E U R O P E A N   S O U T H E R N   O B S E R V A T O R Y

Organisation Européenne pour des Recherches Astronomiques dans l'Hémisphère Austral

Europäische Organisation für astronomische Forschung in der südlichen Hemisphäre

VERY LARGE TELESCOPE

Data Management Division

Astronomical Tcl and C++ Utilities Package

Programmer's Manual

Doc.No. VLT-MAN-ESO-19400-1551

Issue 1.0, Version 1.0

Date 19/01/98

	Name	Date	Signature
Prepared	A. Brighton	19/01/98	
Approved	M. Albrecht		
Released	P. Quinn		

Change Record

Issue/Rev.	Date	Section/Page affected	Reason/Initiation/Document/Remarks
1.0	11/01/98	All	Created

1 INTRODUCTION

This manual describes the *astrotcl* package, a collection of C, C++ and Tcl utilities for astronomical software development, wrapped up in a dynamically loadable Tcl package. This version contains support for world coordinates, FITS image I/O and compression and is intended to be reusable by other C++ and Tcl based applications.

1.1 Purpose

The purpose of this manual is to describe the *Astronomical Tcl and C++ utilities package* and its implementation.

1.2 Scope

This document is primarily aimed at software developers using the package. This package is dependent on the *tclutil* package and is currently used by the *rtd*, *cat*, and *skycat* packages.

1.3 Applicable Documents

This document is based on the following documents:

[1] VLT-PRO-ESO-10000-0228, 1.0 10/03/93 -- VLT Software Programming Standards

1.4 Reference Documents

The following documents are referenced in this document:

[1] VLT-MAN-ESO-19400-1550 1.0 19/01/98 -- Tcl and C++ Utilities, Programmer's Manual

2 OVERVIEW

The *astrotcl* package is designed for use in a Tcl/C/C++ environment. It is made up of the following parts, which are described in more detail in the following sections:

- *wcslib* - world coordinates support
- *imageio* - support for reading and writing images
- *compress* - image compression support

2.1 World Coordinates Support

The world coordinates support based in part on Doug Mink's *wcslib*, a C library that is also used by *saoimage* (see `ftp://cfa-ftp.harvard.edu/pub/gsc/WCSTools/`). The C library is not accessed directly by applications, but through a C++ wrapper class called *WCS*.

A second class, called *WorldCoords*, is not based on this C library, but is used to represent and do operations on world coordinates. The *WCS* class takes a FITS header and is then capable of converting between image and world coordinates, while the *WorldCoords* class just represents a world coordinate value (α, δ). Another class *WorldOrImageCoords*, represents, as the name implies, either world or image coordinates, depending on the value of a flag. The table below gives an overview of the classes dealing with world coordinates.

Class	Description
WorldCoords	Represents a world coordinate point (α, δ, *equinox*) .
ImageCoords	Represents an image coordinate point (x, y).
WorldOrImageCoords	Represents either world or image coordinates, depending on the value of a flag. Used as a place holder when it is not clear ahead of time whether or not world coordinate information will be available.
HMS	Represents a value in hours (or degrees), minutes, and seconds.
WCS	Based on Doug Mink's *wcslib* C library and used to convert between world and image coordinates, given a FITS image header and/or the necessary information.

In addition to the C++ classes listed above, a simple Tcl command, *wcs*, is provided to convert between *hh:mm:ss +dd:mm:ss* format and double *degrees* in Tcl applications. A lot more could be done in this area, if it is needed. Currently the *rtd* and *skycat* applications, which are based on this package, provide many of the features found in the WCS class as part of a Tcl interface.

2.2 Image I/O

Although the current software version only supports FITS image I/O, the classes are organized in such a way as to allow other image types to be supported in the future. The main class here, *ImageIO*, uses *reference counting* to manage a pointer to an internal class, which must be a subclass of *ImageIORep*. The use of reference counting helps to avoid unnecessary copies and simplify memory management. Currently, the only subclass implemented is *FitsIO*, which is used to read, write, and store FITS images.

Internally, the *FitsIO* class uses the *wcslib* to scan FITS headers, and also supports automatic compression and decompression of images using the *Compress* class described below. For efficiency, *mmap* is used whenever possible, to avoid long delays reading and writing large FITS files, by mapping the disk files directly to virtual memory. This saves time in image display applications, since the actual image data is only accessed when it is needed. The table below summarizes the classes for ImageIO:

Class	Description
ImageIO	Represents a reference counted image of unknown type.
ImageIORep	Abstract base class for image types.
FitsIO	Class derived from ImageIORep, implements FITS image I/O.

2.3 Image Compression and Decompression

The *Compress* class defined here is based on the CACD *press* C library routines. A number of different FITS image compression types are supported, including *H-Compress*, *gzip*, *UNIX compress* and *ULDA* compression. Methods are available to compress and decompress images to and from memory or files. To save time in reading large FITS files, *mmap* may also be used to map the input files.

3 USER'S GUIDE

This section describes in general the contents of the *astrotcl* package. For details, see the man pages in the Reference section.

3.1 Representing World and Image Pixel Coordinates

Astronomical applications often deal with world or image pixel coordinates. This section describes some C++ classes that are used to represent coordinates. These are fairly simple classes that deal with abstract coordinates, such as (α, δ, *equinox*) for world coordinates or (x, y) for image coordinates. Another class, *WCS*, which is described in the next section, deals with coordinate conversion and FITS header keywords.

3.1.1 World Coordinates

World coordinates are managed by the `WorldCoords` class. Most of the work is done by the various constructors. You can pass in the coordinates as hh:mm:ss[+-]dd:mm:ss, in double degrees or in string format:

```
 // constructor: initialize NULL coordinates
 WorldCoords();

// constructor: ra is H:M:S and dec is D:M:S
WorldCoords(const HMS& ra, const HMS& dec, double equinox = 2000.0);

// constructor: ra and dec are both in degrees
WorldCoords(double ra, double dec, double equinox = 2000.0);

// constructor: r... is H:M:S, d... is D:M:S
WorldCoords(int rh, int rm, double rs, int dd, int dm, float ds,
        double equinox = 2000.0);

// constructor - parse ra and dec from strings
WorldCoords(const char* ra, const char* dec, double equinox = 2000.0);
```

The equinox for the coordinates can be specified as a floating point value, otherwise it defaults to 2000.0. Internally, the coordinates are always converted to J2000, so that there is no confusion when comparing coordinates from different sources. Methods are defined to get the coordinate values in degrees or H:M:S and in a given equinox. Other methods are available to compare coordinates and read and write them.

Example:

```
WorldCoords c1(49.95096, 41.51173);
WorldCoords c2(3, 19, 48.2304, 41, 30, 42.228);
WorldCoords c3(HMS(3, 19, 48.2304), HMS(41, 30, 42.228));
WorldCoords c4(HMS(c1.ra()), HMS(c1.dec()));
WorldCoords c5("3 19 48.2304", "+41 30 42.228", 2000.0);
WorldCoords c6("3:19:48.2304", "+41:30:42.228", 2000.0);
cout << "these coords should all be the same (or very close):" << endl
      << c1 << endl
      << c2 << endl
      << c3 << endl
      << c4 << endl
      << c5 << endl
      << c6 << endl;
```

3.1.2 Image Pixel Coordinates

Image pixel coordinates are represented by class `ImageCoords`, which has an interface similar to `WorldCoords`:

```
// constructor - initialize null coordinates
ImageCoords();

// constructor: initialize with (x, y)
ImageCoords(double x, double y);

// constructor - parse X and Y in string format
ImageCoords(const char* x_str, const char* y_str);
```

Example:

```
ImageCoords c1(123.456, 654.321);
ImageCoords c2("123.456.", "654.321.");

cout << "these coords should be the same (or very close):" << endl
      << c1 << endl
      << c2 << endl;
```

3.1.3 World or Image Coordinates

When we need to handle both world and image coordinates in the same way we use a special class `WorldOrImageCoords`. This class represents either image pixel coordinates (class `ImageCoords`) or World Coordinates (class `WorldCoords`). The constructor takes an instance of either of these classes and sets a flag, `isWcs`, to note which kind it is:

```
// constructor - initialize null coordinates
WorldOrImageCoords();

// constructor: initialize World Coordinates
WorldOrImageCoords(WorldCoords);

// constructor: initialize image pixel coords
WorldOrImageCoords(ImageCoords);
```

Example:

```
WorldOrImageCoords ic1(ImageCoords(123.456, 654.321));
WorldOrImageCoords ic2(ImageCoords("123.456.", "654.321."));
cout << "these coords should be the same (or very close):" << endl
      << ic1 << endl
      << ic2 << endl;

WorldOrImageCoords c1(WorldCoords(49.95096, 41.51173));
WorldOrImageCoords c2(WorldCoords(3, 19, 48.2304, 41, 30, 42.228));
WorldOrImageCoords c3(WorldCoords(HMS(3, 19, 48.2304), HMS(41, 30, 42.228)));
WorldOrImageCoords c4(WorldCoords(HMS(c1.ra()), HMS(c1.dec())));
WorldOrImageCoords c5(WorldCoords("3 19 48.2304", "+41 30 42.228", 2000.0));
WorldOrImageCoords c6(WorldCoords("3:19:48.2304", "+41:30:42.228", 2000.0));

cout << "these coords should all be the same (or very close):" << endl
      << c1 << endl
      << c2 << endl
      << c3 << endl
      << c4 << endl
      << c5 << endl
      << c6 << endl;
```

3.1.4 World Coordinates in C

We also support conversion between *hh:mm:ss [+-]dd:mm:ss* format and floating point degrees for plain, ANSI C based applications.

The following structs, defined in `worldCoords.h`, represent the world coordinate values:

```
typedef struct {
    int hours;
    int min;
    double sec;
    double val;/* value calculated in degrees */
} WC_HMS;


typedef struct {
    WC_HMS ra, dec;
} WC;
```

A number of routines are defined to operate on the WC struct, so that you can specify the coordinates of an object as doubles in degrees or in *hh:mm:ss [+-]dd:mm:ss* format:

World Coordinate C Utility Routines

Routine	Description
wcInitNull	initialize null world coordinates
wcIsNull	return true if the given coords are null
wcInitFromStrings	initialize world coords from char strings containing RA and DEC
wcInitFromHMS	initialize from RA, DEC in H:M:S D:M:S format and equinox
wcInitFromDeg	initialize from RA, DEC in degrees and equinox
wcPrint	print RA and DEC to the given buffers in the given equinox

3.2 Converting between World and Image Pixel Coordinates

Conversion between world and image coordinates is handled by the *WCS* class, which is a C++ wrapper class for Doug Mink's *wcslib* (*saoimage*) C library. This class is initialized with a FITS image header and can then be used to convert coordinates and distances on the image.

Example:

```
WCS wcs(fitsHeaderString);
double x, y, ra, dec;
...
// convert x,y to world coords
if (wcs.pix2wcs(x, y, ra, dec) != 0) {
      return ERROR;
}

// convert ra,dec to image coords
if (wcs.wcs2pix(ra, dec, x, y) != 0) {
      return ERROR;
}
```

The WCS class uses reference counting in order to make it easy to share an instance of the class among other classes. The first instance creates an internal C struct using the *wcslib* C library and the last instance to be deleted deletes the struct again.

```
WCS wcs2 = wcs; // both are reference counted copies of the same object !
```

The WCS class also has support for setting up world coordinates without a FITS header, with the wcs method:

```
wcs.set(ra, dec, secpix, xrefpix, yrefpix, nxpix, nypix, rotate, equinox,
        epoch, proj)
```

The arguments here are:

ra	Center right ascension in degrees.
dec	Center declination in degrees.
secpix	Number of arcseconds per pixel.
xrefpix	Reference pixel X coordinate.
yrefpix	Reference pixel Y coordinate.
nxpix	Number of pixels along x-axis.
nypix	Number of pixels along y-axis.
rotate	Rotation angle (clockwise positive) in degrees.
equinox	Equinox of coordinates, 1950 and 2000 supported.
epoch	Epoch of coordinates, used for FK4/FK5 conversion no effect if 0.
proj	Projection

For changing the center of the projection, you can use the *shift* method:

```
wcs.shift(ra, dec, equinox);
```

The leaves all of the other parameters the same, but changes the center reference position. More information can be found in the man page WCS(3).

3.3 Image I/O

The main class interface for reading, writing and accessing images is called *ImageIO*. This is a reference counted class, so it can easily be shared among different classes without any memory management concerns. Class *ImageIO* manages a pointer to a subclass of *ImageIORep* that is specialized for a specific image type. There is currently only one class subclass, for FITS images, called *FitsIO*. One flexible way to create an *ImageIO* object is to pass the constructor a pointer to the subclass of *ImageIORep* to be managed:

```
ImageIO imio = FitsIO::read(filename);
```

The *FitsIO::read* static method returns a pointer to a *FitsIO* object, created by reading the given file.

3.3.1 Adding New Image Formats

Currently, we only supports FITS format images. Although it is planned to keep FITS as the internal image format, we do plan to add support for other image formats. This will be done by deriving subclasses the abstract base class *ImageIORep*. Note that The public interface is through the class *Im-*

ageIO, which uses a pointer to an *ImageIORep* subclass for reference counting.

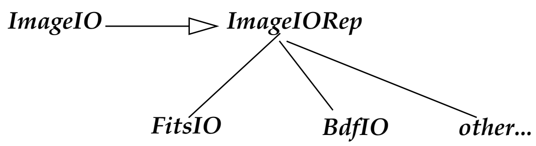

Figure 1: *Class hierarchy for Image I/O*

To add support for a new image format, you can derive a class from ImageIORep that reads in the image and converts it to FITS format. (Note: Class BdfIO, for Midas BDF images, is not implemented yet).

3.4 Image Compression

There are many situations in which you need to compress or decompress images. For example, to save space in an archive, or to save network transfer time. The *Compress* class provides methods to compress and decompress images using one of the following compression types:

Keyword	Description
UNIX_COMPRESS	UNIX style compress, as with the UNIX compress(1) command
H_COMPRESS	H-compress (lossy compression, reduces the amount of data)
ULDA_COMPRESS	ULDA compressed FITS file
GZIP_COMPRESS	Compress using the GNU gzip command.

The following example compresses and then decompresses the file “test.fits” using the H-compress compression type:

```
if (c.compress(“test.fits”, “test.hfits”, Compress::H_COMPRESS) != 0)
      return ERROR;

if (c.decompress(“test.hfits”, “test.fits”, Compress::H_COMPRESS) != 0)
      return ERROR
```

See the man page Compress(3) for more details.

4 REFERENCE

This section contains man pages for the Itcl and C++ classes and other utilities.

4.0.1 Compress(3)

NAME

Compress - utility class for compressing/decompressing FITS files

SYNOPSIS

```
class Compress {
protected:

public:
    enum CompressType {
        NO_COMPRESS,            // no compression
        UNIX_COMPRESS,          // Compressed FITS file (UNIX)
        H_COMPRESS,             // Hcompressed FITS file
        ULDA_COMPRESS,          // ULDA compressed FITS file
        GZIP_COMPRESS           // GZIPed FITS file
    };


    Compress();

    int compress(int read_fd, int write_fd, CompressType type,
                 int compress_flag = 1);

    int decompress(int read_fd, int write_fd, CompressType type);

    int compress(const char* infile, const char* outfile, CompressType type,
                 int compress_flag = 1, int mmap_flag = 1);

    int decompress(const char* infile, const char* outfile, CompressType type,
                   int mmap_flag = 1);

    int compress(const char* file, CompressType type, int compress_flag = 1,
                 int mmap_flag = 1);

    int decompress(const char* file, CompressType type, int mmap_flag = 1);


    int compress(const char* inbuf, int inbufsz, char*& outbuf, int& outbufsz,
                       CompressType ctype, int compress_flag = 1);

    int decompress(const char* inbuf, int inbufsz, char*& outbuf, int& outbufsz,
                   CompressType ctype);

};
```

DESCRIPTION

This class is a C++ wrapper around the CADC "press" routines for FITS image compression. The methods all do the same thing: compress or decompress an image. Some methods take file names as arguments, others pointers to memory areas. The type of compression is specified as an enum value "CompressType".

The methods called "compress" all take an optional flag argument that indicates compression or decompression. The methods called "decompress" are shortcut, inline methods that call compress with the decompress flag on.

CONSTRUCTORS

```
Compress()
        Initialize the object.
```

METHODS

```
compress(read_fd, write_fd, type, compress_flag)
        Compress (or decompress), reading from the given read file
        descriptor and writing the results to the given write fd. If
        compress_flag is true, compress, otherwise decompress the
        file.

 decompress(read_fd, write_fd, type)
        Decompress, reading from the given read file descriptor and
        writing the results to the given write fd.

compress(infile, outfile, type, compress_flag, mmap_flag)
        Compress (or decompress) the given input file and put the
        result in outfile.  If compress_flag is true, compress,
        otherwise decompress the file.  If mmap_flag is true, use mmap
        to map the file to memory.  Note: we can just open the file
        and use the fd, but the "press" C routines do unbuffered I/O
        on each char, which is slow. We can mmap the file to memory
        and use the "mem to mem" version to improve speed somewhat.

decompress(infile, outfile, type, mmap_flag)
        Decompress the given input file and put the result in outfile.

compress(file, type, compress_flag, mmap_flag)
        Compress (decompress) the file in place using the given
        compress type.  If compress_flag is true, compress, otherwise
        decompress the file.

decompress(file, type, mmap_flag)
        Decompress the file in place.

compress(inbuf, inbufsz, outbuf, outbufsz, ctype, compress_flag)
        Compress (or decompress) the contents of inbuf using the given
        compress type and allocate the results to outbuf.  inbufsz is
        the size of the input buffer.  outbufsz is an estimate of the
        outbuf size on input and the actual size on output.  If
        compress_flag is true, compress, otherwise decompress the
        file.  It is the caller's responsibility to free() the outbuf
        when no longer needed.

decompress(inbuf, inbufsz, outbuf, outbufsz, ctype)
        Decompress inbuf and allocate results in outbuf.
```

```
- - - - - -
Last change: 02 Feb 98
```

4.0.2 FitsIO(3)

NAME

FitsIO - C++ Class for Reading, Writing and Managing FITS Images

SYNOPSIS

```
#include "FitsIO.h"

class FitsIO : public ImageIORep {
...
public:
    FitsIO(int width, int height, int bitpix, double bzero,
           double bscale, const Mem& header, const Mem& data);

    ~FitsIO() {}

    static FitsIO* read(const char* filename, int memOptions = 0);
    int write(const char *filename) const;

    static FitsIO* initialize(Mem& header);
    static FitsIO* initialize(Mem& header, Mem& data);

    static FitsIO* blankImage(double ra, double dec, double equinox,
                              double radius, int width, int height,
                              unsigned long color0);

    int get(const char* keyword, double& val) const;
    int get(const char* keyword, float& val) const;
    int get(const char* keyword, int& val) const;
    char* get(const char* keyword) const;

    static int get(const Mem& header, const char* keyword, double& val);
    static int get(const Mem& header, const char* keyword, float& val);
    static int get(const Mem& header, const char* keyword, int& val);
    static char* get(const Mem& header, const char* keyword);

    int getFitsHeader(ostream& os) const;

    int put(const char* keyword, double val, const char* comment);
    int put(const char* keyword, float val, const char* comment);
    int put(const char* keyword, int val, const char* comment);
    int put(const char* keyword, const char* val, const char* comment);
};
```

DESCRIPTION

This class handles the reading and writing of FITS images and access to the FITS header and keywords. It is based in part on the saoimage routines in the wcslib directory. Currently the reading and writing is implemented in this class, but that may be replaced by a standard FITS I/O package in the future.

This class is a subclass of ImageIORep, which is the internal class used by class ImageIO for reference counting. The public interface is generally through the ImageIO class, although class FitsIO may be used directly in cases where you know that the image is already in FITS format or you are creating a new image in FITS format, which is the internal format used by the RTD.

Besides reading and writing FITS images, this class can be used to create a FITS image from data in memory and to create a "blank" image with World Coordinate information for plotting astronomical objects.

CREATING A FITSIO OBJECT

To create a FitsIO object from a FITS file, use the read method, which returns a pointer to an allocated FitsIO object given the file name. If you have the image data in memory, you can use one of the constructors to create the object.

METHODS

FitsIO(width, height, bitpix, bzero, bscale, header, data)

This constructor is called by the static "read" method once the FITS file has been read. The parameters are based on values read from the FITS header. The header and data arguments are instances of class Mem(3), which uses reference counting to manage shared and unshared memory.

read(filename, int memOptions = 0)

Read a FITS file and return an initialized FitsIO object for it, or NULL if there are errors. If filename is "-", stdin is read into a temp image file and used as the input. The Mem class is used to speed up loading the file. The optional mem_options argument controls whether the memory is mapped read-only or read/write. See class Mem(3) for the available options.

write(filename)

Write the data to a FITS file.

blankImage(ra, dec, equinox, radius, width, height, color0)

Generate a blank image with a FITS header based on the given fields and including support for World Coordinates. RA and DEC are specified in degrees. Width and Height are in pixels. "color0" is the value to use for the image pixels (usually the value for "black").

get(keyword, val)

Find and set the value for the given FITS keyword and return 0 if OK (found). This method is overloaded for various data types.

get(keyword)

Find and return the string value for the given FITS keyword, or NULL if not found.

getFitsHeader(os)

Write an ASCII formatted copy of the FITS header to the given stream, format it in 80 char lines and replace any NULL chars with blanks.

put(keyword, val, comment)

Insert the given FITS keyword and value with the given comment in the FITS header and return 0 if all is OK. If there is not enough space in the FITS header, extend the size of the FITS header by one header block and if the header is part of an mmap'ed file, rewrite the file with the new enlarged header. This creates a temp file in the same dir as the file called <file>.tmp.

SEE ALSO

ImageIO(3), ImageData(3), Mem(3)

```
- - - - - -
Last change: 02 Feb 98
```

4.0.3 HMS(3)

NAME

HMS - C++ class for working with H:M:S (hours minutes seconds) values

SYNOPSIS

```
#include "HMS.h"

class HMS {
public:
    HMS();
    HMS(int hours, int min, double sec);
    HMS(double val);

    int isNull() const;

    int hours() const;
    int min() const;
    double sec() const;
    double val() const;

    friend ostream& operator<<(ostream&, const HMS&);
    friend istream& operator>>(istream&, HMS&);

    friend int operator< (const HMS& a, const HMS& b);
    friend int operator<=(const HMS& a, const HMS& b);
    friend int operator> (const HMS& a, const HMS& b);
    friend int operator>=(const HMS& a, const HMS& b);
    friend int operator==(const HMS& a, const HMS& b);
    friend int operator!=(const HMS& a, const HMS& b);
};
```

DESCRIPTION

HMS is a simple class for representing values in Hours:Minutes:Seconds format or in degrees, converting back and forth and comparing.

CONSTRUCTORS

HMS()

Initialize a NULL H:M:S value (see isNull() method).

HMS(hours, min, sec)

Initialize from hours, minutes and seconds.

HMS(val)

Initialize from a value in degrees.

METHODS

isNull()

Return true if the HMS object has the NULL value (created with default constructor).

hours()
min()
sec()
val()

Return the parts of the H:M:S value or the value (val) in degrees.

```
operator<<(ostream, hms)
operator>>(istream, hms)
        I/O operators - using format H:M:S.sss.

operator< (a, b);
operator<=(a, b);
operator> (a, b);
operator>=(a, b);
operator==(a, b);
operator!=(a, b);
        Comparison operators - compare two H:M:S values.
```

SEE ALSO

WorldCoords(3)

\- - - - - -

Last change: 02 Feb 98

4.0.4 ImageCoords(3)

NAME

ImageCoords - class representing image coordinates (x, y)

SYNOPSIS

```
#include "ImageCoords.h"

class ImageCoords {
...
public:
    ImageCoords();
    ImageCoords(double x, double y);
    ImageCoords(const char* x_str, const char* y_str);

    isNull() const;
    void setNull();

    friend ostream& operator<<(ostream&, const ImageCoords& pos);
    void print(char* x_buf, char* y_buf);
    void print(ostream& os);

    void get(double& x, double& y);

    int operator==(const ImageCoords& pos) const;
    int operator!=(const ImageCoords& pos) const;

    friend ImageCoords operator-(const ImageCoords& a, const ImageCoords& b);

    double x() const;
    double y() const;

    double dist(ImageCoords& pos) const;
    static double dist(double x0, double y0, double x1, double y1);

    int box(double radius, ImageCoords& pos1, ImageCoords& pos2) const;

    static ImageCoords center(const ImageCoords& pos1, const ImageCoords& pos2,
                              double& radius, double& width, double& height);
    int status() const;
};
```

DESCRIPTION

This class is used to represent image pixel coordinates (x, y). It has an interface similar to the WorldCoords(3) class, and is used by the WorldOrImageCoords(3) class to represent image coordinates.

CONSTRUCTORS

ImageCoords()
    Initialize null coordinates.

ImageCoords(x, y)
    Initialize coordinates with (x, y)

ImageCoords(x_str, y_str)
    Parse X and Y in string format.

METHODS

```
isNull()
        Return true if the coordinates are null.

setNull()
        Set to the null value.

ostream& operator<<(os, pos)
        Output operator: format: "x y"

print(x_buf, y_buf)
        Print the coordinates to the given buffers.

print(os)
        Print coordinates to the given stream.

get(x, y)
        Get the values of x and y.

operator==(pos)
        Check for equality.

operator!=(pos)
        Check for inequality.

operator-(a, b) {
        Return the difference between two points.

x()
        return the value of x.

y()
        return the value of y.

dist(pos)
        Get distance between this point and the given one.

dist(x0, y0, x1, y1)
        Static member to get the distance between two points.

box(radius, Ipos1, pos2)
        Given a radius, set pos1 and pos2 to the two endpoints that
        form a box with center at this position.

center(pos1, const pos2, radius, width, height)
        Given the endpoints of a box (pos1, pos2), set width, height
        and radius and return the center position of the box.

status()
        Return the status ater the constructor (0 is OK).
```

SEE ALSO

WorldCoords(3), WorldOrImageCoords(3)

\- - - - - -
Last change: 02 Feb 98

4.0.5 ImageIO(3)

NAME

ImageIO - C++ abstract base Class for Reading and Writing Images

SYNOPSIS

```
#include "FitsIO.h"

// types of image data (these mostly correspond to the FITS BITPIX values)
enum ImageDataType {
    UNKNOWN_IMAGE  = -1,        // unknown type
    BYTE_IMAGE  = 8,            // 8 bit images
    X_IMAGE  = -8,              // special, color scaled, X image data
    SHORT_IMAGE = 16,           // 16 bit signed
    USHORT_IMAGE = -16,         // 16 bit unsigned
    LONG_IMAGE = 32,            // 32 bit integer
    FLOAT_IMAGE = -32           // 32 bit floating point
};

// image formats
enum ImageFormat {
    FITS_IMAGE  = 1             // FITS image
    ...
};


class ImageIORep {
friend class ImageIO;
...
public:
    virtual ~ImageIORep() {}

    virtual int get(const char* keyword, double& val) const = 0;
    virtual int get(const char* keyword, float& val) const = 0;
    virtual int get(const char* keyword, int& val) const = 0;
    virtual char* get(const char* keyword) const = 0;

    virtual int write(const char *filename) const = 0;
    double scaleValue(double d) const;
    double unScaleValue(double d) const;

    // member access
    int width() const;
    int height() const;
    int bitpix() const;
    double bscale() const;
    double bzero() const;
    int status() const;
    void status(int s);

    Mem& header();
    Mem& data();
    int header(const Mem&);
    int data(const Mem&);
};


class ImageIO;
...
public:
    ImageIO(int width, int height, int bitpix, double bzero, double bscale,
           const Mem& header, const Mem& data, ImageFormat);
```

```
    ImageIO(const ImageIO&);
    ImageIO(ImageIORep* rep) : rep_(rep);
    ~ImageIO();
    ImageIO& operator=(const ImageIO&);

    static ImageIO read(const char* filename, ImageFormat);
    int write(const char *filename) const;

    int get(const char* keyword, double& val) const;
    int get(const char* keyword, float& val) const;
    int get(const char* keyword, int& val) const;
    char* get(const char* keyword) const;

    double scaleValue(double d) const;
    double unScaleValue(double d) const;

    // member access
    int width() const;
    int height() const;
    int bitpix() const;
    double bscale() const;
    double bzero() const;
    const Mem& header() const;
    const Mem& data() const;

    int header(const Mem& m);
    int data(const Mem& m);
    const char* headerPtr() const;
    const void* dataPtr() const;

    int status() const;

};
```

DESCRIPTION

Class ImageIO is used to read, write and manage images of various formats. The image header and data are optionally kept in shared memory, so they can be accessed by external processes that may want to do image processing or other operations. Regardless of the original image format and derived class, the header and data are always kept in FITS format. Other image types are converted to FITS format by the derived classes (see below).

CLASS STRUCTURE

Class ImageIO uses reference counting to make it easier to share objects of this type for displaying in multiple windows. The actual base class is ImageIORep. An ImageIO object contains a pointer to an ImageIORep object, which may be shared by multiple ImageIO objects. To add new image types, to the ImageIO class, classes are derived from ImageIORep.

METHODS

ImageIO(width, height, bitpix, bzero, bscale, header, data, format)

This constructor calls the correct ImageIORep derived class constructor for the given image format. The parameters are based on values in the FITS image header. The header and data arguments are instances of class Mem(3), which uses reference counting to manage shared and unshared memory. Note that the header and data should be in FITS format or converted to FITS

```
        format before calling this constructor.

read(filename, format)
        Read an image file of the specified format and return an
        ImageIO object for it. The member "status" indicates if this
        was successfull.

write(filename)
        Write the image header and data to a file in a format defined
        in the derived class.

get(keyword, val)
        Find and set the value for the given FITS keyword and return 0
        if OK (found). This method is overloaded for various data types.

get(keyword)
        Find and return the string value for the given FITS keyword,
        or NULL if not found.

scaleValue(d)
        Apply the FITS keyword values for BZERO and BSCALE to the
        given value (d = BZERO+d*BSCALE)

unScaleValue(d)
        Reverse the effect of BZERO and BSCALE (d = (d-BZERO)/BSCALE)

width()
        Return the width of the image in pixels.

height()
        Return the height of the image in pixels.

bitpix()
        Return the value of the BITPIX keyword (data type of image).

bscale()
        Return value for the BSCALE keyword.

bzero()
        Return the value for the BZERO keyword.

header()
        Return a reference to the FITS header (class Mem(3)).

data()
        Return a reference to the FITS image data (class Mem(3)).
```

SEE ALSO

FitsIO(3), ImageData(3), Mem(3)

\- - - - - -

Last change: 02 Feb 98

4.0.6 WCS(3)

NAME

WCS - C++ Wrapper class for SAO WCS C library

SYNOPSIS

```
#include "WCS.h"

class WCS {
...
public:
    WCS();
    WCS(const char* header);
    WCS(const WCS&);
    WCS(WCSRep* rep);
    ~WCS();
    WCS& operator=(const WCS&);

    int isWcs();

    int pixWidth() const;
    int pixHeight() const;
    double dist(double ra0, double dec0, double ra1, double dec1);
    char* pix2wcs(double x, double y, char* buf, int bufsz, int hms_flag = 1);
    int pix2wcs(double x, double y, double& ra, double& dec);
    int wcs2pix(double ra, double dec, double &x, double &y);
    int set(double ra, double dec, double secpix, int nxpix, int nypix,
    int shift(double ra, double dec, double equinox);
    double equinox();
    char* equinoxStr();
    double epoch();
    double rotate();
    double width();
    double height();
    double radius();
    double secPix();
    WorldCoords center();
    int status() const;
};
```

DESCRIPTION

Class WCS is a reference counted wrapper class for Doug Mink's (saoimage) WCS C library. The constructor takes a pointer to a FITS image header and uses that to set up world coordinates for the image. Methods are implemented to convert between image pixel and world coordinates and to set world coordinates when there is no FITS header. See the HTML documentation in http://tdc-www.harvard.edu/software/wcstools.html for information about the underlying C library. The latest version of the wcs C library can be found under: ftp://cfa-ftp.harvard.edu/pub/gsc/WCS/.

METHODS

WCS()
WCS(const char* header)

The default constructor creates a null WCS object. If you pass the constructor a pointer to the image FITS header, the object is initialized from it.

```
isWcs()
        Returns nozero if the image supports world coordinates (has
        the necessary FITS keywords in the image header or from some
        other source). Note that if this method returns 0, none of the
        other methods should be called.

pix2wcs(int x, int y, char* buf)
        Return the world coordinates string for the given image coords.

pix2wcs(int x, int y, double& ra, double& dec)
        Return the world coords (in degrees, as 2 doubles) for the
        image coords.

wcs2pix(double ra, double dec, int &x, int &y)
         Get the image coordinates for the given world coords.

dist(double ra0, double dec0, double ra1, double dec1)
        Return the WCS distance between the 2 given WCS points in
        arcmin.

width()
height()
radius()
        Return the width, height, radius of the image in arc-minutes.

set(double ra, double dec, double secpix, double xrefpix, double yrefpix,
    int nxpix, int nypix, double rotate, int equinox, double epoch,
    const char* proj);

        Set up the WCS structure from the given information about the
        image. This method can be used to add world coordinate support
        to an image even if the image doesn't have it in the header.

    Arguments:
        ra      = Center right ascension in degrees
        dec     = Center declination in degrees
        secpix  = Number of arcseconds per pixel
        xrefpix = Reference pixel X coordinate
        yrefpix = Reference pixel Y coordinate
        nxpix   = Number of pixels along x-axis
        nypix   = Number of pixels along y-axis
        rotate  = Rotation angle (clockwise positive) in degrees
        equinox = Equinox of coordinates, 1950 and 2000 supported
        epoch   = Epoch of coordinates, used for FK4/FK5 conversion no effect if 0
        proj    = Projection

shift(double ra, double dec, double equinox)
    Reset the center of the WCS structure.

equinox()
equinoxStr()
        Return the WCS equinox as a double (1) or a string (2).

epoch()
        Return the WCS epoch as a double.

rotate()
        Return the rotation angle in degrees.

secPix()
        Return the number of world coordinate arcseconds per pixel.

center()
```

```
        Return the world coordinates of the center of the image.

pixWidth() const
pixHeight()
        Return image dimensions in pixels.

status()
status(int s)
        Get/set the status of the object (0 is OK).
```

SEE ALSO

ImageData(3), RtdImage(3), WorldCoords(3), FitsIO(3), ImageIO(3)

\- - - - - -

Last change: 02 Feb 98

4.0.7 WorldCoords(3)

NAME

WorldCoords - C++ class for working with world coordinates

SYNOPSIS

```
#include "WorldCoords.h"

class WorldCoords {
...
public:
    WorldCoords();
    WorldCoords(const HMS& ra, const HMS& dec,
                double equinox = 2000.0);

    WorldCoords(double ra_deg, double dec_deg,
                double equinox = 2000.0);

    WorldCoords(int rh, int rm, double rs,
                int dd, int dm, float ds,
                double equinox = 2000.0);

    WorldCoords(const char* ra_str, const char* dec_str,
                double equinox = 2000.0);

    isNull() const;
    void setNull();

    void print(char* ra_buf, char* dec_buf, double equinox = 2000.0);

    friend istream& operator>>(istream&, WorldCoords& pos);
    friend ostream& operator<<(ostream&, const WorldCoords& pos);

    friend int operator==(const WorldCoords& a, const WorldCoords& b);
    friend int operator!=(const WorldCoords& a, const WorldCoords& b);

    friend WorldCoords operator-(const WorldCoords& a, const WorldCoords& b);

    const HMS& ra() const;
    const HMS& dec() const;

    int status();
};
```

DESCRIPTION

This class is used to represent world coordinates and to convert between different representations of world coordinates. The member variables in the class are always kept internally in both H:M:S format and as floating point values in the default equinox J2000.

For example, for a given WorldCoords object "pos", the coordinates in H:M:S[+-]D:M:S format are given by:

```
(pos.ra(), pos.dec())
```

ra() and dec() both return a reference to an HMS object, which is used to represent values in hours, minutes and seconds (see HMS(3)). The coordinates in decimal degrees are given by

```
(pos.ra().val()*15, pos.dec().val()).
```

The ra value must be multiplied by 15 to convert from hours to degrees.

CONSTRUCTORS

```
WorldCoords()
        Initialize null world coordinates.
        (Null coordinates are useful in some cases where you have an
        optional second position for an area instead of a single
        point).

WorldCoords(ra, dec, equinox)
        Initialize the world coordinates from RA, DEC in H:M:S D:M:S
        format.
        Arguments:
                ra      (in)     - RA in H:M:S
                dec     (in)     - DEC in D:M:S
                equinox (in)     - optional epoch (2000.0, 1950.0, ...)

WorldCoords(rh, rm, rs, dd, dm, ds, equinox)
        Initialize the world coordinates from H M S D M S as separate
        values.

        Arguments:
                rh      (in)     - RA hours
                rm      (in)     - RA minutes
                rs      (in)     - RA seconds
                dd      (in)     - DEC degrees
                dm      (in)     - DEC minutes
                ds      (in)     - DEC seconds
                equinox (in)     - optional epoch (2000.0, 1950.0, ...)

WorldCoords(ra_deg, dec_deg, equinox)
        Initialize the world coordinates from RA, DEC in degrees in
        floating point format.
        Arguments:
                ra      (in)     - right ascension
                dec     (in)     - declination
                equinox (in)     - epoch (2000.0, 1950.0, ...)

WorldCoords(ra_str, dec_str, equinox);
        Initialize the world coordinates from RA and DEC in string
        format. Allowed formats of input strings:

                hh mm ss.s +/-dd mm ss.s
                hh:mm:ss.s +/-dd:mm:ss.s
                h.hhhh +/-d.dddd

        Arguments:
                ra      (in)     - String containing right ascension
                dec     (in)     - String containing declination
                equinox (in)     - epoch (2000.0, 1950.0, ...)
```

METHODS

```
isNull()
        Return true if the given coords are null (this is true if they
        were created with the default constructor - i.e.: with no
        arguments).

setNull()
        Set to the null value.
```

```
print(ra_buf, dec_buf, equinox)
        Print the coordinates in the given buffers in H:M:S format in
        given equinox.

operator>>(istream, pos)
        Input operator: format: H:M:S[+-]D:M:S - reads the position
        from the given istream.

operator<<(ostream, pos)
        Output operator: format: H:M:S[+-]D:M:S - writes the position
        to the given ostream.

operator==(pos1, pos2)
operator!=(pos1, pos2)
        Comparison operators - check for equality/inequality.

ra()
        Return the RA part of the coordinates (class HMS). ra().val()*15
        is the value in degrees, ra.hours(), ra().min() and ra().sec()
        give the H:M:S values.
dec()
        Return the DEC part of the coordinates (class
        HMS). dec().val() is the value in degrees, dec.hours(),
        dec().min() and dec().sec() give the H:M:S values.

status()
        Return the status after the constructor is done (0 is OK, 1
        for errors).
```

SEE ALSO

HMS(3)

\- - - - - -

Last change: 02 Feb 98

4.0.8 worldCoords(3)

NAME

worldCoords - C utility routine for working with world coordinates

SYNOPSIS

```
#include "worldCoords.h"

typedef struct {
    int hours;
    int min;
    double sec;
    double val;                /* value calculated in degrees */
} WC_HMS;


typedef struct {
    WC_HMS ra, dec;
} WC;


WC* wcInitNull(WC*);

int wcIsNull(WC* wc);

WC* wcInitFromHMS(WC*, int rh, int rm, double rs,
        int dd, int dm, int ds, double equinox);

WC* wcInitFromDeg(WC*, double ra, double dec, double equinox);

WC* wcInitFromStrings(WC*, char* ra, char* dec, double equinox);

void wcPrint(WC* wc, char* ra_buf, char* dec_buf, double equinox);
```

DESCRIPTION

The routines described here present a C interface to the C++ WorldCoords class and are used to convert between different representations of world coordinates. The values in the struct are always kept in both H:M:S[+-]D:M:S format (members: hours, min, sec) and as floating point values in hours (member: val) in the default equinox of J2000.

C ROUTINES

```
wcInitNull
        Initialize null world coordinates and return the argument.
        (Null coordinates are useful in some cases where you have
        an optional second position for an area instead of a single
        point).
        Arguments:
                wc   (in/out) - pointer to WC struct to be filled in
        Return value:
                pointer to wc argument.

wcIsNull
        Return true if the given coords are null.
        Arguments:
                wc   (in) - pointer to initialized WC struct
        Return value:
                boolean value
```

```
wcInitFromHMS
        Initialize the world coordinates from RA, DEC in H:M:S D:M:S
        format and return the first parameter.
        Arguments:
                wc      (in/out) - pointer to WC struct to fill out
                rh      (in)     - RA hours
                rm      (in)     - RA minutes
                rs      (in)     - RA seconds
                dd      (in)     - DEC degrees
                dm      (in)     - DEC minutes
                ds      (in)     - DEC seconds
                equinox (in)     - epoch (2000.0, 1950.0, ...)
        Return value:
                pointer to wc argument

wcInitFromDeg
        Initialize the world coordinates from RA, DEC in degrees in
        floating point format and return the first parameter.
        Arguments:
                wc      (in/out) - pointer to WC struct to fill out
                ra      (in)     - right ascension
                dec     (in)     - declination
                equinox (in)     - epoch (2000.0, 1950.0, ...)
        Return value:
                pointer to wc argument

wcInitFromStrings
        Initialize the world coordinates from RA, DEC in the format
        "H:M:S.sss", "[+-]D:M:S.sss" and return the first parameter.
        The minutes and seconds are optional, so the formats
        "H.hhh", "[+-]D.ddd" and "H:M", "D:M" are also supported.
        Arguments:
                wc      (in/out) - pointer to WC struct to fill out
                ra      (in)     - right ascention
                dec     (in)     - declination
                equinox (in)     - epoch (2000.0, 1950.0, ...)
        Return value:
                pointer to wc argument

wcPrint
        Print RA and DEC to the given buffers in the given equinox.
        Arguments:
                wc      (in)     - pointer to WC struct
                ra_buf  (in)     - buffer to contain RA in H:M:S
                dec_buf (in)     - buffer to contain DEC in D:M:S
                equinox (in)     - desired epoch (2000.0, 1950.0, ...)
        Return value:
                none.
```

SEE ALSO

WorldCoords(3)

\- - - - - -

Last change: 02 Feb 98

4.0.9 WorldOrImageCoords(3)

NAME

WorldOrImageCoords - class representing either world or image coordinates

SYNOPSIS

```
#include "WorldOrImageCoords.h"

class WorldOrImageCoords {
...
public:
    WorldOrImageCoords();
    WorldOrImageCoords(WorldCoords wc);
    WorldOrImageCoords(ImageCoords ic);

    isNull() const;
    void setNull();

    friend ostream& operator<<(ostream& os, const WorldOrImageCoords& pos);
    void print(char* x_buf, char* y_buf, double equinox = 2000., int hmsFlag=1);
    void print(ostream& os, double equinox = 2000.);

    void get(double& x, double& y, double equinox = 2000.);

    int operator==(const WorldOrImageCoords& pos) const;
    int operator!=(const WorldOrImageCoords& pos) const;
   friend WorldOrImageCoords operator-(const WorldOrImageCoords& a, const WorldOrIm-
ageCoords& b);

    double ra_deg() const;
    double dec_deg() const;

    WorldCoords& wc();
    ImageCoords& ic();

    const WorldCoords& wc() const
    const ImageCoords& ic() const;

    double dist(WorldOrImageCoords& pos, double& pa) const;
    double dist(WorldOrImageCoords& pos) const;

    int box(double radius, WorldOrImageCoords& pos1, WorldOrImageCoords& pos2) const;

    static WorldOrImageCoords center(const WorldOrImageCoords& pos1,
                                     const WorldOrImageCoords& pos2,
                                     double& radius,
                                     double& width, double& height);

    const HMS& ra() const;
    const HMS& dec() const;
    double x() const;
    double y() const;
    double isWcs() const;
    int status() const;
};
```

DESCRIPTION

```
Class WorldOrImageCoords is designed to be used in situations where
you might need to deal with either world coordinates or image
coordinates, but don't know ahead of time which.  We could use a
```

common base class and virtual methods, but that would require using pointers or references, while it is often more convenient to use instances of these classes. Also, the methods in both classes have slightly different signatures.

After initializing the class with with a WorldCoords(n) object or an ImageCoords(n) object, you can use the isWcs() method to check which one it is, if you don't know. Otherwise, most of the methods work transparently, independent of the coordinate type. However, you should not attempt, for example, to fetch the value of "ra" and "dec" without being sure that the object represents world coordinates.

CONSTRUCTORS

```
WorldOrImageCoords()
        Initialize null coordinates.

WorldOrImageCoords(wc)
        Initialize world coordinates.

WorldOrImageCoords(ic)
        Initialize image coordinates.
```

METHODS

```
isNull();
        Return true if the coordinates are null

setNull()
        Set to the null value.

operator<<(os, pos)
        Output operator, prints world coordinates as hh:mm:ss dd:mm:ss,
        image coordinates as x y.

print(x_buf, y_buf, equinox, hmsFlag)
        Print the coordinates to the given buffer. For world
        coordinates, if hmsFlag is non-zero, prints in H:M:S [+-]D:M:S
        format, otherwise in decimal degrees.


print(os, equinox)
        Print the coordinates to the given stream.

get(x, y, equinox)
        Get teh valules of x and y in the given equinox.

operator==(pos)
        Check for equality.

operator!=(pos)
        Check for inequality.

operator-(a, b)
        Return the difference between two points.

ra_deg()
        Return ra in degrees.

dec_deg()
        Return dec in degrees.

wc()
```

```
        Return the internal WorldCoords class.

ic()
        Return the internal ImageCoords class.

dist(pos, pa)
        Get distance and pa angle between the given point and this point.

dist(pos)
        Get distance between between the given point and this point.

box(radius, pos1, pos2)
        Given a radius, set pos1 and pos2 to the 2 endpoints that form
        a box with center at this position.

center(pos1, pos2, radius, width, height) {
        Given the endpoints of a box (pos1, pos2), set width, height
        and radius and return the center position of the box.

ra()
        Return the value of ra.

dec()
        Return the value of dec.

x()
        Return the value of the x coordinate.

y()
        Return the value of the y coordinate.

isWcs()
        Return true if this class represents world coordinates.

status()
        Return the status after the constructor (0 is OK).
```

SEE ALSO

WorldCoords(3), ImageCoords(3)

\- - - - - -

Last change: 02 Feb 98

4.0.10

5 Installing the Astrotcl Package

5.1 Before you build the Astrotcl Package...

Make sure you have a proper Itcl-2.2 distribution (Tcl, Tk, BLT, TclX and ITCL extensions) with the necessary patches applied. Astrotcl requires the following software to be already installed (not included):

- itcl-2.2 - [Incr Tcl] (includes tcl7.6, tk4.2)
- BLT-2.1 - BLT Toolkit
- tclX-7.6.0 - Extended Tcl

These packages are available from the TCL archives.

See: http://www.tcltk.com/ for Itcl.

and http://www.NeoSoft.com/tcl/ for TclX and other contributed Tcl software

You can also get a copy of the whole Tcl/Tk source tree, with the patches already applied from ftp://ftp.archive.eso.org/pub/skycat/.

Astrotcl also depends on the following package:

- Tclutil - Tcl and C++ Utilities Package

which is available from the same location as this package: ftp://ftp.archive.eso.org/pub/skycat/.

5.2 Build the Astrotcl Package

To make the Astrotcl package, configure and make as follows:

```
configure
make
make install
```

The default install dir is /usr/local. You can specify the -prefix argument to configure to change this:

```
configure -prefix $INSTALLDIR
```

Note: The Astrotcl configure script reads configure information produced by the Tclutil configure script. If you want to set different configuration options, for example, to use a different compiler or to enable or disable shared libraries, reconfigure the Tclutil package first and then this package.

5.3 VLT Make Procedure

As an alternative to running configure and make, you can also do this:

```
cd src
make
make install
```

The Makefile in the $ASTOTCL/src directory runs configure and then make as described above. You can also specify options to that Makefile, for example:

```
cd src
make PREFIX=$INSTALLDIR
```

The PREFIX variable defaults to /usr/local and is the prefix of the directory in which to install the software.

5.4 Start the demo application

To run the demo, type:

```
INSTALLDIR/lib/astrotcl/demos/astrotcl
```

This package does not currently include any applications, so this just starts astrotcl_wish with the correct environment.

5.5 If you are using shared libraries

The Astrotcl shared library libastrotcl.sl (in HPs) or libastrotcl.so (on Suns) is built with the same options used to build the Tclutil and Tcl shared libraries. The options are read from the file tclutilConfig.sh, which is produced the Tclutil configure script.

You may need to modify the SHLIB_PATH (HP) or LD_LIBRARY_PATH (Sun) environment variable so that the necessary shared libraries are found at run time. Both variables have the same format: a colon ":" separated list of directories to search for shared libraries.

From a tcl script you can load the ASTROTCL library dynamically with the command "load <path>/libastrotcl.sl" or "load <path>/libastrotcl.so" or it can be loaded automatically as a *package*. See the Tcl man pages for more information.