfully to a FitsChan.

The following sub-sections describe the details of the implementation of each of the first four FITS-WCS papers. Here, the term "pixel axes" is used to refer to the FITS pixel coordinates (i.e. the centre of the first image pixel has a value 1.0 on each pixel axis); the term "IWC axes" is used to refer to the axes of the Intermediate World Coordinate system; and the term "WCS axes" is used to refer to the axes of the final physical coordinate system described by the CTYPEi keywords.

G.1 Paper I - General Linear Coordinates

When reading a FrameSet from a FitsChan, these conventions are used if the CTYPEi keyword values within the FitsChan do not conform to the conventions described in later papers, in which case the axes are assumed to be linear. When writing a FrameSet to a FitsChan, these conventions are used for axes which are described by a simple Frame (*i.e.* not a SkyFrame, SpecFrame, *etc.*).

Table 1 describes the use made by AST of each keyword defined by FITS-WCS paper I.

G.1.1 Requirements for a Successful Write Operation

When writing a FrameSet in which the WCS Frame is a simple Frame to a FitsChan, success depends on the Mapping from pixel coordinates (the base Frame in the FrameSet) to the WCS Frame being linear. The write operation will fail if this is not the case.

G.1.2 Use and Choice of CTYPEi keywords

When reading a FrameSet from a FitsChan the CTYPEi values in the FitsChan are used to set the Symbol attributes of the corresponding WCS Frame. The Label attributes of the WCS Frame are set from the CNAMEi keywords, if present in the header. Otherwise they are set from the CTYPEi comments strings in the header, so long as each axis has a unique non-blank comment. Otherwise, the Label attributes are set to the CTYPEi values. The above procedure is over-ridden if the axis types conform to the conventions described in paper II or III, as described below.

When writing a FrameSet to a FitsChan, each CTYPEi value is set to the value of the Symbol attribute of the corresponding axis in the Frame being written. If a value has been set explicitly for the axis Label attribute, it is used as the axis comment (except that any existing comments

Keyword	**Read**	**Write**
WCSAXESa	Ignored.	Set to the number of axes in the WCS Frame - only written if different to NAXIS.
CRVALia	Used to create the pixel to WCS Mapping.	Always written (see “Choice of Reference Point” below).
CRPIXja	Used to create the pixel to WCS Mapping.	Always written (see “Choice of Reference Point” below).
CDELTia	Used to create the pixel to WCS Mapping.	Only written if the CDMatrix attribute of the FitsChan is set to zero.
CROTAi	Used to create the pixel to WCS Mapping.	Only written in FITS-AIPS and FITS-AIPS++ encodings.
CTYPEia	Used to choose the class and attributes of the WCS Frame, and to create the pixel to WCS Mapping (note, “STOKES” and “COMPLEX” axes are treated as unknown linear axes).	Always written (see “Use and Choice of CTYPE keywords” below).
CUNITia	Used to set the Units attributes of the WCS Frame.	Only written if the Units attribute of the WCS Frame has been set explicitly. If so, the Units value for each axis is used as the CUNIT value.
PC$i_j\,a$	Used to create the pixel to WCS Mapping.	Only written if the CDMatrix attribute of the FitsChan is set to zero.
CD$i_j\,a$	Used to create the pixel to WCS Mapping.	Only written if the CDMatrix attribute of the FitsChan is set to a non-zero value.
PVi_ma	Ignored for linear axes.	Not written if the axes are linear.
PSi_ma	Ignored.	Not used.
WCSNAMEa	Used to set the Domain attribute of the WCS Frame.	Only written if the Domain attribute of the WCS Frame has been set explicitly. If so, the Domain value is used as the WCSNAME value.
CRDERia	Ignored.	Not used.
CSYERia	Ignored.	Not used.

Table 1: Use of FITS-WCS Paper I keywords

in the FitsChan take precedence if the keyword value has not changed). The above procedure is over-ridden if the Frame is a SkyFrame or a SpecFrame, in which case the CTYPEi value is derived from the System attribute of the Frame and the nature of the pixel to WCS Mapping according to the conventions of papers II and III, as described below.

G.1.3 Choice of Reference Point

When writing a FrameSet to a FitsChan, the pixel coordinates of the reference point for linear axes (i.e. the CRPIXj values) are chosen as follows:

- If the FrameSet is being written to a FitsChan which previously contained a set of axis descriptions with the same identifying letter, then the previous CRVAL$j$values are converted into the coordinate system of the Frame being written (if possible). These values are then transformed into the pixel Frame, and the closest integer pixel values are used as the CRPIX keywords.
- If the above step could not be performed for any reason, the central pixel is used as the reference point. This requires the image dimensions to be present in the FitsChan in the form of a set of NAXISj keyword values.
- If both the above two steps failed for any axis, then the pixel reference position is set to a value of 1.0 on the pixel axis.

The pixel to WCS Mapping is then used to find the corresponding CRVAL$j$values.

Again, the above procedure is over-ridden if the Frame is a SkyFrame or a SpecFrame, in which case the conventions of papers II and III are used as described below.

G.1.4 Choice of Axis Ordering

When reading a FrameSet from a FitsChan, WCS axis i in the current Frame of the resulting FrameSet corresponds to axis i in the FITS header.

When writing a FrameSet to a FitsChan, the axis ordering for the FITS header is chosen to make the CDi_j or PCi_j matrix predominately diagonal. This means that the axis numbering in the FITS header will not necessarily be the same as that in the AST Frame.

G.1.5 Alternate Axis Descriptions

When reading a FrameSet from a FitsChan which contains alternate axis descriptions, each complete set of axis descriptions results in a single Frame being added to the final FrameSet, connected via an appropriate Mapping to the base pixel Frame. The Ident attribute of the Frame is set to hold the single alphabetical character which is used to identify the set of axis descriptions within the FITS header (a single space is used for the primary axis descriptions).

When writing a FrameSet to a FitsChan, it is assumed that the base Frame represents pixel coordinates, and the current Frame represents the primary axis descriptions. If there are any other Frames present in the FrameSet, an attempt is made to create a complete set of "alternate" set of keywords describing each additional Frame. The first character in the Ident attribute of the Frame is used as the single character descriptor to be appended to the keyword, with the proviso that a given character can only be used once. If a second Frame is found with an Ident attribute which has already been used, its Ident attribute is ignored and the next free character is used instead. Note, failure to write a set of alternate axis descriptions does not result in failure of the entire write operation: the primary axis descriptions are still written, together with any other alternate axis descriptions which can be produced successfully.

G.2 Paper II - Celestial Coordinates

These conventions are used when reading a FrameSet from a FitsChan containing appropriate CTYPEi values, and when writing a FrameSet in which the WCS Frame is a SkyFrame.

Table 2 describes the use made by AST of each keyword whose meaning is defined or extended by FITS-WCS paper II.

G.2.1 Requirements for a Successful Write Operation

When writing a FrameSet in which the WCS Frame is a SkyFrame to a FitsChan, success depends on the following conditions being met:

1. The Mapping from pixel coordinates (the base Frame in the FrameSet) to the WCS SkyFrame includes a WcsMap.
2. The Mapping prior to the WcsMap (*i.e.* from pixel to IWC) is linear.
3. The Mapping after the WcsMap (*i.e.* from native spherical to celestial coordinates) is a spherical rotation for the celestial axes, and linear for any other axes.
4. The TabOK attribute is set to a non-zero positive value in the FitsChan, and the longitude and latitude axes are separable. In this case the Mapping will be described by a pair of 1-dimensional look-up tables, using the "-TAB" algorithm described in FITS-WCS paper III.

If none of the above conditions hold, the write operation will be unsuccessful.

G.2.2 Choice of LONPOLE/LATPOLE

When writing a FrameSet to a FitsChan, the choice of LONPOLE and LATPOLE values is determined as follows:

1. If the projection represented by the WcsMap is azimuthal, then any values set for attributes "PVi_3" and "PVi_4" (where "i" is the index of the longitude axis) within the WcsMap are used as the LONPOLE and LATPOLE values. Reading a FrameSet from a FITS-WCS header results in the original LONPOLE and LATPOLE values being stored within a WcsMap within the FrameSet. Consequently, if a FrameSet is read from a FITS-WCS header and it is subsequently written out to a new FITS-WCS header, the original

Keyword	**Read**	**Write**
CTYPE*ia*	All coordinate systems and projection types listed in paper II are supported (note, "CUBEFACE" axes are treated as unknown linear axes). In addition, "-HPX" (HEALPix) is supported.	Determined by the System attribute of the SkyFrame and the WcsType attribute of the WcsMap within the FrameSet.
CUNIT*ia*	Ignored (assumed to be 'degrees').	Not written.
PV*i_ma*	Used to create the pixel to WCS Mapping (values are stored as attributes of a WcsMap within this Mapping).	Values are obtained from the WcsMap in the pixel to WCS Mapping.
LONPOLE*a*	Used to create the pixel to WCS Mapping. Also stored as a PVi_m attribute for the longitude axis of the WcsMap.	Only written if not equal to the default value defined in paper II (see "Choice of LONPOLE/LATPOLE" below).
LATPOLE*a*	Used to create the pixel to WCS Mapping. Also stored as a PV attribute for the longitude axis of the WcsMap.	Only written if not equal to the default value defined in paper II (see "Choice of LONPOLE/LATPOLE" below).
RADESYS*a*	Used to set the attributes of the SkyFrame. All values supported except that ecliptic coordinates are currently always assumed to be FK5.	Always written. Determined by the System attribute of the SkyFrame.
EQUINOX*a*	Used to set the Equinox attribute of the SkyFrame.	Written if relevant. Determined by the Equinox attribute of the SkyFrame.
EPOCH	Used to set the Equinox attribute of the SkyFrame.	Only written if using FITS-AIPS and FITS-AIPS++ encodings. Determined by the Equinox attribute of the SkyFrame.
MJD-OBS	Used to set the Epoch attribute of the SkyFrame. DATE-OBS is used if MJD-OBS is not present. A default value based on RADESYS and EQUINOX is used if used if DATE-OBS is not present either.	Determined by the Epoch attribute of the SkyFrame. Only written if this attribute has been set to an explicit value (in which case DATE-OBS is also written).

Table 2: Use of FITS-WCS Paper II keywords

LONPOLE and LATPOLE values will usually be used in the new header (the exception being if the WcsMap has been explicitly modified before being written out again). Any extra rotation of the sky is absorbed into the CDi_j or PCi_j matrix (this is possible only if the projection is azimuthal).

2. If the projection represented by the WcsMap is azimuthal but no values have been set for the "PVi_3" and "PVi_4" attributes within the WcsMap, then the default LONPOLE and LATPOLE values are used. This results in no LONPOLE or LATPOLE keywords being stored in the header since default values are never stored. Any extra rotation of the sky is absorbed into the CDi_j or PCi_j matrix (this is possible only if the projection is azimuthal).
3. If the projection represented by the WcsMap is not azimuthal, then the values of LONPOLE and LATPOLE are found by transforming the coordinates of the celestial north pole (*i.e* longitude zero, latitude $+\pi/2$) into native spherical coordinates using the inverse of the Mapping which follows the WcsMap.

G.2.3 User Defined Fiducial Points

When reading a FrameSet from a FitsChan, projection parameters PVi_0, PVi_1 and PVi_2 (for longitude axis "i") are used to indicate a user-defined fiducial point as described in section 2.5 of paper II. This results in a shift of IWC origin being applied *before* the WcsMap which converts IWC into native spherical coordinates. The values of these projection parameters, if supplied, are stored as the corresponding PVi_m attributes of the WcsMap.

When writing a FrameSet to a FitsChan, the PV attributes of the WcsMap determine the native coordinates of the fiducial point (the fixed defaults for each projection described in paper II are used if the PV attributes of the WcsMap have not been assigned a value). The corresponding celestial coordinates are used as the CRVALi keywords and the corresponding pixel coordinates as the CRPIXj keywords.

G.2.4 Common Non-Standard Features

A collection of common non-standard features are supported when reading a FrameSet from a FitsChan, in addition to those embodied within the available encodings of the FitsChan class. These are translated into the equivalent standard features before being used to create a FrameSet. Note, the reverse operation is never performed: it is not possible to produce non-standard features when writing a FrameSet to a FitsChan (other than those embodied in the available encodings of the FitsChan class). The supported non-standard features include:

- EQUINOX keywords with string values equal to a date preceded by the letter B or J (*e.g.* "B1995.0").
- EQUINOX or EPOCH keywords with value zero (these are converted to B1950).
- The IRAF "ZPX" projection is represented by a WcsMap with type of AST__ZPN. Projection parameter values are read from any WATi_nnn keywords, and corresponding PVi_m attributes are set in the WcsMap. The WATi_nnn keywords may specify corrections to the basic ZPN projection by including "lngcor" or "latcor" terms. These are supported if they use half cross-terms, in either simple or Chebyshev representation.

- The IRAF "TNX" projection is represented by a WcsMap with type of AST__TPN (a distorted TAN projection retained within the WcsMap class from an early draft of the FITS-WCS paper II). Projection parameter values are read from any WAT*i_nnn* keywords, and corresponding PV attributes are set in the WcsMap. If the TNX projection cannot be converted exactly into an AST__TPN projection, ASTWARN keywords are added to the FitsChan containing a warning message (but only if the Warnings attribute of the FitsChan is set appropriately). Currently, TNX projections that use half cross-terms, in either simple or Chebyshev representation, are supported.
- "QV" parameters for TAN projections (as produced by AUTOASTROM are renamed to the equivalent "PV" parameters.
- TAN projections that have associated "PV" parameters on the latitude axis are converted to the corresponding TPN (distorted TAN) projections. This conversion can be controlled using the PolyTan attribute of the FitsChan class.

G.3 Paper III - Spectral Coordinates

These conventions are used when reading a FrameSet from a FitsChan which includes appropriate CTYPE*i* values, and when writing a FrameSet in which the WCS Frame is a SpecFrame.

Table 3 describes the use made by AST of each keyword whose meaning is defined or extended by FITS-WCS paper III.

G.3.1 Requirements for a Successful Write Operation

When writing a FrameSet in which the WCS Frame is a SpecFrame to a FitsChan, the write operation is successful only if the Mapping from pixel coordinates (the base Frame in the FrameSet) to the SpecFrame satisfies one of the following conditions:

1. It is linear.
2. It is logarithmic.
3. It is linear if the SpecFrame were to be re-mapped into one of the other spectral systems supported by FITS-WCS paper III.
4. It contains a GrismMap, and the Mapping before the GrismMap (from pixel coordinates to grism parameter) is linear, and the Mapping after the GrismMap is either null or represents a change of spectral system from wavelength (air or vacuum) to one of the supported spectral systems.
5. The TabOK attribute is set to a non-zero positive value in the FitsChan.

If none of the above conditions hold, the write operation will be unsuccessful. Note, if the FitsChan's TabOK attribute is set to a positive non-zero value then any Mapping that does not meet any of the earlier conditions will be written out as a look-up table, using the "-TAB" algorithm described in FITS-WCS paper III. If the TabOK attribute is to zero (the default) or

Keyword	**Read**	**Write**
CTYPE*ia*	All coordinate systems and projection types listed in paper III are supported algorithm (the "-LOG" algorithm may also be applied to non-spectral linear axes; the "-TAB" algorithm requires the TabOK attribute to be set in the FitsChan).	Determined by the System attribute of the SpecFrame and the nature of the pixel to SpecFrame Mapping.
CUNIT*ia*	Used to set the Units attribute of the SpecFrame (note, SpecFrames always have an "active" Units attribute (see astSetActiveUnit).	Always written.
PV*i_ma*	Used to create the pixel to WCS Mapping (values are stored as attributes of a GrismMap).	Set from the attributes of the GrismMap, if present, and if set explicitly.
SPECSYS*a*	Used to set the StdOfRest attribute of the SpecFrame (all systems are supported except CMBDIPOL).	Set from the StdOfRest attribute of the SpecFrame, but only if it has been set explicitly.
SSYSOBS*a*	Ignored.	Never written.
OBSGEO-X/Y/Z	Used to set the ObsLon and ObsLat attributes of the Frame (the observers height above sea level is ignored).	Set from the ObsLon and ObsLat attributes of the Frame, if they have been set explicitly (it is assumed that the observer is at sea level).
MJD-AVG	Used to set the Epoch attributes of the SpecFrame.	Set from the Epoch attribute of the SpecFrame, if it has been set explicitly.
SSYSSRC*a*	Used to set the SourceVRF attribute of the SpecFrame (all systems are supported except CMBDIPOL).	Set from the SourceVRF attribute of the SpecFrame.
ZSOURCE*a*	Used to set the SourceVel attribute of the SpecFrame (the SourceVRF attribute is first set to the system indicated by the SSYSSRC keyword, and the ZSOURCE value is then converted to an apparent radial velocity and stored as the SourceVel attribute).	Set from the SourceVel attribute of the SpecFrame, if it has been set explicitly (the SourceVel value is first converted from apparent radial velocity to redshift).
VELOSYS*a*	Ignored.	Set from the attributes of the SpecFrame that define the standard of rest and the observers position.
RESTFRQ*a*	Used to set the RestFreq attribute of the SpecFrame.	Set from the RestFreq attribute of the SpecFrame, but only if the System attribute is not set to "WAVE", "VOPT", "ZOPT" or "AWAV", and only if RestFreq has been set explicitly.
RESTWAV*a*	Used to set the RestFreq attribute of the SpecFrame (after conversion from wavelength to frequency).	Set from the RestFreq attribute of the SpecFrame (after conversion), but only if the System attribute is set to "WAVE", "VOPT", "ZOPT" or "AWAV", and only if RestFreq has been set explicitly.

negative in the FitsChan, then the write operation will be unsuccessful unless one of the eaerlier conditions is met.[39]

G.3.2 Common Non-Standard Features

The following non-standard features are supported when reading spectral axes from a FitsChan:

- Conversion of "-WAV", "-FRQ" and "-VEL" algorithm codes (specified in early drafts of paper III) to the corresponding "-X2P" form.
- Conversion of "RESTFREQ" to "RESTFRQ"

G.4 Paper IV - Coordinate Distortions

This paper proposes that an additional 4 character code be appended to the end of the CTYPE*i* keyword to specify the nature of any distortion away from the basic algorithm described by the first 8 characters of the CTYPE*i* value. Currently AST ignores all such codes when reading a FrameSet from a FitsChan (except for the "-SIP" code defined by the Spitzer Space Telescope project - see below). This means that a FrameSet can still be read from such headers, but the Mapping which gives the WCS position associated with a given pixel position will reflect only the basic algorithm and will not include the effects of the distortion.

If such a FrameSet is then written out to a FitsChan, the resulting CTYPE*i* keywords will include no distortion code.

G.4.1 The "-SIP" distortion code

The Spitzer Space Telescope project (http://www.spitzer.caltech.edu/) has developed its own system for encoding 2-dimensional image distortion within a FITS header, based on the proposals of paper IV. A description of this system is available in http://ssc.spitzer.caltech.edu/postbcd/doc/shupeADASS.pd In this system, the presence of distortion is indicated by appending the distortion code "-SIP" to the CTYPE*i* keyword values for the celestial axes. The distortion takes the form of a polynomial function which is applied to the pixel coordinates, after subtraction of the CRPIX*j* values.

This system is a strictly 2 dimensional system. When reading a FrameSet from a FitsChan which includes the "-SIP" distortion code, AST assumes that it is only applied to the first 2 WCS axes in a FITS header (i.e. CTYPE1 and CTYPE2). If the "-SIP" distortion code is attached to other axes, it will be ignored. The distortion itself is represented by a PolyMap within the resulting FrameSet.

If a FrameSet is read from a FitsChan which includes "-SIP" distortion, and an attempt is then made to write this FrameSet out to a FitsChan, the write operation will fail unless the distortion is insignificant (*i.e.* is so small that the tests for linearity built into AST are passed). In this case, no distortion code will be appended to the resulting CTYPE*i* keyword values.

---

[39] If the -TAB algorithm is used, the positive value of the TabOK attribute is used as the table version number (the EXTVER header) in the associated FITS binary table.

H Changes and New Features

H.1 Changes Introduced in V1.1

The following describes the most significant changes which occurred in the AST library between versions V1.0 and V1.1 (not the most recent version):

1. A new "How To…" section (§3) has been added to this document. It contains simple recipies for performing commonly-required operations using AST.
2. A new astUnformat function has been provided to read formatted coordinate values for the axes of a Frame (§7.8). In essence, this function is the inverse of astFormat. It may be used to decode user-supplied formatted values representing coordinates, turning them into numerical values for processing. Celestial coordinates may also be read using this function (§8.7) and free-format input is supported.
3. The Format attribute string used by a SkyFrame when formatting celestial coordinate values now allows the degrees/hours field to be omitted, so that celestial coordinates may be given in (*e.g.*) arc-minutes and/or arc-seconds (§8.6). As a result, the degrees/hours field is no longer included by default. A new "t" format specifier has been introduced (see the Format attribute) to allow minutes and/or seconds of time to be specified if required.
4. A new function astMapBox has been introduced. This allows you to find the extent of a "bounding box" which just encloses another box after it has been transformed by a Mapping. A typical use might be to calculate the size which an image would have if it were transformed by the Mapping.
5. A new class of Object, the IntraMap, has been introduced (§20). This is a specialised form of Mapping which encapsulates a privately-defined coordinate transformation function (*e.g.* written in C) so that it may be used like any other AST Mapping. This allows you to create Mappings that perform any conceivable coordinate transformation.
6. The internal integrity of a FrameSet is now automatically preserved whenever changes are made to any attributes which affect the current Frame (either by setting or clearing their values). This is accomplished by appropriately re-mapping the current Frame to account for any change to the coordinate system which it represents (§14.6).
7. The internal structure of a FrameSet is now automatically tidied to eliminate redundant nodes whenever any of its Frames is removed or re-mapped. Automatic simplification of any compound Mappings which result may also occur. The effect of this change is to prevent the accumulation of unnecessary structure in FrameSets which are repeatedly modified.
8. Some improvements have been made to the algorithms for simplifying compound Mappings, as used by astSimplify.
9. The textual representation used for some Objects (*i.e.* when they are written to a Channel) has changed slightly, but remains compatible with earlier versions of AST.
10. Interfaces to the internal functions and macros used by AST for handling memory and error conditions are now provided *via* the "ast.h" header file. This is for the benefit of those writing (*e.g.*) new graphics interfaces for AST.

11. A problem has been fixed which could result when using astRead to read FITS headers in which the CDELT value is zero. Previously, this could produce a Mapping whose inverse transformation was not defined and this could unnecessarily restrict the use to which it could be put. The problem has been overcome by supplying a suitable small CDELT value for FITS axes which have only a single pixel.

12. A bug has been fixed which could occasionally cause a MatrixMap to be used with the wrong Invert attribute value when it forms part of a compound Mapping which is being simplified using astSimplify.

13. A problem has been fixed which could prevent tick marks being drawn on a coordinate axis close to a singularity in the coordinate system.

H.2 Changes Introduced in V1.2

The following describes the most significant changes which occurred in the AST library between versions V1.1 and V1.2 (not the most recent version):

1. A new function, astPolyCurve, has been introduced to allow more efficient plotting of multiple geodesic curves (§21.3).

2. A new set of functions, astResample<X>, has been introduced to perform resampling of gridded data such as images (*i.e.* re-gridding) under the control of a geometrical transformation specified by a Mapping.

3. The command-line options "−pgp" and "−pgplot", which were previously synonymous when used with the "ast_link" and "ast_link_adam" commands, are no longer synonymous. The option "−pgp" now causes linking with the Starlink version of PGPLOT (which uses GKS to generate its output), while "−pgplot" links with the standard (or "native") version of PGPLOT.

4. The function astMapBox has been changed to execute more quickly, although this has been achieved at the cost of some loss of robustness when used with difficult Mappings.

5. A new value of "FITS-IRAF" has been introduced for the Encoding attribute of a FitsChan. This new encoding provides an interim solution to the problem of storing coordinate system information in FITS headers, until the proposed new FITS-WCS standard becomes stable.

6. When a FrameSet is created from a set of FITS header cards (by reading from a FitsChan using a "foreign" encoding), the base Frame of the resulting FrameSet now has its Domain attribute set to "GRID". This reflects the fact that this Frame represents FITS data grid coordinates (equivalent to FITS pixel coordinates—see §7.13). Previously, this Domain value was not set.

7. astFindFits now ignores trailing spaces in its keyword template.

8. astPutFits now recognises "D" and "d" as valid exponent characters in floating point numbers.

9. The FitsChan class is now more tolerant of common minor violations of the FITS standard.

10. The FitsChan class now incorporates an improved test for the linearity of Mappings, allowing more reliable conversion of AST data into FITS (using "foreign" FITS encodings).

11. Some further improvements have been made to the algorithms for simplifying compound Mappings, as used by astSimplify.

12. A new UnitRadius attribute has been added to the SphMap class. This allows improved simplification of compound Mappings (CmpMaps) involving SphMaps and typically improves performance when handling FITS world coordinate information.

13. A MatrixMap no longer propagates input coordinate values of AST__BAD automatically to all output coordinates. If certain output coordinates do not depend on the affected input coordinate(s) because the relevant matrix elements are zero, then they may now remain valid.

14. A minor bug has been corrected which could cause certain projections which involve half the celestial sphere to produce valid coordinates for the other (unprojected) half of the sphere as well.

15. A bug has been fixed which could occasionally cause astConvert to think that conversion between a CmpFrame and another Frame was possible when, in fact, it wasn't.

H.3 Changes Introduced in V1.3

The following describes the most significant changes which occurred in the AST library between versions V1.2 and V1.3 (not the most recent version):

1. A new set of functions, astResample<X>, has been introduced to provide efficient resampling of gridded data, such as spectra and images, under the control of a geometrical transformation specified by a Mapping. A variety of sub-pixel interpolation schemes are supported.

2. A new class, PcdMap, has been introduced. This is a specialised form of Mapping which implements 2-dimensional pincushion or barrel distortion.

3. A bug has been fixed which could cause a FitsChan to produce too many digits when formatting floating point values for inclusion in a FITS header if the numerical value was in the range -0.00099999. . . to -0.0001.

4. A bug has been fixed which could cause a FitsChan to lose the comment associated with a string value in a FITS header.

5. A FitsChan now reports an error if it reads a FITS header which identifies a non-standard sky projection (previously, this was accepted without error and a Cartesian projection used instead).

6. A bug has been fixed which could prevent conversion between the coordinate systems represented by two CmpFrames. This could only occur if the CmpFrames contained a relatively large number of nested Frames.

7. Further improvements have been made to the simplification of compound Mappings, including fixes for several bugs which could cause indefinite looping or unwanted error messages.

8. Some memory leaks have been fixed.

9. A small number of documentation errors have been corrected.

H.4 Changes Introduced in V1.4

The following describes the most significant changes which have occurred in the AST library between versions V1.3 and V1.4 (not the most recent version):

1. A new MathMap class has been introduced. This is a form of Mapping that allows you to define coordinate transformations in a flexible and transportable way using arithmetic operations and mathematical functions similar to those available in C.

2. **WARNING—INCOMPATIBLE CHANGE.** Transformation functions used with the IntraMap class (see, for example, astIntraReg) now require a "this" pointer as their first parameter. **Existing implementations will not continue to work correctly with this version of AST unless this parameter is added.** There is no need for existing software to make use of this pointer, but it must be present.

   This change has been introduced so that transformation functions can gain access to IntraMap attributes.

3. A new IntraFlag attribute has been added to the IntraMap class. This allows the transformation functions used by IntraMaps to adapt to produce the required transformation on a per-IntraMap basis (§20.9).

4. The Plot attributes MajTickLen and MinTickLen, which control the length of major and minor tick marks on coordinate axes, may now be subscripted using an axis number. This allows tick marks of different lengths to be used on each axis. It also allows tick marks to be suppressed on one axis only by setting the length to zero.

5. The value of the Plot attribute NumLab, which controls the plotting of numerical labels on coordinate axes, no longer has any effect on whether labelling of a coordinate grid is interior or exterior (as controlled by the Labelling attribute).

6. The FitsChan class now provides some support for the IRAF-specific "ZPX" sky projection, which is converted transparently into the equivalent FITS "ZPN" projection (see the description of the Encoding attribute for details).

7. The FitsChan class now recognises the coordinate system "ICRS" (International Celestial Reference System) as equivalent to "FK5". This is an interim measure and full support for the (exceedingly small) difference between ICRS and FK5 will be added at a future release.

   Note that "ICRS" is not yet recognised as a coordinate system by other classes such as SkyFrame, so this change only facilitates the importation of foreign data.

8. A bug in the FitsChan class has been fixed which could result in longitude values being incorrect by 180 degrees when using cylindrical sky projections, such as the FITS "CAR" projection.
9. A bug in the FitsChan class has been fixed which could result in the FITS sky projection parameters ProjP(0) to ProjP(9) being incorrectly named PROJP1 to PROJP10 when written out as FITS cards.
10. A bug in the FitsChan class has been fixed which could cause confusion between the FITS-IRAF and FITS-WCS encoding schemes if both a CD matrix and a PC matrix are erroneously present in a FITS header.
11. Some minor memory leaks have been fixed.
12. A small number of documentation errors have been corrected.

H.5 Changes Introduced in V1.5

The following describes the most significant changes which have occurred in the AST library between versions V1.4 and V1.5 (not the most recent version):

1. The FitsChan class has been modified to support the latest draft FITS WCS standard, described in the two papers "Representation of world coordinates in FITS" (E.W. Greisen and M. Calabretta, dated 30th November, 1999), and "Representation of celestial coordinates in FITS" (M. Calabretta and E.W. Greisen, dated 24th September, 1999). These are available at http://www.cv.nrao.edu/fits/documents/wcs/wcs.html.

   The FITS-WCS encoding now uses these updated conventions. The main changes are:

   - Rotation and scaling of pixel axes is now represented by a matrix of `CDj_i` keywords instead of a combination of `PCjjjiii` and `CDELTj` keywords.
   - Projection parameters are now associated with particular axes and are represented by `PVi_m` keywords instead of the `PROJPm` keywords.
   - The tangent plane projection ("TAN") can now include optional polynomial correction terms.
   - An entire set of keywords must be supplied for each set of secondary axis descriptions, and each such keyword must finish with a single character indicating which set it belongs to. This means that keywords which previously occupied eight characters have been shorten to seven to leave room for this extra character. Thus `LONGPOLE` has become `LONPOLE` and `RADECSYS` has become `RADESYS`.

2. Two new encodings have been added to the FitsChan class:

   FITS-PC This encoding uses the conventions of the now superseded FITS WCS paper by E.W. Greisen and M. Calabretta which used keywords `CDELTj` and `PCjjjiii` to describe axis scaling and rotation. These are the conventions which were used by the FITS-WCS encoding prior to version 1.5 of AST. This encoding is provided to allow existing data which use these conventions to be read. It should not in general be used to create new data.

FITS-AIPS This encoding is based on the conventions described in the document "Non-linear Coordinate Systems in AIPS" by Eric W. Greisen (revised 9th September, 1994 and available by ftp from fits.cv.nrao.edu /fits/documents/wcs/aips27.ps.Z). This encoding uses `CROTAi` and `CDELTi` keywords to describe axis rotation and scaling.

3. The FitsChan class now provides some support for the IRAF-specific "TNX" sky projection, which is converted transparently into the equivalent FITS "TAN" projection (see the description of the Encoding attribute for details).
4. FrameSets originally read from a DSS encoded FITS header can now be written out using the FITS-WCS encoding (a TAN projection with correction terms will be used) in addition to the DSS encoding. The reverse is also possible: FrameSets originally read from a FITS-WCS encoded FITS header and which use a TAN projection can now be written out using the DSS encoding.
5. The algorithm used by the FitsChan class to verify that a FrameSet conforms to the FITS-WCS model has been improved so that FrameSets including more complex mixtures of parallel and serial Mappings can be written out using the FITS-WCS encoding.
6. The FitsChan class has been changed so that long strings included in the description of an Object can be saved and restored without truncation when using the NATIVE encoding. Previously, very long Frame titles, mathematical expressions, *etc.* were truncated if they exceeded the capacity of a single FITS header card. They are now split over several header cards so that they can be restored without truncation. Note, this facility is only available when using NATIVE encoding.
7. The FitsChan class has a new attribute called Warnings which can be used to select potentially dangerous conditions under which warnings should be issued. These conditions include (for instance) unsupported features within non-standard projections, missing keywords for which default values will be used, *etc.*
8. The WcsMap class has been changed to support the changes made to the FITS-WCS encoding in the FitsChan class:
   - Projection parameters are now associated with a particular axis and are specified using a new set of attributes called PVj_m. Here, "j" is the index of an axis of WcsMap, and "m" is the index of the projection parameter.
   - The old attributes ProjP(0) to ProjP(9) are still available but are now deprecated in favour of the new PVj_m attributes. They are interpreted as aliases for PV(axlat)_0 to PV(axlat)_9, where "axlat" is the index of the latitude axis.
   - The GLS projection projection has been renamed as SFL, but the AST__GLS type has been retained as an alias for AST__SFL.

H.6 Changes Introduced in V1.6

The following describes the most significant changes which have occurred in the AST library between versions V1.5 and V1.6:

1. The C interface to several methods (astTranN, astMark and astPolyCurve) have been changed to make them easier to call from C++. Parameters which previously had type

"double (*)[]" have been changed to the simpler "double *". Using the old types may result in non-fatal compiler warnings, but should not change the behaviour of the methods.

2. A bug has been fixed in the Plot class which could cause groups of tick marks to be skipped when using very small gaps.
3. A bug has been fixed in the Plot class which could cause axes to be labeled outside the visible window, resulting in no axes being visible.
4. The FITS-WCS encoding used by the FitsChan class now includes the WCSNAME keyword. When creating a FrameSet from FITS headers, the values of the WCSNAME keywords are now used as the Domain names for the corresponding Frames in the returned FrameSet. When writing a FrameSet to a FITS header the Domain names of each Frame are stored in WCSNAME keywords in the header.
5. The FITS-WCS encoding used by the FitsChan class now attempts to retain the identification letter associated with multiple axis descriptions. When reading a FrameSet from a FITS header, the identification letter is stored in the Ident attribute for each Frame. When writing a FrameSet to a FITS header, the identification letter is read from the Ident attribute of each Frame. The letter to associate with each Frame can be changed by assigning a new value to the Frame's Ident attribute.
6. The FITS-WCS, FITS-PC, FITS-IRAF and FITS-AIPS encodings used by the FitsChan class now create a SkyFrame with the System attribute set to "Unknown" if the CTYPE keywords in the supplied header refers to an unknown celestial coordinate system. Previously, a Frame was used instead of a SkyFrame.
7. The FITS-WCS, FITS-PC, FITS-IRAF and FITS-AIPS encodings used by the FitsChan class no longer report an error if the FITS header contains no CTYPE keywords. It is assumed that a missing CTYPE keyword implies that the world coordinate system is linear and identically equal to "intermediate world coordinates".
8. The new value "noctype" is now recognized by the Warnings attribute of the FitsChan class. This value causes warnings to be issued if CTYPE keywords are missing from foreign encodings.
9. A new attribute called AllWarnings has been added to the FitsChan class. This is a read-only, space separated list of all the known condition names which can be specified in the Warnings attribute.
10. The FitsChan class now attempts to assigns a Title to each Frame in a FrameSet read using a foreign encoding. The Title is based on the Domain name of the Frame. If the Frame has no Domain name, the default Title supplied by the Frame class is retained.
11. The FitsChan class uses the comments associated with CTYPE keywords as axis labels when reading a foreign encoding. This behaviour has been modified so that the default labels provided by the Frame class are retained (instead of using the CTYPE comments) if any of the CTYPE comments are identical.
12. A new "interpolation" scheme identified by the symbolic constant AST__BLOCKAVE has been added to the AST_RESAMPLE<X> set of functions. The new scheme calculates each output pixel value by finding the mean of the input pixels in a box centred on the output pixel.

13. The SkyFrame class can now be used to represent an arbitrary spherical coordinate system by setting its System attribute to "Unknown".
14. The indices of the latitude and longitude axes of a SkyFrame can now be found using new read-only attributes LatAxis and LonAxis. The effects of any axis permutation is taken into account.
15. A new attribute called Ident has been added to the Object class. This serves the same purpose as the existing ID attribute, but (unlike ID) its value is transferred to the new Object when a copy is made.
16. A bug has been fixed which could prevent complex CmpFrames behaving correctly (for instance, resulting in the failure of attempts to find a Mapping between a CmpFrame and itself).

H.7 Changes Introduced in V1.7

The following describes the most significant changes which have occurred in the AST library between versions V1.6 and V1.7:

1. The Frame class has a new method called astAngle which returns the angle subtended by two points at a third point within a 2 or 3 dimensional Frame.
2. The Frame class has a new method called astOffset2 which calculates a position which is offset away from a given starting point by a specified distance along a geodesic curve which passes through the starting point at a given position angle. It can only be used with 2-dimensional Frames.
3. The Frame class has a new method called astAxDistance which returns the increment between two supplied axis values. For axes belonging to SkyFrames, the returned value is normalized into the range $\pm\pi$.
4. The Frame class has a new method called astAxOffset which returns an axis value a given increment away from a specified axis value. For axes belonging to SkyFrames, the returned value is normalized into the range $\pm\pi$ (for latitude axes) or zero to 2π (for longitude axes).
5. The Plot class has a new method called astGenCurve which allows generalised user-defined curves to be drawn. The curve is defined by a user-supplied Mapping which maps distance along the curve into the corresponding position in the current Frame of the Plot. The new method then maps these current Frame position into graphics coordinates, taking care of any non-linearities or discontinuities in the mapping.
6. The Plot class has a new method called astGrfSet which allows the underlying primitive graphics functions to be selected at run-time. Previously, the functions used by the Plot class to produce graphics could only be selected at link-time, using the options of the ast_link command. The new Plot method allows an application to over-ride the functions established at link-time, by specifying alternative primitive graphics routines. In addition, the two new Plot methods astGrfPush and astGrfPop allow the current graphics routines to be saved and restore on a first-in-last-out stack, allowing temporary changes to be made to the set of registered graphics routines.

7. The DrawAxes attribute of the Plot class can now be specified independantly for each axis, by appending the axis index to the end of the attribute name.

8. A bug has been fixed in the Plot class which could result in axis labels being drawn on inappropriate edges of the plotting box when using "interior" labelling.

9. A bug has been fixed in the IntraMap class which could cause IntraMaps to be corrupted after transforming any points.

10. Bugs have been fixed in the FitsChan class which could cause inappropriate ordering of headers within a FitsChan when writing or reading objects using NATIVE encodings.

11. A bug has been fixed in the FitsChan class which could cause the celestial longitude of a pixel to be estimated incorrectly by 180 degrees if the reference point is at either the north or the south pole.

H.8 Changes Introduced in V1.8-2

The following describes the most significant changes which have occurred in the AST library between versions V1.7 and V1.8-2:

1. The SkyFrame class has a new attribute called NegLon which allows longitude values to be displayed in the range $-\pi$ to $+\pi$, instead of the usual range zero to $2.\pi$.

2. Some new functions (astAngle, astAxAngle, astResolve, astOffset2, astAxOffset, astAxDistance) have been added to the Frame class to allow navigation of the coordinate space to be performed without needing to know the underlying geometry of the co-ordinate system (for instance, whether it is Cartesian or spherical).

   Note, version 1.8-1 contained many of these facilities, but some have been changed in version 1.8-2. Particularly, positions angles are now referred to the second Frame axis for *all* classes of Frames (including SkyFrames), and the astBear function has been replaced by astAxAngle.

H.9 Changes Introduced in V1.8-3

The following describes the most significant changes which occurred in the AST library between versions V1.8-2 and V1.8-3:

1. A new method called astDecompose has been added to the Mapping class which enables pointers to be obtained to the component parts of CmpMap and CmpFrame objects.

2. Functions within proj.c and wcstrig.c have been renamed to avoid name clashes with functions in more recent versions of Mark Calabretta's wcslib library.

H.10 Changes Introduced in V1.8-4

The following describes the most significant changes which occurred in the AST library between versions V1.8-3 and V1.8-4:

1. The FitsChan class has a new attribute called DefB1950 which can be used to select the default reference frame and equinox to be used if a FitsChan with foreign encoding contains no indication of the reference frame or equinox.
2. A bug has been fixed in the FitsChan class which could prevent astWrite from creating a set of FITS headers from an otherwise valid FrameSet, when when using FITS-AIPS encoding.
3. A bug has been fixed in the FitsChan class which could cause astRead to mis-interpret the FITS CROTA keyword when using FITS-AIPS encoding.

H.11 Changes Introduced in V1.8-5

The following describes the most significant changes which occurred in the AST library between versions V1.8-4 and V1.8-5:

1. The Plot class defines new graphical elements Axis1, Axis2, Grid1, Grid2, NumLabs1, NumLabs2, TextLab1, TextLab2, Ticks1 and Ticks2. These allow graphical attributes (colour, width, etc) to be set for each axis individually. Previously, graphical attributes could only be set for both axes together, using graphical elements Axes, Grid, NumLabs, TextLabs and Ticks.

H.12 Changes Introduced in V1.8-7

The following describes the most significant changes which occurred in the AST library between versions V1.8-5 and V1.8-7:

1. A new attribute called CarLin has been added to the FitsChan class which controls the way CAR projections are handled when reading a FrameSet from a non-native FITS header. Some FITS writers use a CAR projection to represent a simple linear transformation between pixel coordinates and celestial sky coordinates. This is not consistent with the definition of the CAR projection in the draft FITS-WCS standard, which requires the resultant Mapping to include a 3D rotation from native spherical coordinates to celestial spherical coordinates, thus making the Mapping non-linear. Setting CarLin to 1 forces astRead to ignore the FITS-WCS standard and treat any CAR projections as simple linear Mappings from pixel coordinates to celestial coordinates.
2. A bug has been fixed which could result in axis Format attributes set by the user being ignored under certain circumstances.
3. A bug in the way tick marks positions are selected in the Plot class has been fixed. This bug could result in extra ticks marks being displayed at inappropriate positions. This bug manifested itself, for instance, if the Mapping represented by the Plot was a simple Cartesian to Polar Mapping. In this example, the bug caused tick marks to be drawn at negative radius values.

4. A bug has been fixed which could prevent attribute settings from being read correctly by astSet, etc., on certain platforms (MacOS, for instance).

H.13 Changes Introduced in V1.8-8

The following describes the most significant changes which occurred in the AST library between versions V1.8-7 and V1.8-8:

1. A bug has been fixed in the FitsChan class which could cause problems when creating a FrameSet from a FITS header containing WCS information stored in the form of Digitised Digitised Sky Survey (DSS) keywords. These problems only occurred for DSS fields in the southern hemisphere, and resulted in pixel positions being mapped to sky positions close to the corresponding *northern* hemispshere field.
2. A new method called astBoundingBox has been added to the Plot class. This method returns the bounding box of the previous graphical output produced by a Plot method.
3. A new attribute called Invisible has been added to the Plot class which suppresses the graphical output normally produced by Plot methods. All the calculations needed to produce the normal output are still performed however, and so the bounding box returned by the new astBoundingBox method is still usable.
4. Bugs have been fixed related to the appearance of graphical output produced by the Plot class. These bugs were to do with the way in which graphical elements relating to a specific axis (e.g. `Colour(axis1)`, etc.) interacted with the corresponding generic element (e.g. `Colour(axes)`, etc.).

H.14 Changes Introduced in V1.8-13

The following describes the most significant changes which occurred in the AST library between versions V1.8-8 and V1.8-13:

1. The FitsChan class has been modified so that LONPOLE keywords are only produced by astWrite when necessary. For zenithal projections such as TAN, the LONPOLE keyword can always take its default value and so is not included in the FITS header produced by astWrite. Previously, the unnecessary production of a LONPOLE keyword could prevent FrameSets being written out using encodings which do not support the LONPOLE keyword (such as FITS-IRAF).
2. The FitsChan class has been modified to retain leading and trailing spaces within COMMENT cards.
3. The FitsChan class has been modified to only use CTYPE comments as axis labels if all non-celestial axes have unique non-blank comments (otherwise the CTYPE keyword values are used as labels).
4. The FitsChan class has been modified so that it does not append a trailing "Z" character to the end of DATE-OBS keyword values.

5. The FitsChan class has been modified to use latest list of FITS-WCS projections, as described in the FITS-WCS paper II, "Representations of celestial coordinates in FITS" (Calabretta & Greisen, draft dated 23 April 2002). Support has been retained for the polynomial correction terms which previous drafts have allowed to be associated with TAN projections.

6. The WcsMap class has additional projection types of AST__TPN (which implements a distorted TAN projection) and AST__SZP. The AST__TAN projection type now represents a simple TAN projection and has no associated projection parameters. In addition, the usage of projection parameters has been brought into line with the the FITS-WCS paper II.

7. The WcsMap class has been modified so that a "get" operation on a projection parameter attribute will return the default value defined in the FITS-WCS paper II if no value has been set for the attribute. Previously, a value of AST__BAD was returned in such a situation.

8. The Frame class has new attributes Top(axis) and Bottom(axis) which allow a "plottable range" to be specified for each Frame axis. The grid produced by the astGrid method will not extend beyond these limits.

H.15 Changes Introduced in V2.0

Note, Frame descriptions created using AST V2.0 will not be readable by applications linked with earlier versions of AST. This applies to Frame descriptions created using:

- the Channel class
- the FitsChan class if the NATIVE Encoding is used
- the astShow function

Applications must be re-linked with AST V2.0 in order to be able to read Frame descriptions created by AST v2.0.

The following describes the most significant changes which have occurred in the AST library between versions V1.8-13 and V2.0 (the current version):

1. The default value for the Domain attribute provided by the CmpFrame class has been changed from "CMP" to a string formed by concatenating the Domain attributes of the two component Frames, separated by a minus sign. If both component Domains are blank, then the old default of "CMP" is retained for the CmpFrame Domain.

2. The implementation of the astWrite function within the FitsChan class has been modified. It will now attempt to produce a set of FITS header cards to describe a FrameSet even if the number of axes in the Current Frames is greater than the number in the Base Frame (that is, if there are more WCS axes than pixel axes). This has always been possible with NATIVE encoding, but has not previously been possible for foreign encodings. The WCSAXES keyword is used to store the number of WCS axes in the FITS header.

3. Another change to the astWrite function within the FitsChan class is that the ordering of "foreign" axes (*i.e.* CTYPE keywords) is now chosen to make the CD (or PC) matrix as diagonal as possible - any element of axis transposition is removed by this re-ordering as recommended in FITS-WCS paper I. Previously the ordering was determined by the order of the axes in the Current Frame of the supplied FrameSet. This change does not affect NATIVE encoding.

4. Support for spectral coordinate systems has been introduced throught the addition of two new classes, SpecFrame and SpecMap. The SpecFrame is a 1-dimensional Frame which can be used to describe positions within an electromagnetic spectrum in various systems (wavelength, frequency, various forms of velocity, *etc.*) and referred to various standards of rest (topocentric, geocentric, heliocentric LSRK, *etc.*). The SpecMap is a Mapping which can transform spectral axis values between these various systems and standards of rest. Note, FitsChans which have a foreign encoding (*i.e.* any encoding other than NATIVE) are not yet able to read or write these new classes.

5. Facilities have been added to the Frame class which allow differences in axis units to be taken into account when finding a Mapping between two Frames. In previous versions of AST, the Unit attribute was a purely descriptive item intended only for human readers - changing the value of Unit made no difference to the behaviour of the Frame. As of version 2.0, the Unit attribute can influence the nature of the Mappings between Frames. For instance, if the astFindrame or astConvert method is used to find the Mapping between an Axis with Unit set to "m" and another Axis with Unit set to "km", then the method will return a ZoomMap which introduces a scaling factor of 0.001 between the two axes. These facilities assume that units are specified following the rules included in FITS-WCS paper I (*Representation of World Coordinates in FITS*, Greisen & Calabretta).

   In order to minimise the risk of breaking existing software, the default behaviour for simple Frames is to ignore the Unit attribute (*i.e.* to retain the previous behaviour). However, the new Frame method astSetActiveUnit may be used to "activate" (or deactivate) the new facilities within a specific Frame. Note, the new SpecFrame class is different to the simple Frame class in that the new facilities for handling units are always active within a SpecFrame.

6. The System and Epoch attributes fo the SkyFrame class have been moved to the parent Frame class. This enables all sub-classes of Frame (such as the new SpecFrame class) to share these attributes, and to provide suitable options for each class.

7. The Frame class has a new attribute called AlignSystem, which allows control over the alignment process performed by the methods astFindFrame and astConvert.

8. The CmpFrame class has been modified so that attributes of a component Frame can be accessed without needing to extract the Frame first. To do this, append an axis index to the end of the attribute name. For instance, if a CmpFrame contains a SpecFrame and a SkyFrame (in that order), then the StdOfRest attribute of the SpecFrame can be referred to as the "StdOfRest(1)" attribute of the CmpFrame. Likewise, the Equinox attribute of the SkyFrame can be accessed as the "Equinox(2)" (or equivalently "Equinox(3)") attribute of the CmpFrame. The "System(1)" attribute of the CmpFrame will refer to the System attribute of the SpecFrame, whereas the "System(2)" and "System(3)" attributes of the CmpFrame will refer to the System attribute of the SkyFrame (the "System" attribute without an axis specifier will refer to the System attribute of the CmpFrame as a whole,

since System is an attribute of all Frames, and a CmpFrame is a Frame and so has its own System value which is independant of the System attributes of its component Frames).

9. The algorithms used by the Plot class for determining when to omit overlapping axis labels, and the abbreviation of redundant leading fields within sexagesimal axis labels, have been improved to avoid some anomolous behaviour in previous versions.
10. The curve drawing algorithm used by the Plot class has been modified to reduce the chance of it "missing" small curve sections, such as may be produced if a grid line cuts across the plot very close to a corner. Previously, these missed sections could sometimes result in axis labels being omitted.
11. A new function (astVersion) has been added to return the version of the AST library in use.
12. Bugs have been fixed in the Plot class which caused serious problems when plotting high precision data. These problems could range from the omission of some tick marks to complete failure to produce a plot.

Programs which are statically linked will need to be re-linked in order to take advantage of these new facilities.

H.16 Changes Introduced in V3.0

The following describes the most significant changes which occurred in the AST library between versions V2.0 and V3.0:

1. Many changes have been made in the FitsChan class in order to bring the FITS-WCS encoding into line with the current versions of the FITS-WCS papers (see http://www.atnf.csiro.au/people/mcal
    - The rotation and scaling of the pixel axes may now be specified using either CDi_j keywords, or PCi_j and CDELTj keywords. A new attribute called CDMatrix has been added to the FitsChan class to indicate which set of keywords should be used when writing a FrameSet to a FITS-WCS header.
    - The FITS-WCS encoding now supports most of the conventions described in FITS-WCS paper III for the description of spectral coordinates. The exceptions are that the SSYSOBS keyword is not supported, and WCS stored in tabular form (as indicated by the "-TAB" algorithm code) is not supported.
    - User-specified fiducial points for WCS projections are now supported by FitsChans which use FITS-WCS encoding. This use keywords PVi_0, PVi_1 and PVi_2 for the longitude axis.
    - When reading a FITS-WCS header, a FitsChan will now use keywords PVi_3 and PVi_4 for the longitude axis (if present) in preference to any LONPOLE and LATPOLE keywords which may be present. When writing a FITS-WCS header, both forms are written out.
    - The number of WCS axes is stored in the WCSAXES keyword if its value would be different to that of the NAXIS keyword.

- Helio-ecliptic coordinates are now supported by FitsChans which use FITS-WCS encoding. This uses CTYPE codes "HLON" and "HLAT". The resulting SkyFrame will have a System value of "HELIOECLIPTIC", and all the usual facilities, such as conversion to other celestial systems, are available.
- The FITS-WCS encoding now supports most of the conventions described in FITS-WCS paper III for the description of spectral coordinates. The exceptions are that the SSYSOBS keyword is not supported, and WCS stored in tabular form (as indicated by the "-TAB" algorithm code) is not supported.
- When reading a FITS-WCS header, a FitsChan will now ignore any distortion codes which are present in CTYPE keywords. Here, a "distortion code" is the final group of four characters in a CTYPE value of the form "xxxx-yyy-zzz", as described in FITS-WCS paper IV. The exception to this is that the "-SIP" distortion code (as used by the Spitzer Space Telescope project - see http://ssc.spitzer.caltech.edu/postbcd/doc/shupeADASS.pdf) is interpreted correctly and results in a PolyMap being used to represent the distortion in the resulting FrameSet. Note, "-SIP" distortion codes can only be read, not written. A FrameSet which uses a PolyMap will not in general be able to be written out to a FitsChan using any foreign encoding (although NATIVE encoding can of course be used).
- The Warnings attribute of the FitsChan class now accepts values "BadVal" (which gives warnings about conversion errors when reading FITS keyword values), "Distortion" (which gives warnings about unsupported distortion codes within CTYPE values), and "BadMat" (which gives a warning if the rotation/scaling matrix cannot be inverted).
- When writing a FrameSet to a FitsChan which uses a non-Native encoding, the comment associated with any card already in the FitsChan will be retained if the keyword value being written is the same as the keyword value already in the FitsChan.
- A FrameSet which uses the non-FITS projection type AST__TPN (a TAN projection with polynomial distortion terms) can now be written to a FitsChan if the Encoding attribute is set to FITS-WCS. The standard "-TAN" code is used within the CTYPE values, and the distortion coefficients are encoded in keywords of the form " QVi_ma", which are directly analogous to the standard "PVi_ma" projection parameter keywords. Thus a FITS reader which does not recognise the QV keywords will still be able to read the header, but the distortion will be ignored.
- The default value for DefB1950 attribute now depends on the value of the Encoding attribute.
- A new appendix has been added to SUN/210 and SUN/211 giving details of the implementation provided by the FitsChan class of the conventions contained in the first four FITS-WCS papers.

2. The SkyFrame class now supports two new coordinate systems "ICRS" and "HELIOECLIPTIC". The default for the System attribute for SkyFrames has been changed from "FK5" to "ICRS".

3. The astRate function has been added which allows an estimate to be made of the rate of change of a Mapping output with respect to one of the Mapping inputs.

4. All attribute names for Frames of any class may now include an optional axis specifier. This includes those attributes which describe a property of the whole Frame. For instance,

the Domain attribute may now be specified as "Domain(1)" in addition to the simpler "Domain". In cases such as this, where the attribute describes a property of the whole Frame, axis specifiers will usually be ignored. The exception is that a CmpFrame will use the presence of an axis specifier to indicate that the attribute name relates to the primary Frame containing the specified axis, rather than to the CmpFrame as a whole.

5. A new subclass of Mapping, the PolyMap, has been added which performs a general N-dimensional polynomial mapping.
6. A new subclass of Mapping, the GrismMap, has been added which models the spectral dispersion produced by a grating, prism or grism.
7. A new subclass of Mapping, the ShiftMap, has been added which adds constant values onto all coordinates (this is equivalent to a WinMap with unit scaling on all axes).
8. Minor bugs have been fixed within the Plot class to do with the choice and placement of numerical axis labels.
9. The SphMap class has a new attribute called PolarLong which gives the longitude value to be returned when a Cartesian position corresponding to either the north or south pole is transformed into spherical coordinates.
10. The WcsMap class now assigns a longitude of zero to output celestial coordinates which have a latitude of plus or minus 90 degrees.
11. The NatLat and NatLon attributes of the WcsMap class have been changed so that they now return the fixed native coordinates of the projection reference point, rather than the native coordinates of the user-defined fiducial point.
12. Notation has been changed in both the WcsMap and FitsChan classes to reflect the convention used in the FITS-WCS papers that index "i" refers to a world coordinate axis, and index "j" refers to a pixel axis.
13. Changes have been made to several Mapping classes in order to allow the astSimplify function to make simplifications in a CmpMap which previously were not possible.
14. The SlaMap class has been extended by the addition of conversions between FK5 and ICRS coordinates, and between FK5 and helio-ecliptic coordinates.
15. The SpecMap class has been changed to use the equation for the refractive index of air as given in the current version of FITS-WCS paper III. Also, the forward and inverse transformations between frequency and air-wavelength have been made more compatible by using an iterative procedure to calculate the inverse.

H.17 Changes Introduced in V3.1

The following describes the most significant changes which have occurred in the AST library between versions V3.0 and V3.1 (the current version):

1. Addition of a new class called XmlChan - a Channel which reads and writes AST objects in the form of XML.

2. A bug has been fixed in the Plot class which could cause incorrect graphical attributes to be used for various parts of the plot if either axis has no tick marks (i.e. if both major and minor tick marks have zero length).

Programs which are statically linked will need to be re-linked in order to take advantage of these new facilities.

H.18 Changes Introduced in V3.2

The following describes the most significant changes which have occurred in the AST library between versions V3.1 and V3.2:

1. A new function astPutCards has been added to the FitsChan class. This allows multiple concatenated header cards to be stored in a FitsChan in a single call, providing an alternative to the existing astPutCards function.
2. Some signficant changes have been made to the simplification of Mappings which should resultin a greater degree of simplication taking place.Some bugs have also been fixed which could result in an infinite loop being entered when attempting to simplify certain Mappings.
3. The FitsChan class now translates the spectral algorithm codes "-WAV", "-FRQ" and "-VEL" (specified in early drafts of paper III) to the corresponding "-X2P" form when reading a spectral axis description from a set of FITS header cards.
4. A bug has been fixed in the FitsChan class which could cause keywords associated with alternate axis descriptions to be mis-interpreted.
5. The Plot class now provides facilities for modifying the appearance of sub-strings within text strings such as axis labels, titles, *etc*, by producing super-scripts, sub-scripts, changing the font colour, size, *etc*. See attribute Escape.
6. The default value of the Tol attribute of the Plot class has been changed from 0.001 to 0.01. This should not usually cause any significant visible change to the plot, but should make the plotting faster. You may need to set a lower value for Tol if you are producing a particularly large plot.
7. The algorithm for finding the default value for the Gap attribute has been changed. This attribute specifies the gap between major axis values in an annotated grid drawn by the Plot class. The change in algorithm may cause the default value to be different to previous versions in cirtain circumstances.
8. Some bugs have been fixed in the Plot class which could cause the system to hang for a long time while drawing certain all-sky grids (notable some of the FITS Quad-cube projections).
9. The SkyAxis class has extended the Format attribute by the addition of the "g" option. this option is similar to the older "l" option in that it results in characters ("h", "m", "s", *etc*) being used as delimiters between the sexagesimal fields of the celestial position. The difference is that the "g" option includes graphics escape sequences in the returned formatted string which result in the field delimiter characters being drawn as super-scripts when plotted as numerical axis values by a Plot.

10. The Plot class has been extended to include facilities for producing logarithmic axes. See attributes LogPlot, LogTicks, LogGap and LogLabel.

11. New functions astGCap and astGScales have been added to the interface defined by file `grf.h`. The ast_link command has been modified so that the `-mygrf` switch loads dummy versions of the new grf functions. This means that applications should continue to build without any change. However, the facilities for interpreting escape sequences within strings drawn by the Plot class will not be available unless the new grf functions are implemented. If you choose to implement them, you should modify your linking procedure to use the `-grf` switch in place of the older `-mygrf` switch. See the description of the ast_link command for details of the new switches. Also note that the astGQch function, whilst included in verb+grf.h+ in pervious versions of AST, was not actually called. As of this version of AST, calls are made to the astGQch function, and so any bugs in the implementation of astGQch may cause spurious behaviour when plotting text strings.

12. A new 'static' method called astEscapes has been added which is used to control and enquire whether astGetC and astFormat will strip any graphical escape sequences which may be present out of the returned value.

13. New attribute XmlPrefix has been added to the XmlChan class. It allows XML written by the XmlChan class to include an explicit namespace prefix on each element.

14. New attribute XmlFormat has been added to the XmlChan class. It specifies the format in which AST objects should be written.

15. A new class of Mapping, the TranMap, has been introduced. A TranMap takes its forward transformation from an existing Mapping, and its inverse transformation from another existing Mapping.

16. A bug has been fixed in WcsMap which caused error reports to include erroneous axis numbers when referring to missing parameter values.

H.19 Changes Introduced in V3.3

The following describes the most significant changes which have occurred in the AST library between versions V3.2 and V3.3:

1. Options have been added to the SkyFrame class which allows the origin of celestial coordinates to be moved to any specified point. See the new attributes SkyRef, SkyRefIs, SkyRefP and AlignOffset.

2. An option has been added to the FitsChan class which allows extra Frames representing cartesian projection plane coordinates ("intermediate world coordinates" in the parlance of FITS-WCS) to be created when reading WCS information from a foreign FITS header. This option is controlled by a new attribute called Iwc.

3. The FitsChan class which been modified to interpret FITS-WCS CAR projection headers correctly if the longitude reference pixel (CRPIX) is very large.

4. The FITS-AIPS++ encoding in the FitsChan class now recognised spectral axes if they conform to the AIPS convention in which the spectral axis is descirbed by a CTYPE keyword od the form "AAAA-BBB" where "AAAA" is one of FREQ, VELO or FELO, and "BBB" is one of LSR, LSD, HEL or OBS. Such spectral axes can be both read and written.

5. The FitsChan class now has a FITS-AIPS++ encoding which represents WCS information using FITS header cards recognised by the AIPS++ project. Support for spectral axes is identical to the FITS-AIPS encoding.

6. The organisation of the AST distribution and the commands for building it have been changed. Whereas AST used to be built and installed with `./mk build; ./mk install`, it now builds using the more standard idiom `./configure; make; make install`. The installation location is controlled by the `--prefix` argument to ./configure (as is usual for other packages which use this scheme). Note that the INSTALL environment variable now has a *different* meaning to that which it had before, and it should generally be *unset*. Also, there is no need to set the SYSTEM variable.

7. Shared libraries are now installed in the same directory as the static libraries. In addition, links to sharable libraries are installed with names which include version information, and "libtool libraries" are also installed (see http://www.gnu.org/software/libtool/manual.html).

8. The `ast_dev` script has been removed. Instead, the location of the AST include files should be specified using the -I option when compiling.

H.20 Changes Introduced in V3.4

The following describes the most significant changes which have occurred in the AST library between versions V3.3 and V3.4:

1. The Mapping class has a new method (astLinearApprox) which calculates the co-efficients of a linear approximation to a Mapping.

2. The Format attribute for simple Frames and SkyFrames has been extended. It has always been possible, in both classes, to specify a precision by including a dot in the Format value followed by an integer (*e.g.* "`dms.1`" for a SkyFrame, or "`%.10g`" for a simple Frame). The precision can now also be specified using an asterisk in place of the integer (*e.g.* "`dms.*`" or "`%.*g`"). This causes the precision to be derived on the basis of the Digits attribute value.

3. The Plot class has been changed so that the default value used for the Digits attribute is chosen to be the smallest value which results in no pair of adjacent labels being identical. For instance, if an annotated grid is being drawn describing a SkyFrame, and the Format(1) value is set to "`hms.*g`" (the "g" causes field delimiters to be drawn as superscripts), and the Digits(1) value is unset, then the seconds field will have a number of decimal places which results in no pair of labels being identical.

4. Addition of a new class classed DSBSpecFrame. This is a sub-class of SpecFrame which can be used to describe spectral axes associated with dual sideband spectral data.

5. The FitsChan class will now read headers which use the old "-GLS" projection code, converting them to the corresponding modern "-SFL" code, provided that the celestial axes are not rotated.

6. The FitsChan class has a new Encoding, "FITS-CLASS", which allows the reading and writing of FITS headers using the conventions of the CLASS package - see http://www.iram.fr/IRAMFR/GIL html/class.html).

H.21 Changes Introduced in V3.5

The following describes the most significant changes which have occurred in the AST library between versions V3.4 and V3.5:

1. AST now provides facilities for representing regions of various shapes within a coordinate system. The Region class provides general facilities which are independent of the specific shape of region being used. Various sub-classes of Region are also now available which provide means of creating Regions of specific shape. Facilities provided by the Region class include testing points to see if they are inside the Region, testing two Regions for overlap, transforming Regions from one coordinate system to another *etc.*

2. A new class of 1-dimensional Frame called FluxFrame has been added which can be used to describe various systems for describing ovserved value at a single fixed spectral position.

3. A new class of 2-dimensional Frame called SpecFluxFrame has been added which can be used to describe a 2-d frame spanned by a spectral position axis and and an observed value axis.

4. A new class of Mapping called RateMap has been added. A RateMap encapsulates a previously created Mapping. The inputs of the RateMap correspond to the inputs of the encapsulated Mapping. All RateMaps have just a single output which correspond to the rate of change of a specified output of the encapsulated Mapping with respect to a specified input.

5. The SkyFrame class now supports a value of "J2000" for System. This system is an equatorial system based on the mean dynamical equator and equinox at J2000, and differs slightly from an FK5(J2000) system.

6. A new class called KeyMap has been added. A KeyMap can be used to store a collection of vector or scalar values or Objects, indexed by a character string rather than an integer.

7. The parameter list for the astRate method of the Mapping class has been modified. It no longer returns a second derivative estimate. Existing code which uses this method will need to be changed.

8. Methods (astSetFits¡X¿) have been added to the FitsChan class to allow values for named keywords to be changed or added.

H.22 Changes Introduced in V3.6

The following describes the most significant changes which occurred in the AST library between versions V3.5 and V3.6:

1. If the Format attribute associated with an axis of a SkyFrame starts with a percent character ("%"), then axis values are now formatted and unformatted as a decimal radians value, using the Format syntax of a simple Frame.
2. The Plot class has a new attribute called Clip which controls the clipping performed by AST at the plot boundary.
3. The keys used to label components of the PolyMap structure when a PolyMap is written out through a Channel have been changed. The new keys are shorter than the old keys and so can written succesfully to a FitsChan. The new PolyMap class always writes new styles keys but can read either old or new style keys. Consequently, PolyMap dumps written by this version of AST cannot be read by older versions of AST.
4. A mimimal cut down subset of the C version of SLALIB is now included with the AST distribution and built as part of building AST. This means that it is no longer necessary to have SLALIB installed separately at your site. The SLALIB code included with AST is distrubuted under the GPL. The default behaviour of the ast_link script is now to link with this internal slalib subset. However, the "-csla" option can still be used to force linking with an external full C SLALIB library. A new option "-fsla" has been introduced which forces linking with the external full Fortran SLALIB library.

H.23 Changes Introduced in V3.7

The following describes the most significant changes which occurred in the AST library between versions V3.6 and V3.7:

1. Support for time coordinate systems has been introduced throught the addition of two new classes, TimeFrame and TimeMap. The TimeFrame is a 1-dimensional Frame which can be used to describe moments in time (either absolute or relative) in various systems (MJD, Julian Epoch, *etc.*) and referred to various time scales (TAI, UTC, UT1, GMST, *etc*). The TimeMap is a Mapping which can transform time values between these various systems and time scales. Note, FitsChans which have a foreign encoding (*i.e.* any encoding other than NATIVE) are not able to read or write these new classes.

H.24 Changes Introduced in V4.0

The following describes the most significant changes which occurred in the AST library between versions V3.7 and V4.0:

1. Experimental support for reading IVOA Space-Time-Coordinates (STC-X) descriptions using the XmlChan class has been added. Support is included for a subset of V1.20 of the draft STC specification.

2. A new set of methods (AST_REBIN¡X¿/astRebin¡X¿) has been added to the Mapping class. These are flux-conserving alternatives to the existing AST_RESAMPLE¡X¿/astResample¡X¿ methods.

H.25 Changes Introduced in V4.1

The following describes the most significant changes which occurred in the AST library between versions V4.0 and V4.1:

1. A new control flag has been added to the AST_RESAMPLE¡X¿/astResample¡X¿ functions which produces approximate flux conservation.
2. New constants AST__SOMB and AST__SOMBCOS have been added to ast.h. These specify kernels for astResample and astRebin based on the "Sombrero" function ($2 * J1(x)/x$ where $J1(x)$ is the first order Bessel function of the first kind).
3. The SkyFrame class now supports a System value of AZEL corresponding to horizon (azimuth/elevation) coordinates.
4. The FitsChan class allows the non-standard strings "AZ–" and "EL–" to be used as axis types in FITS-WCS CTYPE keyword values.
5. The Frame class now has attributes ObsLon and ObsLat to specify the geodetic longitude and latitude of the observer.
6. The ClockLon and ClockLat attributes have been removed from the TimeFrame class. Likewise, the GeoLon and GeoLat attributes have been removed from the SpecFrame class. Both classes now use the ObsLon and ObsLat attributes of the parent Frame class instead. However, the old attribute names can be used as synonyms for ObsLat and ObsLon. Also, dumps created using the old scheme can be read succesfully by AST V4.1 and converted to the new form.
7. A new function astMapSplit has been added to the Mapping class. This splits a Mapping into two component Mappings which, when combined in parallel, are equivalent to the original Mapping.
8. The default value for the SkyRefIs attribute has been changed from "Origin" to "Ignored". This means that if you want to use a SkyFrame to represent offsets from some origin position, you must now set the SkyRefIs attribute explicitly to either "Pole" or "Origin", in addition to assigning the required origin position to the SkyRef attribute.

H.26 Changes Introduced in V4.2

The following describes the most significant changes which occurred in the AST library between versions V4.1 and V4.2:

1. The SideBand attribute of the DSBSpecFrame class can now take the option "LO" in addition to "USB" and "LSB". The new option causes the DSBSpecFrame to represent the offset from the local oscillator frequency, rather than either of the two sidebands.

2. The FitsChan class has been changed so that it writes out a VELOSYS keyword when creating a FITS-WCS encoding (VELOSYS indicates the topocentric apparent velocity of the standard of rest). FitsChan also strips out VELOSYS keywords when reading a FrameSet from a FITS-WCS encoding.
3. The FitsChan class has a new method called astRetainFits that indicates that the current card in the FitsChan should not be stripped out of the FitsChan when an AST Object is read from the FitsChan. Unless this method is used, all cards that were involved in the creation of the AST Object will be stripped from the FitsChan afte a read operation.
4. A problem with unaligned memory access that could cause bus errors on Solaris has been fixed.
5. A new read-only attribute called ObjSize has been added to the base Object Class. This gives the number of bytes of memory occupied by the Object. Note, this is the size of the internal in-memory representation of the Object, not the size of the textual representation produced by writing the Object out through a Channel.
6. A new function astTune has been added which can be used to get and set global AST tuning parameters. At the moment there are only two such parameter, both of which are concerned with memory management within AST.
7. A new method called astTranGrid has been added to the Mapping class. This method creates a regular grid of points covering a rectangular region within the input space of a Mapping, and then transforms this set of points into the output space of the Mapping, using a piecewise-continuous linear approximation to the Mapping if appropriate in order to achive higher speed.
8. A new subclass of Mapping has been added called SwitchMap. A SwitchMap represents several alternate Mappings, each of which is used to transforms input positions within a different region of the input coordinate space.
9. A new subclass of Mapping has been added called SelectorMap. A SelectorMap tests each input position to see if it falls within one of several Regions. If it does, the index of the Region containing the input position is returned as the Mapping output.
10. The behaviour of the astConvert method when trying to align a CmpFrame with another Frame has been modified. If no conversion between positions in the Frame and CmpFrame can be found, an attempt is now made to find a conversion between the Frame and one of two component Frames contained within the CmpFrame. Thus is should now be possible to align a SkyFrame with a CmpFrame containing a SkyFrame and a SpecFrame (for instance). The returned Mapping produces bad values for the extra axes (i.e. for the SpecFrame axis in the above example).
11. The "ast_link_adam" and "ast_link" scripts now ignore the `-fsla` and `-csla` options, and always link against the minimal cut-down version of SLALIB distributed as part of AST.

H.27 Changes Introduced in V4.3

The following describes the most significant changes which occurred in the AST library between versions V4.2 and V4.3:

1. The astGetFitsS function now strips trailing white space from the returned string, if the original string contains 8 or fewer characters
2. The SpecFrame class has a new attribute called SourceSys that specified whether the SourceVel attribute (which specifies the rest frame of the source) should be accessed as an apparent radial velocity or a redshift. Note, any existing software that assumes that SourceVel always represents a velocity in km/s should be changed to allow for the possibility of SourceVel representing a redshift value.

H.28 Changes Introduced in V4.4

The following describes the most significant changes which occurred in the AST library between versions V4.3 and V4.4:

1. The astFindFrame function can now be used to search a CmpFrame for an instance of a more specialised class of Frame (SkyFrame, TimeFrame, SpecFrame, DSBSpecFrame or FluxFrame). That is, if an instance of one of these classes is used as the "template" when calling astFindFrame, and the "target" being searched is a CmpFrame (or a FrameSet in which the current Frame is a CmpFrame), then the component Frames within the CmpFrame will be searched for an instance of the supplied template Frame, and, if found, a suitable Mapping (which will include a PermMap to select the required axes from the CmpFrame) will be returned by astFindFrame. Note, for this to work, the MaxAxes and MinAxes attributes of the template Frame must be set so that they cover a range that includes the number of axes in the target CmpFrame.
2. The SkyFrame, SpecFrame, DSBSpecFrame, TimeFrame and FluxFrame classes now allow the MaxAxes and MinAxes attributes to be set freely to any value. In previous versions of AST, any attempt to change the value of MinAxes or MaxAxes was ignored, resulting in them always taking the default values.
3. The DSBSpecFrame class has a new attribute called AlignSB that specifies whether or not to take account of the SideBand attributes when aligning two DSBSpecFrames using astConvert.
4. The Frame class has a new attribute called Dut1 that can be used to store a value for the difference between the UT1 and UTC timescales at the epoch referred to by the Frame.
5. The number of digits used to format the Frame attributes ObsLat and ObsLon has been increased.
6. The use of the SkyFrame attribute AlignOffset has been changed. This attribute is used to control how two SkyFrames are aligned by astConvert. If the template and target SkyFrames both have a non-zero value for AlignOffset, then alignment occurs between the offset coordinate systems (that is, a UnitMap will always be used to align the two SkyFrames).
7. The Plot class has a new attribute called ForceExterior that can be used to force exterior (rather than interior) tick marks to be produced. By default, exterior ticks are only produced if this would result in more than 3 tick marks being drawn.
8. The TimeFrame class now supports conversion between angle based timescales such as UT1 and atomic based timescales such as UTC.

H.29 Changes Introduced in V4.5

The following describes the most significant changes that occurred in the AST library between versions V4.4 and V4.5:

1. All FITS-CLASS headers are now created with a frequency axis. If the FrameSet supplied to astWrite contains a velocity axis (or any other form of spectral axis) it will be converted to an equivalent frequency axis before being used to create the FITS-CLASS header.
2. The value stored in the FITS-CLASS keyword "VELO-LSR" has been changed from the velocity of the source to the velocity of the reference channel.
3. Addition of a new method call astPurgeWCS to the FitsChan class. This method removes all WCS-related header cards from a FitsChan.
4. The Plot class has a new attribute called GrfContext that can be used to comminicate context information between an application and any graphics functions registered with the Plot class via the astGrfSet function.
5. Functions registered with the Plot class using astGrfSet now take a new additional integer parameter, "grfcon". The Plot class sets this parameter to the value of the Plot's GrfContext attribute before calling the graphics function. NOTE, THIS CHANGE WILL REQUIRE EXISTING CODE THAT USES astGrfSet TO BE MODIFIED TO INCLUDE THE NEW PARAMETER.
6. The astRebinSeq functions now have an extra parameter that is used to record the total number of input data values added into the output array. This is necessary to correct a flaw in the calculation of output variances based on the spread of input values. NOTE, THIS CHANGE WILL REQUIRE EXISTING CODE TO BE MODIFIED TO INCLUDE THE NEW PARAMETER (CALLED "NUSED").
7. Support has been added for the FITS-WCS "HPX" (HEALPix) projection.
8. A new flag "AST__VARWGT" can be supplied to astRebinSeq. This causes the input data values to be weighted using the reciprocals of the input variances (if supplied).
9. The Frame class has a new read-only attribute called NormUnit that returns the normalised value of the Unit attribute for an axis. Here, "normalisation" means cancelling redundant units, etc. So for instance, a Unit value of "s*(m/s)" would result in a NormUnit value of "m".
10. A new function astShowMesh has been added to the Region class. It displays a mesh of points covering the surface of a Region by writing out a table of axis values to standard output.
11. The Plot class now honours the value of the LabelUp attribute even if numerical labels are placed around the edge of the Plot. Previously LabelUp was only used if the labels were drawn within the interior of the plot. The LabelUp attribute controls whether numerical labels are drawn horizontally or parallel to the axis they describe.
12. A bug has been fixed that could segmentation violations when setting attribute values.

H.30 Changes Introduced in V4.6

The following describes the most significant changes which have occurred in the AST library between versions V4.5 and V4.6:

1. The TimeFrame class now support Local Time as a time scale. The offset from UTC to Local Time is specified by a new TimeFrame attribute called LTOffset.
2. A new class called Plot3D has been added. The Plot3D class allows the creation of 3-dimensional annotated coordinate grids.
3. A correction for diurnal aberration is now included when converting between AZEL and other celestial coordinate systems. The correction is based on the value of the ObsLat Frame attribute (the geodetic latitude of the observer).
4. A bug has been fixed which caused the DUT1 attribute to be ignored by the SkyFrame class when finding conversions between AZEL and other celestial coordinate systems.

H.31 Changes Introduced in V5.0

The following describes the most significant changes which occurred in the AST library between versions V4.6 and V5.0:

1. The AST library is now thread-safe (assuming that the POSIX pthreads library is available when AST is built). Many of the macros defined in the ast.h header file have changed. It is therefore necessary to re-compile all source code that includes ast.h.
2. New methods astLock and astUnlock allow an AST Object to be locked for exclusive use by a thread.
3. The TimeFrame class now support Local Time as a time scale. The offset from UTC to Local Time is specified by a new TimeFrame attribute called LTOffset.
4. The Channel class has a new attribute called Strict which controls whether or not to report an error if unexpected data items are found within an AST Object description read from an external data source. Note, the default behaviour is now not to report such errors. This differs from previous versions of AST which always reported an error is unexpected input items were encountered.

H.32 Changes Introduced in V5.1

The following describes the most significant changes which occurred in the AST library between versions V5.0 and V5.1:

1. The astUnlock function now has an extra parameter that controls whether or not an error is reported if the Object is currently locked by another thread.
2. The Prism class has been modified so that any class of Region can be used to define the extrusion axes. Previously, only a Box or Interval could be used for this purpose.

3. The values of the AST__THREADSAFE macro (defined in ast.h) have been changed from “yes” and “no” to “1” and “0”.
4. Improvements have been made to the way that Prisms are simplified when astSimplify is called. The changes mean that more types of Prism will now simplify into a simpler class of Region.
5. The PointList class has a new method, astPoints, that copies the axis values from the PointList into a supplied array.
6. The PointList class has a new (read-only) attribute, ListSize, that gives the number of points stored in the PointList.
7. The handling of warnings within different classes of Channel has been rationalised. The XmlStrict attribute and astXmlWarnings function have been removed. The same functionality is now available via the existing Strict attribute (which has had its remit widened), a new attribute called ReportLevel, and the new astWarnings function. This new function can be used on any class of Channel. Teh FitsChan class retains its long standing ability to store warnings as header cards within the FitsChan, but it also now stores warnings in the parent Channel structure, from where they can be retrieved using the astWarnings function.
8. A new function called astIntercept has been added to the Frame class. This function finds the point of intersection beteeen two geodesic curves.
9. A bug in the type-checking of Objects passed as arguments to constructor functions has been fixed. This bug could lead to applications crashing or showing strange behaviour if an inappropriate class of Object was supplied as an argument to a constructor.
10. The astPickAxes function will now return a Region, if possible, when applied to a Region. If this is not possible, a Frame will be returned as before.
11. The default gap size between the ISO date/time labels used by the Plot class when displaying an annotated axis described by a TimeFrame has been changed. The changes are meant to improve the labelling of calendar time axes that span intervals from a day to a few years.
12. A new function called astTestFits has been added to the FitsChan class. This function tests a FitsChan to see if it contains a defined value for specified FITS keyword.
13. The AST__UNDEF¡X¿ parameters used to flag undefined FITS keyword values have been removed. Use the new astTestFits function instead.
14. The astIsUndef¡X¿ functions used to test FITS keyword values have been removed. Use the new astTestFits function instead.

H.33 Changes Introduced in V5.2

The following describes the most significant changes which occurred in the AST library between versions V5.1 and V5.2:

1. A new method called astSetFitsCM has been added to the FitsChan class. It stores a pure comment card in a FitsChan (that is, a card with no keyword name or equals sign).

2. A new attribute called ObsAlt has been added to the Frame class. It records the geodetic altitude of the observer, in metres. It defaults to zero. It is used when converting times to or from the TDB timescale, or converting spectral positions to or from the topocentric rest frame, or converting sky positions to or from horizon coordinates. The FitsChan class will include its effect when creating a set of values for the OBSGEO-X/Y/Z keywords, and will also assign a value to it when reading a set of OBSGEO-X/Y/Z keyword values from a FITS header.

3. The TimeMap conversions "TTTOTDB" and "TDBTOTT", and the SpecMap conversions "TPF2HL" and "HLF2TP", now have an additional argument - the observer's geodetic altitude.

4. The Polygon class has been modified to make it consistent with the IVOA STC definition of a Polygon. Specifically, the inside of a polygon is now the area to the left of each edge as the vertices are traversed in an anti-clockwise manner, as seen from the inside of the celestial sphere. Previously, AST used the anti-clockwise convention, but viewed from the outside of the celestial sphere instead of the inside. Any Polygon saved using previous versions of AST will be identified and negated automatically when read by AST V5.2.

5. A new class of Channel, called StcsChan, has been added that allows conversion of suitable AST Objects to and from IVOA STC-S format.

6. A new method called astRemoveRegions has been added to the Mapping class. It searches a (possibly compound) Mapping (or Frame) for any instances of the AST Region class, and either removes them, or replaces them with UnitMaps (or equivalent Frames). It can be used to remove the masking effects of Regions from a compound Mapping or Frame.

7. A new method called astDownsize has been added to the Polygon class. It produces a new Polygon that contains a subset of the vertices in the supplied Polygon. The subset is chosen to retain the main features of the supplied Polygion, in so far as that is possible, within specified constraints.

8. A new constructor called astOutline has been added to the Polygon class. Given a 2D data array, it identifies the boundary of a region within the array that holds pixels with specified values. It then creates a new Polygon to describe this boundary to a specified accuracy.

9. A new set of methods, called astMapGetElem¡X¿ has been added to the KeyMap class. They allow a single element of a vector valued entry to be returned.

10. A new attribute called KeyError has been added to the KeyMap Class. It controls whether the astMapGet... family of functions report an error if an entry with the requested key does not exist in the KeyMap.

H.34 Changes Introduced in V5.3

The following describes the most significant changes which occurred in the AST library between versions V5.2 and V5.3:

1. The details of how a Frame is aligned with another Frame by the astFindFrame and astConvert functions have been changed. The changes mean that a Frame can now be aligned with an instance of a sub-class of Frame, so long as the number of axes and the Domain values are consistent. For instance, a basic 2-dimensional Frame with Domain "SKY" will now align succesfully with a SkyFrame, conversion between the two Frames being achieved using a UnitMap.

2. The arrays that supply input values for astMapPut1¡X¿ are now declared "const".

3. Added method astMatchAxes to the Frame class. This method allows corresponding axes within two Frames to be identified.

4. The astAddFrame method can now be used to append one or more axes to all Frames in a FrameSet.

H.35 Changes Introduced in V5.3-1

The following describes the most significant changes which have occurred in the AST library between versions V5.3 and V5.3-1:

1. The utility functions provided by the AST memory management layer are now documented in an appendix.

2. The KeyMap class now supports entries that have undefined values. A new method called astMapPutU will store an entry with undefined value in a keymap. Methods that retrieve values from a KeyMap (astMapGet0¡X¿, etc.) ignore entries with undefined values when searching for an entry with a given key.

3. The KeyMap class has a new method called astMapCopy that copies entries from one KeyMap to another KeyMap.

4. The KeyMap class has a new boolean attribute called MapLocked. If non-zero, an error is reported if an attempt is made to add any new entries to a KeyMap (the value associated with any old entry may still be changed without error). The default is zero.

5. The Object class has a new method called astHasAttribute/AST_HASATTRIBUTE that returns a boolean value indicating if a specified Object has a named attribute.

6. The SkyFrame class has two new read-only boolean attributes called IsLatAxis and IsLonAxis that can be used to determine the nature of a specified SkyFrame axis.

7. A bug has been fixed in the astRebin(Seq) methods that could cause flux to be lost from the edges of the supplied array.

8. A bug has been fixed in the astRebin(Seq) methods that caused the first user supplied parameter to be interpreted as the full width of the spreading kernel, rather than the half-width.

9. The StcsChan class now ignores case when reading STC-S phrases (except that units strings are still case sensitive).

10. A new Mapping method, astQuadApprox, produces a quadratic least-squares fit to a 2D Mapping.

11. A new Mapping method, astSkyOffsetMap, produces a Mapping from absolute SkyFrame coordinates to offset SkyFrame coordinates.

12. The Channel class now has an Indent attribute that controls indentation in the text created by astWrite. The StcsIndent and XmlIndent attributes have been removed.

13. All classes of Channel now use the string “¡bad¿” to represent the floating point value AST__BAD, rather than the literal formatted value (typically “-1.79769313486232e+308”).

14. The KeyMap class now uses the string “¡bad¿” to represent the floating point value AST__BAD, rather than the literal formatted value (typically “-1.79769313486232e+308”).

15. The KeyMap class has a new method called astMapPutElem¡X¿ that allows a value to be put into a single element of a vector entry in a KeyMap. The vector entry is extended automatically to hold the new element if required.

16. The DSBSpecFrame class now reports an error if the local oscillator frequency is less than the absoliute value of the intermediate frequency.

H.36 Changes Introduced in V5.3-2

The following describes the most significant changes which occurred in the AST library between versions V5.3-1 and V5.3-2:

1. A bug has been fixed in the FitsChan class that could cause wavelength axes to be assigned the units “m/s” when reading WCS information from a FITS header.

2. The astSet function now allows literal commas to be included in string attribute values. String attribute values that include a literal comma should be enclosed in quotation marks.

3. A bug in FitsChan has been fixed that caused “-SIN” projection codes within FITS-WCS headers to be mis-interpreted, resulting in no FrameSet being read by astRead.

4. The KeyMap class has a new attribute called “SortBy”. It controls the order in which keys are returned by the astMapKey function. Keys can be sorted alphabetically or by age, or left unsorted.

5. Access to KeyMaps holding thousands of entries is now significantly faster.

6. KeyMaps can now hold word (i.e. short integer) values.

H.37 Changes Introduced in V5.4-0

The following describes the most significant changes which occurred in the AST library between versions V5.3-2 and V5.4-0:

1. the FitsChan class now has an option to support reading and writing of FITS-WCS headers that use the -TAB algorithm described in FITS-WCS paper III. This option is controlled by a new FitsChan attribute called TabOK. See the documentation for TabOK for more information.
2. A new class called "Table" has been added. A Table is a KeyMap in which each entry represents a cell in a two-dimensional table.
3. A new class called "FitsTable" has been added. A FitsTable is a Table that has an associated FitsChan holding headers appropriate to a FITS binary table.
4. KeyMaps can now hold byte values. These are held in variables of type "unsigned char".
5. KeyMaps have a new attribute called KeyCase that can be set to zero to make the handling of keys case insensitive.
6. a memory leak associated with the use of the astMapPutElem¡X¿ functions has been fixed.
7. A new method called astMapRename has been added to rename existing entry in a KeyMap.

H.38 Changes Introduced in V5.5-0

The following describes the most significant changes which occurred in the AST library between versions V5.4-0 and V5.5-0:

1. The FitsChan "TabOK" attribute is now an integer value rather than a boolean value. If TabOK is set to a non-zero positive integer before invoking the astWrite method, its value is used as the version number for any table that is created as a consequence of the write operation. This is the value stored in the PVi_1a keyword in the IMAGE header, and the EXTVER keyword in the binary table header. In previous versions of AST, the value used for these headers could not be controlled and was fixed at 1. If TabOK is set to a negative or zero value, the -TAB algorithm will not be supported by either the astWrite or astRead methods.

H.39 Changes Introduced in V5.6-0

The following describes the most significant changes which occurred in the AST library between versions V5.5-0 and V5.6-0:

1. New functions astBBuf and astEBuf have been added to the Plot class. These control the buffering of graphical output produced by other Plot methods.

2. New functions astGBBuf and astGEBuf have been added to the interface defined by file `grf.h`. The ast_link command has been modified so that the `-grf_v3.2` switch loads dummy versions of the new grf functions. This means that applications that use the `-grf_v3.2` switch should continue to build without any change. However, the new public functions astBBuf and astEBuf will report an error unless the new grf functions are implemented. If you choose to implement them, you should modify your linking procedure to use the `-grf` (or `-grf_v5.6`) switch in place of the older `-grf_v3.2` switch. See the description of the ast_link command for details of these switches.

3. New method astGetRegionMesh returns a set of positions covering the boundary, or volume, of a supplied Region.

H.40 ChangesIntroduced in V5.6-1

The following describes the most significant changes which occurred in the AST library between versions V5.6-0 and V5.6-1:

1. Tables can now have any number of parameters describing the global properties of the Table.

2. Frames now interpret the unit string "A" as meaning "Ampere" rather than "Angstrom", as specified by FITS-WCS paper I.

3. A bug has been fixed in the astFindFrame method that allowed a template Frame of a more specialised class to match a target frame of a less specialised class. For example, this bug would allow a template SkyFrame to match a target Frame. This no longer happens.

H.41 Changes Introduced in V5.7-0

The following describes the most significant changes which occurred in the AST library between versions V5.6-1 and V5.7-0:

1. The FitsChan class support for the IRAF-specific "TNX" projection has been extended to include reading TNX headers that use a Chebyshev representation for the distortion polynomial.

2. The FitsChan class support for the IRAF-specific "ZPX" projection has been extended to include reading ZPX headers that use simple or Chebyshev representation for the distortion polynomial.

3. A bug has been fixed in the FitsChan class that caused headers including the Spitzer "-SIP" distortion code to be read incorrectly if no inverse polynomial was specified in the header.

4. A new attribute called PolyTan has been added to the FitsChan class. It can be used to indicate that FITS headers that specify a TAN projection should be interpreted according to the "distorted TAN" convention included in an early draft of FITS-WCS paper II. Such headers are created by (for instance) the SCAMP tool (http://www.astromatic.net/software/scamp).

5. The PolyMap class now provides a method called astPolyTran that adds an inverse transformation to a PolyMap by sampling the forward transformation on a regular grid, and then fitting a polynomial function from the resulting output values to the grid of input values.

H.42 Changes Introduced in V5.7-1

The following describes the most significant changes which occurred in the AST library between versions V5.7-0 and V5.7-1:

1. - All classes of Channel can now read to and write from specified text files, without the need to provide source and sink functions when the Channel is created. The files to use are specified by the new attributes SourceFile and SinkFile.
2. - The FitsChan class now ignores trailing spaces in character-valued WCS keywords when reading a FrameSet from a FITS header.
3. - If the FitsChan astRead method reads a FITS header that uses the -SIP (Spitzer) distortion code within the CTYPE values, but which does not provide an inverse polynomial correction, the FitsChan class will now use the PolyTran method of the PolyMap class to create an estimate of the inverse polynomial correction.

H.43 Changes Introduced in V5.7-2

The following describes the most significant changes which occurred in the AST library between versions V5.7-1 and V5.7-2:

1. The Object class has a new function astToString (C only), which creates an in-memory textual serialisation of a given AST Object. A corresponding new function called astFromString re-creates the Object from its serialisation.
2. The PolyMap class can now use an iterative Newton-Raphson method to evaluate the inverse the inverse transformation if no inverse transformation is defined when the PolyMap is created.
3. The FitsChan class has a new method astWriteFits which writes out all cards currently in the FitsChan to the associated external data sink (specified either by the SinkFile attribute or the sink function supplied when the FitsChan was created), and then empties the FitsChan.
4. The FitsChan class has a new read-only attribute called “Nkey”, which holds the number of keywords for which values are held in a FitsChan.
5. The FitsChan astGetFits¡X¿ methods can now be used to returned the value of the current card.
6. The FitsChan class has a new read-only attribute called “CardType”, which holds the data type of the keyword value for the current card.

7. The FitsChan class has a new method astReadFits which forces the FitsChan to reads cards from the associated external source and appends them to the end of the FitsChan.
8. - If the FitsChan astRead method reads a FITS header that uses the -SIP (Spitzer) distortion code within the CTYPE values, but which does not provide an inverse polynomial correction, and for which the PolyTran method of the PolyMap class fails to create an accurate estimate of the inverse polynomial correction, then an iterative method will be used to evaluate the inverse correction for each point transformed.

H.44 Changes Introduced in V6.0

The following describes the most significant changes which occurred in the AST library between versions V5.7-2 and V6.0:

1. This version of AST is the first that can be used with the Python AST wrapper module, starlink.Ast, available at http://github.com/timj/starlink-pyast.
2. When reading a FITS-WCS header, the FitsChan class now recognises the non-standard "TPV" projection code within a CTYPE keyword value. This code is used by SCAMP (see www.astromatic.net/software/scamp) to represent a distorted TAN projection.
3. The Plot class has been changed to remove visual anomalies (such as incorrectly rotated numerical axis labels) if the graphics coordinates have unequal scales on the X and Y axes.

   - The graphics escape sequences used to produce graphical sky axis labels can now be changed using the new function astTuneC.

H.45 Changes Introduced in V6.0-1

The following describes the most significant changes which occurred in the AST library between versions V6.0 and V6.0-1:

1. The FitsChan class now recognises the Spitzer "-SIP" distortion code within FITS headers that describe non-celestial axes, as well as celestial axes.
2. A bug has been fixed that could cause inappropriate equinox values to be used when aligning SkyFrames if the AlignSystem attribute is set.
3. The versioning string for AST has changed from "$< major > . < minor > - < release >$" to "$< major > . < minor > . < release >$".

H.46 Changes Introduced in V7.0.0

The following describes the most significant changes which occurred in the AST library between versions V6.0-1 and V7.0.0:

1. Fundamental positional astronomy calculations are now performed using the IAU SOFA library where possible, and the Starlink PAL library otherwise (the PAL library contains a subset of the Fortran Starlink SLALIB library re-written in C). Copies of these libraries are bundled with AST and so do not need to be obtained or built separately, although external copies of SOFA and PAL can be used if necessary by including the "`--with-external_pal`" option when configuring AST.

H.47 Changes Introduced in V7.0.1

The following describes the most significant changes which occurred in the AST library between versions V7.0.0 and V7.0.1:

1. The levmar and wcslib code distributed within AST is now stored in the main AST library (libast.so) rather than in separate libraries.

H.48 Changes Introduced in V7.0.2

The following describes the most significant changes which occurred in the AST library between versions V7.0.1 and V7.0.2:

1. The libast_pal library is no longer built if the "–with-external_pal" option is used when AST is configured.

H.49 Changes Introduced in V7.0.3

The following describes the most significant changes which occurred in the AST library between versions V7.0.2 and V7.0.3:

1. A bug has been fixed which could cause an incorrect axis to be used when accessing axis attributes within CmpFrames. This could happen if axes within the CmpFrame have been permuted.
2. A bug has been fixed in the SkyFrame class that could cause the two values of the SkyRef and/or SkyRefP attributes to be reversed.
3. Bugs have been fixed in the CmpRegion class that should allow the border around a compound Region to be plotted more quickly, and more accurately. Previously, component Regions nested deeply inside a CmpRegion may have been completely or partially ignored.
4. A bug has been fixed in the Plot3D class that caused a segmentation violation if the MinTick attribute was set to zero.
5. The astResampleX set of methods now includes astResampleK and astResampleUK that handles 64 bit integer data.

H.50 Changes Introduced in V7.0.4

The following describes the most significant changes which occurred in the AST library between versions V7.0.3 and V7.0.4:

1. The previously private grf3d.h header file is now installed into prefix/include.

H.51 Changes Introduced in V7.0.5

The following describes the most significant changes which occurred in the AST library between versions V7.0.4 and V7.0.5:

1. The FitsChan class can now read FITS headers that use the SAO convention for representing distorted TAN projections, based on the use of “COi_m” keywords to hold the coefficients of the distortion polynomial.

H.52 Changes Introduced in V7.0.6

The following describes the most significant changes which occurred in the AST library between versions V7.0.5 and V7.0.6:

1. A bug has been fixed in astRebinSeq¡X¿ which could result in incorrect normalisation of the final binned data and variance values.
2. When reading a FrameSet from a FITS-DSS header, the keywords CNPIX1 and CNPIX2 now default to zero if absent. Previously an error was reported.

H.53 Changes Introduced in V7.1.0

The following describes the most significant changes which occurred in the AST library between versions V7.0.6 and V7.1.0:

1. IMPORTANT! The default behaviour of astRebinSeq is now NOT to conserve flux. To conserve flux, the AST__CONSERVEFLUX flag should be supplied when calling astRebinSeq¡X¿. Without this flag, each output value is a weighted mean of the neighbouring input values.
2. A new flag AST__NONORM can be used with astRebinSeq¡X¿ to indicate that normalisation of the output arrays is not required. In this case no weights array need be supplied.
3. A bug has been fixed in astAddFrame method that could result in the incorrect inversion of Mappings within the FrameSet when the AST__ALLFRAMES flag is supplied for the ”iframe” parameter.
4. The astRate method has been re-written to make it faster and more reliable.

H.54 Changes Introduced in V7.1.1

The following describes the most significant changes which occurred in the AST library between versions V7.1.0 and V7.1.1:

1. When a FitsChan is used to write an "offset" SkyFrame (see attribute SkyRefIs) to a FITS-WCS encoded header, two alternate axis descriptions are now created - one for the offset coordinates and one for the absolute coordinates. If such a header is subsequently read back into AST, the original offset SkyFrame is recreated.
2. A bug has been fixed in FitsChan that caused inappropriate CTYPE values to be generated when writing a FrameSet to FITS-WCS headers if the current Frame describes generalised spherical coordinates (i.e. a SkyFrame with System=Unknown).

H.55 Changes Introduced in V7.2.0

The following describes the most significant changes which occurred in the AST library between versions V7.1.1 and V7.2.0:

1. A new method call astMapDefined has been added to the KeyMap class. It checks if a gtiven key name has a defined value in a given KeyMap.

H.56 Changes Introduced in V7.3.0

The following describes the most significant changes which occurred in the AST library between versions V7.2.0 and V7.3.0:

1. The interface for the astRebinSeq¡X¿ family of functions has been changed in order to allow a greater number of pixels to be pasted into the output array. The "nused" parameter is now a pointer to a "int64_t" variable, instead of an "int". APPLICATION CODE SHOULD BE CHANGED ACCORDINGLY TO AVOID SEGMENTATION FAULTS AND OTHER ERRATIC BEHAVIOUR.
2. Added a new facility to the FrameSet class to allow each Frame to be associated with multiple Mappings, any one of which can be used to connect the Frame to the other Frames in the FrameSet. The choice of which Mapping to use is controlled by the new "Variant" attribute of the FrameSet class.
3. Mappings (but not Frames) that have a value set for their Ident attribute are now left unchanged by the c astSimplify function. f AST_SIMPLIFY routine.

H.57 Changes Introduced in V7.3.1

The following describes the most significant changes which occurred in the AST library between versions V7.3.0 and V7.3.1:

1. Fix a bug that could cauise a segmentation violation when reading certain FITS headers that use a TNX projection.

H.58 Changes Introduced in V7.3.2

The following describes the most significant changes which have occurred in the AST library between versions V7.3.1 and V7.3.2 (the current version):

1. Fix support for reading FITS header that use a GLS projection. Previously, an incorrect transformation was used for such projections if any CRVAL or CROTA value was non-zero.
2. The KeyMap class has new sorting options "KeyAgeUp" and "KeyAgeDown" that retain the position of an existing entry if its value is changed. See the SortBy attribute.
3. A bug has been fixed in the FitsChan class that caused CDELT keywords for sky axes to be treated as radians rather than degrees when reading a FITS header, if the corresponding CTYPE values included no projection code.

Programs which are statically linked will need to be re-linked in order to take advantage of these new facilities.