SUN/160.6

Starlink Project
Starlink User Note 160.6

P.W. Draper
R.F. Warren-Smith

3 March 2003

IMG
Simple Image Data Access
Version 1.3
Subroutine Library

Abstract

IMG is a subroutine library for accessing astronomical image data and associated header information. It is designed to be easy to use and understand.

Contents

1 INTRODUCTION

This document describes the IMG library and how to use it to access astronomical images and header information from within your programs. The library has been designed to be easy to use and understand and will be of most interest to astronomers who write their own software and require uncomplicated access to their data. In this context, an "image" may be an ordinary (2-D) image, but could also be a spectrum (1-D) or "data cube" (3-D).

The data format used by the IMG library will normally be the Starlink "Extensible N-Dimensional Data Format" or NDF (described in SUN/33), which is typically stored in files with a `.sdf` extension. However, IMG can also access other astronomical formats (such as IRAF, disk FITS or FIGARO) and new ones can easily be added, so using IMG gives your programs access to a wide range of astronomical data in a very straightforward way (see §5.2).

IMG can be used from the Fortran-77 and C programming languages and some familiarity with one of these is assumed, as is knowledge of the UNIX C-shell. If you want to use IMG from C then you should briefly study the earlier sections (which are Fortran specific, but nevertheless show the appropriate calling sequences and discuss more general issues) and then look at appendix C.

2 ACCESSING IMAGE DATA

The easiest way to understand IMG is to look at how simple example programs work. That's what this section does. The examples are generally snippets of more complete programs that can be found in the directory `/star/bin/examples/img` (on non-Starlink machines replace `/star` with wherever you have the software installed). You are encouraged to copy and modify these for your own use.

SUN/101 also contains examples of programming techniques that you might want to to use with IMG. You should ideally read these two documents together, although this isn't absolutely necessary.

2.1 Accessing an existing image

The first example is a complete program. It gets an existing image, works out its mean value and then writes it out:

```
      SUBROUTINE MEAN( ISTAT )                                  [1]

*  Access an input image.
      CALL IMG_IN( 'IN', NX, NY, IP, ISTAT )                    [2]

*  Derive the mean and write it out.
      CALL DOSTAT( %VAL( IP ), NX, NY, ISTAT )                  [3]

*  Free the input image.
      CALL IMG_FREE( 'IN', ISTAT )                              [4]
```

```
      END

      SUBROUTINE DOSTAT( IMAGE, NX, NY, ISTAT )
      INCLUDE 'SAE_PAR'                                        [5]
      REAL IMAGE( NX, NY )                                     [6]

      IF ( ISTAT .NE. SAI__OK ) RETURN                         [7]

*  Initialise the sum and loop over all elements of the image.
      SUM = 0.0
      DO 1 J = 1, NY
         DO 2 I = 1, NX
            SUM = SUM + IMAGE( I, J )
 2       CONTINUE
 1    CONTINUE

*  Write out the mean value.
      WRITE( *, * ) 'Mean = ', SUM / REAL( NX * NY )           [8]

      END
```

The following notes refer to the numbered statements:

(1) Programs that use IMG should *always* have a main subroutine with an `INTEGER` argument. This subroutine should be named after the program and be stored in a file with the same name (`mean` and `mean.f` in this case). You should consider this subroutine as a replacement for the file that normally has a `PROGRAM` statement in it.

(2) The call to `IMG_IN` gets the input image. The image is associated with the "parameter" `'IN'` which usually means that you will be prompted for the name of an image data-file. The subroutine then returns the size of the image – `NX` × `NY` – and a pointer `IP` to its data (don't panic – hold on for the next item).

(3) A call to the subroutine `DOSTAT`, which performs the real work, is now made. Since any realistically useful program needs to be able to handle images of any size, a cheat is necessary – this is the `%VAL(IP)` argument. Basically, this makes the pointer to the image data, `IP`, look just like an array declaration in the calling routine (i.e. like say `REAL IMAGE(NX,NY)`). This isn't standard Fortran, but should work on any machine that has IMG installed. Pointers are stored in `INTEGER` variables.

(4) A call to `IMG_FREE` is made for each image parameter to tidy up before a program ends.

(5) The `INCLUDE` statement in the subroutine `DOSTAT` inserts the contents of the file `'SAE_PAR'`. This defines some standard Fortran parameters for you, such as `SAI__OK` (note the double underscore). Include this file as standard as the parameters it defines are usually necessary.

(6) The image is now available as an adjustable 2-dimensional array within the subroutine `DOSTAT`.

(7) The value of the `ISTAT` argument is tested against `SAI__OK` at the start of `DOSTAT`. This is done in case an error has occurred earlier on (in which case `ISTAT` will not be equal to `SAI__OK`, so the remainder of `DOSTAT` will not execute).

Doing this means that the program will not crash if an earlier error results in (say) the `IMAGE` array containing rubbish, and it makes sure that you get a sensible error message at the end telling you what went wrong.

(8) The calculation result is written out. In a more sophisticated program you might want to use routines from the MSG library (SUN/104) to do this.

To complete the program another file is required. This is an "interface" file (which should be named after the program but with a file extension of `.ifl`). It contains a description of each parameter associated with an image. The following will do for this example program:

```
interface MEAN
   parameter IN
      prompt 'Input image'
   endparameter
endinterface
```

The main point is that it is necessary to include a `parameter ... endparameter` statement for each input or output image. SUN/101 and SUN/115 describe numerous other parameter options if you need more sophisticated control.

Once you have the two files `mean.f` and `mean.ifl` you can compile and run the program by following the instructions in §6.

To recap: The most important IMG concepts to remember from this example are:

(1) *Images are associated with a label known as a "parameter".*

(2) *Image data is accessed by a "pointer" that needs to be passed to a subroutine using the `%VAL` mechanism.*

(3) *It is necessary to call `IMG_FREE` to free images before the program ends.*

2.2 Creating a new image

This second example (taken from the program `flat.f` in the example directory - see §2), shows how to create a new image and then write values to it:

```
*  Create a new image.
      CALL IMG_NEW( 'OUT', 416, 578, IP, ISTAT )              [1]

*  Fill the array with the value 1.
      CALL DOFILL( %VAL( IP ), 416, 578, ISTAT )              [2]

*  Free the image.
      CALL IMG_FREE( 'OUT', ISTAT )                           [3]
      END

      SUBROUTINE DOFILL( IMAGE, NX, NY, ISTAT )
      INCLUDE 'SAE_PAR'
      REAL IMAGE( NX, NY )
      IF ( ISTAT .NE. SAI__OK ) RETURN
```

```
*  Loop over all the elements of the image setting them to 1.0.
      DO 1 J = 1, NY
         DO 2 I = 1, NX
            IMAGE( I, J ) = 1.0
 2       CONTINUE
 1    CONTINUE
      END
```

The following notes refer to the numbered statements:

(1) The call to `IMG_NEW` creates the new image dataset (usually prompting you for a name). The image size in this example is 416×578 pixels.

(2) The newly created image is passed to the subroutine `DOFILL` where all its elements are set to the value 1.0.

(3) Calling `IMG_FREE` ensures that the output image is correctly freed (avoiding possible loss of the data you have written).

2.3 Modifying images

One of the most common things that programs do is modify an existing image or create an image that is a modification of an existing one. The examples shown here deal with both these cases.

2.3.1 Modifying an image

This snippet of Fortran shows an existing image being accessed so that it can be modified in place (that is, without creating a new copy):

```
*  Access an existing image.
      CALL IMG_MOD( 'IN', NX, NY, IP, ISTAT )

*  Fill the image with a value.
      CALL DOFILL( %VAL( IP ), NX, NY, ISTAT )

*  Free the image.
      CALL IMG_FREE( 'IN', ISTAT )
```

2.3.2 Modifying a copy of an image

This example copies the input image first and then modifies the copy. This is essential if the input image needs to be kept. A complete program called `add.f` exists (see §2).

```
*  Access an existing image.
      CALL IMG_IN( 'IN', NX, NY, IPIN, ISTAT )

*  Create a new output image by copying the input image.
      CALL IMG_OUT( 'IN', 'OUT', IPOUT, ISTAT )              [1]
```

```
*  Modify the output image.
      CALL DOFILL( %VAL( IPOUT ), NX, NY, ISTAT )

*  Free the input and output images.
      CALL IMG_FREE( '*', ISTAT )                              [2]
```

The following notes refer to the numbered statements:

(1) The call to `IMG_OUT` creates a new output image associated with the parameter `'OUT'` by copying the one associated with the parameter `'IN'`. It returns a pointer `IPOUT` for the output image data.

(2) Notice that the names of the image parameters are not shown explicitly in this call to `IMG_FREE`. Using a `'*'` indicates that all known images should be freed.

3 ACCESSING HEADER INFORMATION

Most astronomical images have collections of numeric, character or logical values associated with them. Typically these consist of important calibration values, but they are also often used to describe the history or status of the data. Collections of such data, which are not actually part of the image array, are known as "header" information (this name derives from the fact that FITS[1] stores these values at the beginning – head – of a file). IMG refers to a single piece of such information as a "header item" and a separate set of subroutines exists to read, write, copy and make enquiries about them.

3.1 Reading header items

This example (taken from a complete program called `hdrread.f` – see §2) shows how to read the value of a header item from an image associated with the parameter `'IN'`:

```
*  Read the header item.
      CALL HDR_IN( 'IN', ' ', 'OBSERVER', 1, VALUE, ISTAT )
```

The call to `HDR_IN` specifies the image by using its parameter name – `'IN'` (argument 1). If you have not previously accessed this image you will probably be prompted for its file name at this point.

The 2nd, blank argument, indicates that an "ordinary" FITS header item is required (there are other possible "sources" of header information which are described later – §4.2.2).

The 3rd argument `'OBSERVER'` specifies the name of the header item required and the 4th argument `1` indicates that you want the first occurrence of that item (just in case it occurs several times, which some items are allowed to do).

The 5th argument `VALUE` returns the header item value as a character string. If the header item you requested doesn't exist, then `VALUE` is returned unchanged, so you may want to set it to a sensible value (say `'<unknown>'`) beforehand.

[1] FITS, the "Flexible Image Transport System" is the *de facto* standard for the interchange of astronomical data.

3.2 Writing header items

This example (taken from a complete program `hdrwrite.f` – see §2) shows how to write the value of a header item to an image associated with the parameter `'OUT'`:

```
*  Write the new header item.
      CALL HDR_OUT( 'OUT', ' ', 'OBSERVER', 'The observer',    [1]
     :              'Fred Bloggs', ISTAT )

*  Free the image.
      CALL IMG_FREE( 'OUT', ISTAT )                            [2]
```

The following notes refer to the numbered statements:

(1) The call to `HDR_OUT` specifies the image by using its parameter name – `'OUT'` (argument 1). If you have not previously accessed this image you will probably be prompted for its file name at this point. If you have already accessed the image then it should be either an output image or one accessed for modification, as you cannot write items to images accessed by `IMG_IN`.

The 2nd, blank argument, indicates that an "ordinary" FITS header item is to be written.

The 3rd argument (`'OBSERVER'`) specifies the name of the header item and the 4th argument `'The observer'` is a comment to store with the item.

The 5th argument (`'Fred Bloggs'`) is the header item value.

(2) The image is freed. If this wasn't done, the new header item would be lost.

4 ADDITIONAL FEATURES

The IMG library has more capabilities than have been covered in the preceding descriptions, which have been kept as simple as possible. You should read the following if you need to do more than has been covered so far.

4.1 More advanced image access

4.1.1 Getting workspace

Any reasonably complex program sooner or later needs to be able to store image information between processing stages. This program snippet (from an example called `shadow.f` – see §2) shows you how to create a temporary image for this purpose:

```
*  Access the input image.
      CALL IMG_IN( 'IN', NX, NY, IPIN, ISTAT )
      ...
*  Get an image-sized piece of workspace.
      CALL IMG_TMP( 'TEMP', NX, NY, IPTEMP, ISTAT )            [1]
      ...
*  Free all the images (this deletes the temporary image).
      CALL IMG_FREE( '*', ISTAT )                              [2]
```

The following notes refer to the numbered statements:

(1) The call to IMG_TMP creates a temporary image of the size you specify (in this case the same size as the input image). You will *not* be prompted for a file name for a temporary image, and *no* entry is needed for it in the program's interface file.

(2) The call to IMG_FREE will delete the temporary image.

4.1.2 Using "images" which are not 2-dimensional

IMG subroutines can access "image" data with between 1 and 3 dimensions. This allows you to handle spectra (1-D) and cubes (3-D) as well as traditional 2-dimensional images. The subroutine calls needed are almost identical to the normal ones (above) that assume 2 dimensions. They differ only in having more or less arguments for the dimension sizes. The following illustrates the possibilities:

```
CALL IMG_IN1( 'SPECTRUM', NX, IP, ISTAT )
CALL IMG_IN2( 'IMAGE', NX, NY, IP, ISTAT )
CALL IMG_IN3( 'CUBE', NX, NY, NZ, IP, ISTAT )
```

Note how the number of dimensions is specified by appending a number to the routine name, and how IMG_IN2 is just a synonym for IMG_IN (2 dimensions are assumed if you do not say otherwise).

As well as accessing input images, there are equivalent routines for performing most other IMG operations on images with between 1 and 3 dimensions. For instance, the following would create a new 3-dimensional image:

```
CALL IMG_NEW3( 'CUBE', NX, NY, NZ, IP, ISTAT )
```

When accessing an existing image with (say) 2 dimensions, you will also be able to access an appropriate slice from (say) a 3-dimensional image if you specify this when you are prompted for the image name (see §5.1).

4.1.3 Accessing images using different data types

IMG also allows access to images using data types other than REAL (which is the default). The most useful of these are INTEGER, INTEGER*2, and DOUBLE PRECISION. The subroutine calls needed are almost identical to the normal ones except that you should append a character code to the routine name to indicate the data type you require. The character codes (and their data types) are:

Character code	Fortran-77 data type	Description
R	REAL	Single precision
D	DOUBLE PRECISION	Double precision
I	INTEGER	Integer
W	INTEGER*2	Word
UW	INTEGER*2	Unsigned word
B	BYTE	Byte
UB	BYTE	Unsigned byte

so among the possibilities are the calls:

```
      CALL IMG_IND( 'IMAGE', NX, NY, IP, ISTAT )          [1]
      CALL IMG_INI( 'IMAGE', NX, NY, IP, ISTAT )          [2]
      CALL IMG_INR( 'IMAGE', NX, NY, IP, ISTAT )          [3]
```

The following notes refer to the numbered statements:

(1) Access an image using `DOUBLE PRECISION`.

(2) Access an image using `INTEGER`.

(3) Access an image using `REAL`. Note that this is a synonym for `IMG_IN`, since it also gets a 2-D `REAL` image.

You should declare the image array in your program to have the corresponding Fortran data type, as in the following example that accesses and modifies an image using `INTEGER` format:

```
*  Access an input image, allowing it to be modified.
      CALL IMG_MODI( 'IN', NX, NY, IP, ISTAT )            [1]

*  Fill the array with zeros.
      CALL DOZERO( %VAL( IP ), NX, NY, ISTAT )

*  Free the image.
      CALL IMG_FREE( 'IN', ISTAT )
      END

      SUBROUTINE DOZERO( IMAGE, NX, NY, ISTAT )
      INCLUDE 'SAE_PAR'
      INTEGER IMAGE( NX, NY )                             [2]

      IF ( ISTAT .NE. SAI__OK ) RETURN
      DO 1 J = 1, NY
         DO 2 I = 1, NX
            IMAGE( I, J ) = 0
 2       CONTINUE
 1    CONTINUE
      END
```

The following notes refer to the numbered statements:

(1) An `INTEGER` input image is accessed for modification

(2) In the subroutine `DOZERO` the image is declared as an adjustable dimension `INTEGER` array.

All IMG subroutines that access images have variants that allow you to use different data types. There are also versions that allow these to be mixed with different numbers of data dimensions. Some possibilities are:

```
      CALL IMG_IN1D( 'SPECTRUM', NX, IP, ISTAT )          [1]
      CALL IMG_MOD2I( 'IMAGE', NX, NY, IP, ISTAT )        [2]
      CALL IMG_NEW3W( 'CUBE', NX, NY, NZ, IP, ISTAT )     [3]
      CALL IMG_IN2R( 'IMAGE', NX, NY, IP, ISTAT )         [4]
```

The following notes refer to the numbered statements:

(1) This call accesses a 1-D image for reading using `DOUBLE PRECISION`.

(2) This call accesses a 2-D image for modification using the `INTEGER` data type.

(3) This call creates a new 3-D data cube with the word (Fortran `INTEGER*2`) data type.

(4) This call is a synonym for `IMG_IN`, since it gets a 2-D `REAL` image.

If your image isn't stored with the type that you ask for, then it will be converted from the stored type (if possible). If you modify the data it will be re-converted to the storage type when the image is freed. New images are stored using the data type you ask for.

The data types unsigned word (UW), signed byte (B) and unsigned byte (UB) are less commonly used. Indeed, the unsigned data types have no equivalents in Fortran and should be manipulated using the PRIMDAT library (SUN/39).

4.1.4 Accessing multiple images

IMG can access more than one image at a time using a single subroutine call. An example of this is:

```
*  Declare pointers.
      INTEGER IP( 3 )

*  Access images.
      CALL IMG_IN( 'BIAS,FLAT,RAW', NX, NY, IP, ISTAT )       [1]

*  Create a new output image by copying the RAW input image.
      CALL IMG_OUT( 'RAW', 'PROC', IPPROC, ISTAT )            [2]

*  Debias and flatfield data.
      CALL DOPROC( %VAL( IP( 1 ) ), %VAL( IP( 2 ) ),          [3]
     :             %VAL( IP( 3 ) ), NX, NY, %VAL( IPPROC ),
     :             ISTAT )

*  Free all the images.
      CALL IMG_FREE( '*', ISTAT )
      END

      SUBROUTINE DOPROC( BIAS, FLAT, RAW, NX, NY, PROC, ISTAT )
      INCLUDE 'SAE_PAR'
      REAL BIAS( NX, NY ), FLAT( NX, NY ), RAW( NX, NY ),     [4]
     :     PROC( NX, NY )

      IF ( ISTAT .NE. SAI__OK ) RETURN
      DO 1 J = 1, NY
         DO 2 I = 1, NX
            PROC( I, J ) = ( RAW( I, J ) - BIAS( I, J ) ) / FLAT( I, J )
2        CONTINUE
1     CONTINUE
      END
```

The following notes refer to the numbered statements:

(1) This call accesses three images `'BIAS'`, `'FLAT'` and `'RAW'`. Prompts for the actual image files will be made for each of these names and pointers to the image data will returned in IP, which should now be an array of size 3.

(2) The image `'RAW'` is copied to a new file. This also copies any header information, so is usually preferred to using `IMG_NEW`.

(3) The individual pointers are now passed to a subroutine so that the `'BIAS'`, `'FLAT'` and `'RAW'` images together with the copy of `'RAW'` can be accessed.

(4) The image arrays are declared as normal.

There are two advantages to accessing multiple images in one call in this way:

(1) All the images are made available to the program as if they were the same size, regardless of their actual sizes (the largest region that is common to all the images is selected).

(2) All the images are made available with the same data type (`REAL` in this case).

This means that corresponding values in each of the images can be directly inter-compared by your program, even if the images themselves have different sizes or data types.

4.1.5 Creating multiple images

IMG can also create more than one image using a single subroutine call. Some possibilities are:

```
CALL IMG_NEW( 'MODEL1,MODEL2', 1024, 1024, IP, ISTAT )  [1]
CALL IMG_OUT( 'TEMPLATE', 'OUT1,OUT2', IP, ISTAT )      [2]
CALL IMG_TMP( 'T1,T2,T3,T4', NX, NY, IP, ISTAT )        [3]
```

The following notes refer to the numbered statements:

(1) The call to `IMG_NEW` creates two new images with size 1024×1024. The variable `IP` should be declared as an `INTEGER` array of size 2.

(2) The image associated with parameter `'TEMPLATE'` is copied to two new images `'OUT1'` and `'OUT2'`. Again, the pointers are returned in an array `IP(2)`.

(3) Four temporary images of the same size and type are created.

4.1.6 Handling "bad" data

In practice, astronomical images will often contain values that are unreliable (or unknown) and should be ignored. For example, you might not be able to compute a result for every element of an image array (say where a divide by zero would occur), or your detector system may not generate sensible values everywhere. These missing data are usually flagged by setting them to a unique number that allows them to be recognised, and this is known as the "bad" data value.[2]

[2]It may also be known as the "invalid", "flagged" or even "magic" value, but the meaning is the same.

Unfortunately, unless you have complete control over your data, you cannot usually ignore the possibility that an image may contain some of these "bad" values. If you did, and processed the numbers as if they were valid data, you would at best get the wrong answer. More probably, your program would crash.

To prevent you being baffled by this pitfall, IMG will normally check whether your input images contain any bad values and will issue a warning if any are found. Then at least you know *why* your program crashed!

You can deal with bad data values in one of two ways. The simplest is to use a suitable program to detect them and replace them with a sensible alternative value (for example, the KAPPA - SUN/95 - program NOMAGIC will do this). Alternatively, you can modify your IMG program to take proper account of them.

If you decide to modify your program, you'll probably want to inhibit the checks that IMG makes. This is done by appending an exclamation mark (!) to the parameter name of each affected image. So, for instance:

```
      CALL IMG_IN( 'BIAS!,FLAT!,RAW!', NX, NY, IP, ISTAT )
```

inhibits bad data value checking for all the input images.

The following example is a version of the `mean.f` program (see §2.1) that shows the sort of changes needed to deal with bad data:

```
      SUBROUTINE MEAN( ISTAT )

*  Access an input image. Inhibit checks for bad pixels.
      CALL IMG_IN( 'IN!', NX, NY, IP, ISTAT )                [1]

*  Derive the mean and write it out.
      CALL DOSTAT( %VAL( IP ), NX, NY, ISTAT )

*  Free the input image.
      CALL IMG_FREE( 'IN', ISTAT )
      END

      SUBROUTINE DOSTAT( IMAGE, NX, NY, ISTAT )
      INCLUDE 'SAE_PAR'
      INCLUDE 'PRM_PAR'                                      [2]
      REAL IMAGE( NX, NY )

      IF ( ISTAT .NE. SAI__OK ) RETURN

*  Initialise the sum and loop over all elements of the image, checking
*  that every data value is not bad.
      SUM = 0.0
      N = 0
      DO 1 J = 1, NY
         DO 2 I = 1, NX
            IF ( IMAGE( I, J ) .NE. VAL__BADR ) THEN         [3]
               SUM = SUM + IMAGE( I, J )
               N = N + 1
            END IF
 2       CONTINUE
```

```
 1    CONTINUE

*  Write out the mean value.
      IF ( N .GT. 0 ) THEN
         WRITE( *, * ) 'Mean of ', N, ' values = ', SUM / REAL( N )
      ELSE
         WRITE( *, * ) 'Error: all data values are bad'
      END IF
      END
```

The following notes refer to the numbered statements:

(1) An input image is accessed without performing any checks for the presence of bad data.

(2) The file 'PRM_PAR' is included. This defines Fortran parameters for the values used to flag bad data (it should be used by all programs that handle bad data). The parameter names are type-dependent and have the form `VAL__BAD[x]`, where `[x]` is replaced by the character code for the data type you're processing. These codes are the same as those used when accessing images of different data types – see §4.1.3. So, for instance, with `REAL` data you would use `VAL__BADR`, while with `INTEGER` data you would use `VAL__BADI`, etc. Before compiling a program that includes this file you should execute the command `prm_dev`, which creates a soft link to the include file.

(3) Since every input value may now potentially be bad, each one must be checked before use. As the data type being processed is `REAL`, the bad data value we test against is `VAL__BADR`.

If your program accesses more than one input image, you must be careful to check each of them. For instance if you were adding two images together you'd need to do something like the following:

```
      DO 1 I = 1, NX
         IF ( A( I ) .NE. VAL__BADI .AND. B( I ) .NE. VAL__BADI ) THEN
            C( I ) = A( I ) + B( I )
         ELSE
            C( I ) = VAL__BADI
         END IF
1     CONTINUE
```

Where `A` and `B` are the "images" to be added and `C` is the image to store the result (in this case the images are really 1-D `INTEGER` spectra). Note how a bad value is assigned to the output image if we are unable to calculate a result.

If you need to know more about how to handle bad data you should consult SUN/33.

4.2 More advanced header access

4.2.1 Accessing header items using different data types

Header items can come in more than one data type, so extended HDR subroutines (as for IMG) are provided to access them. If the item isn't of the type you want, then a format conversion will be attempted. The way you specify the type you require is similar to that used for IMG routines – you just append the necessary character code to the routine name:

Character code	Fortran-77 data type	Description
C	CHARACTER	Character string
D	DOUBLE PRECISION	Double precision
I	INTEGER	Integer
L	LOGICAL	Logical
R	REAL	Single precision

Since all header item values can be converted to a character representation, this is the safest method to access an item of unknown type (which is why it is the method used by the ordinary HDR_IN subroutine). Some of the possible calls are:

```
CALL HDR_INR( 'IN', ' ', 'BSCALE', 1, BSCALE, ISTAT )    [1]
CALL HDR_INI( 'IN', ' ', 'BINNED', 1, IBFACT, ISTAT )    [2]
CALL HDR_OUTL( 'OUT', ' ', 'CHECKED',                    [3]
:              'Data checked for C/Rs', .TRUE., ISTAT )
```

The following notes refer to the numbered statements:

(1) This example reads in a REAL value BSCALE.

(2) This example reads in an INTEGER value BINNED.

(3) This example writes out a LOGICAL value CHECKED with the comment 'Data checked for C/Rs'.

4.2.2 Using header items from different sources

So far, header items have been assumed to originate only from one source (FITS). In reality, an image may have more than one set of header information, and these are distinguished by giving them different names. FITS headers are normally used for storing information that may be widely distributed, perhaps to people using different data reduction systems, whereas more private collections of header information may be created by using a name of your own choice. Private header information has the advantage that it is unlikely to be trampled on by other software (that believes it knows what the headers mean – rightly or wrongly) and is typically used for information needed only within a particular software package.[3]

Using header items from other sources is achieved simply by replacing ' ' (which is a synonym for 'FITS') with the appropriate name. Storing information in a named source solely used by your programs is encouraged. This is better than trusting other programs not to modify values that you have set. 'MYSOURCE' is used in the following examples:

```
*  Write a new header item.
      CALL HDR_OUT( 'OUT', 'MYSOURCE', ITEM, ' ', VALUE, ISTAT )
      ...
*  Read back the value.
      CALL HDR_IN( 'IN', 'MYSOURCE', ITEM, 1, VALUE, ISTAT )
```

[3]The Starlink application packages CCDPACK (SUN/139) and POLPACK (SUN/223) make extensive use of these facilities.

4.2.3 Accessing header items by index

There are two HDR routines that allow you to access header items using an index number. Header item index numbers run from 1 through to the maximum number of items available. These facilities can be used to list all the header items, or to test for the existence of items with particular names.

The following program snippet shows how to list all the header items associated with an image. It uses the HDR_NUMB subroutine to query the number of items. The complete example is called `hdrlist.f` – see §2.

```
*  See how many header items are available.
      CALL HDR_NUMB( 'IN', ' ', '*', N, ISTAT )            [1]
      DO 1 I = 1, N

*  Get the name of the I'th header item.
         CALL HDR_NAME( 'IN', ' ', I, ITEM, ISTAT )        [2]

*  Get its value.
         CALL HDR_IN( 'IN', ' ', ITEM, 1, VALUE, ISTAT )   [3]

*  And write it out.
         WRITE( *, '( 1X, 3A )' ) ITEM ,' = ', VALUE       [4]
 1    CONTINUE
```

The following notes refer to the numbered statements:

(1) The subroutine HDR_NUMB counts the number of header items or the number of occurrences of an item. The `'*'` argument indicates that all the header items are to be counted. If an item name was given the number of occurrences of that item would be returned (zero if none exist).

(2) Item names may be queried by using an index number. This provides a method of getting all the names when the exact contents of a header item source are not known.

(3) HDR_IN returns the value of the header item as a character string.

(4) The name and value of the header item are written out.

4.2.4 Special behaviour of FITS headers

A complication that we have glossed over so far is the fact that certain FITS header items can occur more than once within an image. Strictly, this is limited to the keywords COMMENT, HISTORY and `' '` (blank) which are often used to construct multi-line descriptions of the contents or history of a FITS file. To handle this, various of the HDR subroutines are "FITS occurrence aware". For instance, the fourth argument of the HDR_IN subroutine specifies the occurrence of the item to be read, and HDR_OUT will append (rather than overwriting) items with these special names.

FITS items can also have a comment associated with them (to be read by the recipient when a FITS file is sent to another astronomical institution). The comment string is given by the fourth argument of the HDR_OUT subroutine, but this facility isn't available for non-FITS header sources where this argument will simply be ignored.

4.2.5 Hierarchical header items

Header items can be stored in hierarchical structures.[4] What this actually means is that a header item can be used as a "container" to store other items (which are known as components). Hierarchical items are distinguished by using a period sign in the name to separate their various components. So, for instance, if you had a series of values that described a useful region on an image you might write these out using something like:

```
*  Store the useful region of this image.
      CALL HDR_OUTI( 'OUT', 'MYSOURCE', 'USEFUL.MINX', ' ',IXLOW, ISTAT )
      CALL HDR_OUTI( 'OUT', 'MYSOURCE', 'USEFUL.MAXX', ' ',IXHIGH, ISTAT )
      CALL HDR_OUTI( 'OUT', 'MYSOURCE', 'USEFUL.MINY', ' ',IYLOW, ISTAT )
      CALL HDR_OUTI( 'OUT', 'MYSOURCE', 'USEFUL.MAXY', ' ',IYHIGH, ISTAT )
```

The number of sub-components and the level of hierarchy isn't restricted, so more elaborate structures are possible (but not necessarily desirable).

You can examine the header items in an image by using the HDSTRACE program (SUN/102). If your image file is called `image` then you would use the command:

```
%  hdstrace image.more.mysource
```

to see the contents of the source `'MYSOURCE'`. To list the contents of a FITS source the most convenient method is to use the KAPPA program FITSLIST (SUN/95).

If you have any hierarchical FITS headers (these were once used extensively by UK observatories), then they should look something like:

```
ING DETHEAD = 'CCD-RCA2'       /Type of detector head
```

when examined by FITSLIST. To access this item using IMG you should call it `'ING.DETHEAD'`. The value of this item is shown being read by the example program `hdrread.f` (see §2) next:

```
% hdrread
ITEM - Header item > ING.DETHEAD
IN - Input image > image
The header item 'ING.DETHEAD' has a value of CCD-RCA2.
%
```

Hierarchical items are processed as normal by the HDR indexing routines (§4.2.3). Each component item counts only once and no significance is attached to the fact it may be part of a hierarchy. When retrieving an indexed header item, its name is returned in full, including all the necessary periods.

5 USING THE NDF LIBRARY TO DO MORE

IMG is really just a simplified interface to the NDF library (SUN/33), so all the facilities of the NDF format and library can be used along with IMG (and HDR) routines, if required. The following outlines some of the more important facilities this provides.

[4] *Warning:* you should not write hierarchical FITS header items as this violates the FITS standard (although data from outside sources may sometimes contain them).

5.1 Using regions and slices of images

The NDF library allows you to specify that a rectangular region or slice of an image should be used, rather than the whole. You identify the part you want by appending a comma separated list of ranges, in parentheses, to the name of the image. For instance if you wanted to access a square region of an image, you might use (the program running here is the `mean` example from §2.1):

```
% mean
IN - Input image > image(100:200,100:200)
```

and the square that covers the region from `100` to `200` pixels in both dimensions will be used. The rest of the image is ignored. Another way that you could get a similar effect would be to use:

```
% mean
IN - Input image > image(155~100,375~100)
```

which selects a square of side `100` centred on `155,375`.

Missing out a "range" results in the whole of that dimension being used:

```
% mean
IN - Input image > image(,100:200)
```

This uses a rectangle that extends from `100` to `200` in the second dimension and that spans the whole of the first dimension.

If you had a data cube and you wanted to process a plane from it, you might use:

```
IN - Input image > cube(,,10)
```

This would use the 2-D image stored in the tenth plane of the cube.

If the region of the image that you specify doesn't exist, then the program will still be supplied with an image array of the requested size, but the "non-existent" parts will be set to the bad value (see §4.1.6). A complete description of how to use image "sections" (as they are called by the NDF library) is given in SUN/33.

5.2 Accessing foreign data formats

IMG can be used to access data in formats other than NDF by using the NDF library's 'on-the-fly' data conversion capabilities. If you have the CONVERT package (SUN/55) available on your system, you can use it to give your IMG programs access to several important additional astronomical data formats (including IRAF, disk FITS and FIGARO). All you need to do is issue the package startup command:

```
% convert
```

before running your IMG program. It will then be able to access these other formats in the same way as its "native" NDF format.

If you have copies of the same image in more than one format, you may need to add the appropriate file type extension – `.imh`, `.fit` and `.dst` for the IRAF, FITS and FIGARO formats respectively – as part of the image name, in order to distinguish them. So a typical session in which an IRAF data frame is read might be (the program is the `mean` example from §2.1):

```
% convert
% mean
IN - Input image > myimage.imh
Mean = 108.3154
%
```

One point of particular relevance to IMG is the transfer of header information between the different data formats. Unless you take special action, only FITS headers will generally be transferred. So, unless you intend to make sole use of the NDF format, having sources of header information other than FITS may cause problems and is probably best avoided for the sake of simplicity.

If you want to process data in formats other than those provided by CONVERT, then you can define your own conversions. You should consult SSN/20 about how to do this.

5.3 Using the NDF library directly

When all else fails, the full power of the NDF library (SUN/33) can be made available using the call:

```
CALL IMG_INDF( 'IN', ID, ISTAT )
```

This returns an NDF identifier (`ID`) to your image dataset (this shouldn't be confused with the pointers to images which we have used so far). This allows you to get at the other components of the NDF (our "images" are really the main NDF data array) such as its variance, quality or world coordinate system. It also allows you to access NDF extensions (our "sources" of header information) in more sophisticated ways. As a very simple example the next program snippet shows how to write out the name of an image dataset (also see IMG_NAME):

```
CALL IMG_INDF( 'IN', ID, ISTAT )                        [1]
CALL NDF_MSG( 'NAME', ID )                              [2]
CALL MSG_OUT( ' ', 'Name of image = ^NAME', ISTAT )     [3]
CALL NDF_ANNUL( ID, ISTAT )                             [4]
```

The following notes refer to the numbered statements:

(1) `IMG_INDF` returns an NDF identifier `ID`.

(2) The NDF subroutine `NDF_MSG` sets a message token `'NAME'` (see SUN/104).

(3) The name is written out.

(4) The NDF identifier is annulled. It is important that you remember to do this once the identifier has been finished with.

6 COMPILING AND RUNNING PROGRAMS

Compiling and linking a simple program is very straight-forward. If your program source file is named `prog.f`, then the commands:

```
% star_dev
% alink prog.f -L/star/lib `img_link_adam`
```

will compile the source and link it against IMG (you should replace the `/star` with where your system is installed on non-Starlink machines). The final product is an executable called `prog`, that you run by typing:

```
% prog
```

(note the `%` is the shell-prompt and should not be typed, and the ‘‘ are grave signs that execute the command `img_link_adam`). If you have a program that has several modules that are compiled independently then you could use a command like:

```
% alink prog.f mysub1.o mysub2.o `img_link_adam`
```

The `star_dev` command only needs to be run once, as it creates a permanent soft link for accessing the include file `SAE_PAR`. The `alink` command is described more fully in SUN/144.

A ALPHABETICAL LIST OF FORTRAN SUBROUTINES

A.1 IMG Fortran subroutines

`IMG_CANCL( PARAM, STATUS )`
Cancel an image/parameter association

`IMG_DELET( PARAM, STATUS )`
Delete an image

`IMG_FREE( PARAM, STATUS )`
Free an image

`IMG_IN[n][x]( PARAM, NX, [NY], [NZ], IP, STATUS )`
Access an existing image for reading

`IMG_INDF( PARAM, INDF, STATUS )`
Obtain an NDF identifier for an image

`IMG_MOD[n][x]( PARAM, NX, [NY], [NZ], IP, STATUS )`
Access an image for modification

`IMG_NAME( PARAM, VALUE, STATUS )`
Return the image name

`IMG_NEW[n][x]( PARAM, NX, [NY], [NZ], IP, STATUS )`
Create a new image

`IMG_OUT[x]( PARAM1, PARAM2, IP, STATUS )`
Create an output image

`IMG_TMP[n][x]( PARAM, NX, [NY], [NZ], IP, STATUS )`
Create a temporary image

A.2 HDR Fortran subroutines

`HDR_COPY( PARAM1, XNAME1, PARAM2, XNAME2, STATUS )`
Copy header information from one image to another

`HDR_DELET( PARAM, XNAME, ITEM, COMP, STATUS )`
Delete a header item

`HDR_IN[x]( PARAM, XNAME, ITEM, COMP, VALUE, STATUS )`
Read a header item

`HDR_MOD( PARAM, STATUS )`
Open an image allowing modification of any header items

`HDR_NAME( PARAM, XNAME, N, ITEM, STATUS )`
Return a header item name

`HDR_NUMB( PARAM, XNAME, ITEM, N, STATUS )`
Return a header item count

`HDR_OUT[x]( PARAM, XNAME, ITEM, COMMEN, VALUE, STATUS )`
Write a header item

B FULL FORTRAN SUBROUTINE DESCRIPTIONS

B.1 IMG Fortran subroutines

IMG_CANCL
Cancel an image/parameter association

Description:

This subroutine may be used to free the resources associated with an image. Its behaviour is similar to that of IMG_FREE, except that it also removes the association between an image and a parameter. If the same parameter is later used to access an image, then a new image will be obtained (whereas if IMG_FREE is used the original image will be re-accessed).

Invocation:

```
CALL IMG_CANCL( PARAM, STATUS )
```

Arguments:

PARAM = CHARACTER * (*) (Given)

Parameter name, specifying either the individual image/parameter association to be cancelled, or '*', indicating that all such associations should be cancelled.

STATUS = INTEGER (Given and Returned)

The global status.

Notes:

- This subroutine attempts to execute even if STATUS is set on entry, although no further error report will be made if it subsequently fails under these circumstances.
- It is possible to cancel more than one association per call using multiple parameter names. Multiple parameter names are specified using a comma separated list of names (i.e. 'IMAGE1,IMAGE2'), or by using "wild-card" characters as part of a parameter name. In this context, a '*' will match any set of characters, while '%' will match any single character. Note that only those parameter names previously used to access images via IMG subroutines are considered as potential matches.
- Identifiers obtained using IMG_INDF are not released by this routine.

IMG_DELET
Delete an image

Description:

This subroutine deletes an image and frees any resources associated with it. It should be used as an alternative to IMG_FREE for any image which is not to be kept.

Invocation:

```
CALL IMG_DELET( PARAM, STATUS )
```

Arguments:

PARAM = CHARACTER * (*) (Given)

Parameter name specifying the image to be deleted.

STATUS = INTEGER (Given and Returned)

The global status.

Notes:

- Only output or images accessed for modification may be deleted.
- It is possible to delete more than one image per call using multiple parameter names. Multiple parameter names are specified using a comma separated list of names (i.e. 'IMAGE1,IMAGE2'). A wildcard capability is not supplied for this subroutine.

IMG_FREE
Free an image

Description:

This subroutine should be used to free the resources associated with an image when it has been finished with. This should always be done before the end of a program.

Invocation:

```
CALL IMG_FREE( PARAM, STATUS )
```

Arguments:

PARAM = CHARACTER * (*) (Given)

Parameter name, specifying either the individual image to be freed or '*', indicating that all images should be freed.

STATUS = INTEGER (Given and Returned)

The global status.

Notes:

- This subroutine attempts to execute even if STATUS is set on entry, although no further error report will be made if it subsequently fails under these circumstances.
- It is possible to free more than one image per call using multiple parameter names. Multiple parameter names are specified using a comma separated list of names (i.e. 'IMAGE1,IMAGE2'), or by using "wild-card" characters as part of a parameter name. In this context, a '*' will match any set of characters, while '%' will match any single character. Note that only those parameter names previously used to access images via IMG subroutines are considered as potential matches.
- Identifiers obtained using IMG_INDF are not released by this routine.

IMG_IN[n][x]
Access an existing image for reading

Description:

This subroutine accesses an input image for reading. It returns the size of the image and a pointer to its data.

The [n] and [x] parts of the name are optional. If used [n] indicates the number of dimensions of the data (1 to 3, the default is 2) and [x] the data type (one of R, D, I, W, UW, B or UB, the default is R). So for instance if you wanted to use 1-D data with a data type of DOUBLE PRECISION, the subroutine that you should call is IMG_IN1D.

Invocation:

```
CALL IMG_IN( PARAM, NX, NY, IP, STATUS )
CALL IMG_IN1[x]( PARAM, NX, IP, STATUS )
CALL IMG_IN2[x]( PARAM, NX, NY, IP, STATUS )
CALL IMG_IN3[x]( PARAM, NX, NY, NZ, IP, STATUS )
```

Arguments:

PARAM = CHARACTER * (*) (Given)
Parameter name (case insensitive).

NX = INTEGER (Returned)
First dimension of the image (in pixels).

NY = INTEGER (Returned)
Second dimension of the image (in pixels).

NZ = INTEGER (Returned)
Third dimension of the image (in pixels).

IP = INTEGER (Returned)
Pointer to image data.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- It is possible to access more than one image per call using multiple parameter names. Multiple parameter names are specified by supplying a comma separated list of names (i.e. 'DATA,BIAS,FLAT'). A pointer to the data of each image is then returned (in this case the IP argument should be passed as an array of size at least the number of parameter names). The advantage of obtaining a sequence of images in this manner is that the images are guaranteed to have the same shape and the same data type.

- The message which complains when "bad" (undefined) pixels are present in the input data can be stopped by following each parameter name by the character "!", i.e. 'DATA!,BIAS!,FLAT' will inhibit checking the images associated with parameters 'DATA' and 'BIAS', but will check the image 'FLAT'. "Bad" pixels have the symbolic values VAL__BAD[x] which are defined in the include file PRM_PAR.

IMG_INDF
Return an image NDF identifier

Description:

This subroutine returns an NDF identifier for an image. This identifier may be passed to subroutines from the NDF library (see SUN/33) to perform lower-level operations on the data which cannot be done using IMG subroutines.

Invocation:

```
CALL IMG_INDF( PARAM, INDF, STATUS )
```

Arguments:

PARAM = CHARACTER * (*) (Given)
Parameter name (case insensitive).

INDF = INTEGER (Returned)
NDF identifier for the image.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- The NDF identifier returned by this subroutine should be annulled (e.g. using NDF_ANNUL) when it is no longer required. The IMG system will not perform this operation itself.
- If this subroutine is called with STATUS set, then a value of NDF__NOID will be returned for the INDF argument, although no further processing will occur. The same value will also be returned if the subroutine should fail for any reason. The NDF__NOID constant is defined in the NDF_PAR include file.

IMG_MOD[n][x]
Access an image for modification

Description:

This routine provides access to an input image. It returns the size of the image and a pointer to its data. Existing values in the image may be modified.

The [n] and [x] parts of the name are optional. If used [n] indicates the number of dimensions of the data (1 to 3, the default is 2) and [x] the data type (one of R, D, I, W, UW, B or UB, the default is R). So for instance if you wanted to use 3-D data with a data type of DOUBLE PRECISION, the subroutine that you should call is IMG_MOD3D.

Invocation:

```
CALL IMG_MOD( PARAM, NX, NY, IP, STATUS )
CALL IMG_MOD1[x]( PARAM, NX, IP, STATUS )
CALL IMG_MOD2[x]( PARAM, NX, NY, IP, STATUS )
CALL IMG_MOD3[x]( PARAM, NX, NY, NZ, IP, STATUS )
```

Arguments:

PARAM = CHARACTER * (*) (Given)
Parameter name (case insensitive).

NX = INTEGER (Returned)
First dimension of the image (in pixels).

NY = INTEGER (Returned)
Second dimension of the image (in pixels).

NZ = INTEGER (Returned)
Third dimension of the image (in pixels).

IP = INTEGER (Returned)
Pointer to image data.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- Access to multiple image data can also be provided by this routine. Multiple parameter names are specified by supplying a comma separated list of names (i.e. 'DATA,BIAS,FLAT'). A pointer to the data of each image is then returned (in this case the IP argument should be passed as an array of size at least the number of parameter names). The advantage of obtaining a sequence of images in this manner is that the images are guaranteed to have the same shape and the same data type.
- The message which complains when "bad" (undefined) pixels are present in the input data can be stopped by following each parameter name by the character "!", i.e. 'DATA!,BIAS!,FLAT' will inhibit checking the images associated with parameters 'DATA' and 'BIAS', but will check the image 'FLAT'. "Bad" pixels have the symbolic value VAL__BAD[x] which is defined in the include file PRM_PAR.
- The input image is accessed in the type that you ask for regardless of its storage type. This may occasionally cause problems if these two types differ. This arises because when the image is converted to and from its storage type it is not always possible to guarantee that a valid value will result (for instance many numbers that you may store in a REAL variable cannot be stored in an INTEGER variable). When an image element cannot be converted it is assigned the appropriate bad value.

IMG_NAME
Return the image name

Description:

This subroutine returns the name of the input image as a character string.

Invocation:

```
CALL IMG_NAME( PARAM, VALUE, STATUS )
```

Arguments:

PARAM = CHARACTER * (*) (Given)

Parameter name of the image (case insensitive).

VALUE = CHARACTER * (*) (Returned)

The name of the image.

STATUS = INTEGER (Given and Returned)

The global status.

Notes:

- The character string VALUE should be made large enough to contain the full name of the image.
- This subroutine will directly access an image if an association has not already been made. Note that it will be opened for read-only access and you will not be able to write or delete any header items. If you need to be able to do this then access the image first, either with one of the IMG_MOD[n][x] subroutines (if you intend to process the image data), or the HDR_MOD subroutine.

IMG_NEW[n][x]
Create a new image

Description:

This subroutine creates a new image and returns a pointer to its data.

The [n] and [x] parts of the name are optional. If used [n] indicates the number of dimensions of the data (1 to 3, the default is 2) and [x] the data type (one of R, D, I, W, UW, B or UB, the default is R). So for instance if you wanted to use 3-D data with a data type of INTEGER, the subroutine that you should call is IMG_NEW3I.

Invocation:

```
CALL IMG_NEW( PARAM, NX, NY, IP, STATUS )
CALL IMG_NEW1[x]( PARAM, NX, IP, STATUS )
CALL IMG_NEW2[x]( PARAM, NX, NY, IP, STATUS )
CALL IMG_NEW3[x]( PARAM, NX, NY, NZ, IP, STATUS )
```

Arguments:

PARAM = CHARACTER * (*) (Given)
Parameter name (case insensitive).

NX = INTEGER (Given)
First dimension of the image (in pixels).

NY = INTEGER (Given)
Second dimension of the image (in pixels).

NZ = INTEGER (Given)
Third dimension of the image (in pixels).

IP = INTEGER (Returned)
Pointer to the image data.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- It is possible to create more than one image per call using multiple parameter names. Multiple names are specified by supplying a comma separated list (i.e. 'NEW1,NEW2'). A pointer to the data of each image is then returned (in this case the IP argument must be passed as an array of size at least the number of parameter names). An advantage of this method is that multiple images can be made using a single invocation of this subroutine.

IMG_OUT[x]
Create an output image

Description:

This subroutine creates a new output image by duplicating an input image. A pointer is returned to the output image data.

The [x] part of the name is optional. If used [x] indicates the data type (one of R, D, I, W, UW, B or UB, the default is R). So for instance if you wanted to make an output image with type REAL, the subroutine which you should call would be IMG_OUT.

Invocation:

```
CALL IMG_OUT[x]( PARAM1, PARAM2, IP, STATUS )
```

Arguments:

PARAM1 = CHARACTER * (*) (Given)
Parameter name for the input image (case insensitive).

PARAM2 = CHARACTER * (*) (Given)
Parameter name for the new output image (case insensitive).

IP = INTEGER (Returned)
Pointer to the mapped output data.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- It is possible to copy the input image to more than one output image using multiple parameter names. Multiple parameter names are specified by supplying a comma separated list of names (i.e. 'OUT1,OUT2'). A pointer to the data of each image is then returned (in this case the IP argument must be passed as an array of size at least the number of parameter names). The advantage of using this method is that multiple copies of an input image can be made using a single invocation of this subroutine. Multiple input image names are not allowed.
- The input image must have already been associated with a parameter before calling this subroutine.
- The output image is created with the storage data type of the input image and is then accessed in the type that you ask for ("[x]"). This may occasionally cause problems if these two types differ. This arises because when the output image is converted to and from its storage type it is not always possible to guarantee that a valid value will result (for instance many numbers that you may store in a REAL variable cannot be stored in an INTEGER variable). When an image element cannot be converted it is assigned the appropriate bad value.

IMG_TMP[n][x]
Create a temporary image

Description:

This subroutine creates a temporary image for use as workspace and returns a pointer to its data. The image will be deleted automatically when it is freed later (e.g. by calling IMG_FREE).

The [n] and [x] parts of the name are optional. If used [n] indicates the number of dimensions of the data (1 to 3, the default is 2) and [x] the data type (one of R, D, I, W, UW, B or UB, the default is R). So for instance if you wanted to use 3-D data with a data type of INTEGER, the subroutine that you should call is IMG_TMP3I.

Invocation:

```
CALL IMG_TMP( PARAM, NX, NY, IP, STATUS )
CALL IMG_TMP1[x]( PARAM, NX, IP, STATUS )
CALL IMG_TMP2[x]( PARAM, NX, NY, IP, STATUS )
CALL IMG_TMP3[x]( PARAM, NX, NY, NZ, IP, STATUS )
```

Arguments:

PARAM = CHARACTER * (*) (Given)
Parameter name (case insensitive).

NX = INTEGER (Given)
First dimension of the image (in pixels).

NY = INTEGER (Given)
Second dimension of the image (in pixels).

NZ = INTEGER (Given)
Third dimension of the image (in pixels).

IP = INTEGER (Returned)
Pointer to the image data.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- It is possible to create more than one temporary image per call using multiple parameter names. These are specified by supplying a comma separated list of names (i.e. 'WORK1, WORK2, WORK3'). A pointer to the data of each image is then returned (in this case the IP argument should be passed as an array of size at least the number of parameter names).

B.2 HDR Fortran subroutines

HDR_COPY
Copy header information from one image to another

Description:

This routine copies a compatible source of header information from one image to another. FITS headers may only be copied to FITS headers, other sources may be copied without restriction.

Invocation:

```
CALL HDR_COPY( PARAM1, XNAME1, PARAM2, XNAME2, STATUS )
```

Arguments:

PARAM1 = CHARACTER * (*) (Given)

Parameter name of the image containing the input source of header information (case insensitive).

XNAME1 = CHARACTER * (*) (Given)

The name of the extension to be copied ('FITS' or ' ' for FITS).

PARAM2 = CHARACTER * (*) (Given)

Parameter name of the image that you want to copy a header source into (case insensitive).

XNAME2 = CHARACTER * (*) (Given)

The name of the destination header source ('FITS' or ' ' for FITS, must be FITS if XNAME1 is FITS).

STATUS = INTEGER (Given and Returned)

The global status. If a header source or destination is FITS and the other isn't then IMG__BDEXT will be returned.

Notes:

- Modified header items associated with the input source will copied to the new image.
- This routine may be used to copy the same header source (from a single input image), to more than one image at a time by using multiple parameter names for PARAM2. Multiple parameter names are provided as a comma separated list (i.e. 'OUT1,OUT2,OUT3').

HDR_DELET
Delete a header item

Description:

This subroutine deletes a header item.

Invocation:

```
CALL HDR_DELET( PARAM, XNAME, ITEM, COMP, STATUS )
```

Arguments:

PARAM = CHARACTER * (*) (Given)
Parameter name of the image (case insensitive).

XNAME = CHARACTER * (*) (Given)
Source of the header information ('FITS' or ' ' for FITS headers).

ITEM = CHARACTER * (*) (Given)
Name of the header item.

COMP = INTEGER (Given)
The component of a multiple FITS header item which is to be deleted, usually 1.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- Only header items in output images or those accessed for modification will be deleted by this routine.
- This subroutine will directly access an image if an association has not already been made.
- The FITS items 'HISTORY', 'COMMENT' and ' ' can have more than one occurrence. These can be deleted one-by-one by repetitive calls to this routine with a value of 1 for the COMP argument (since if you delete the first component what was the second component now becomes the first). A specific occurrence can be deleted by giving the correct component number. The number of occurrences may be queried using the HDR_NUMB subroutine.
- Item names from any source may be hierarchical (i.e. ING.DETHEAD deletes the FITS header item "ING DETHEAD"; BOUNDS.MAXX deletes the MAXX component of BOUNDS in a non-FITS source).
- This subroutine may be used to delete an item in the same source of more than one image dataset at a time by using multiple parameter names. Multiple parameter names are provided as a comma separated list (i.e. 'IN1,IN2,IN3').

HDR_IN[x]
Read a header item

Description:

This subroutine returns the value of a header item.

The [x] part of the name is optional. If used [x] indicates the data type (one of R, D, I, L or C, the default is C) the item is to be returned in. So for instance if you want an item with type REAL, the subroutine that you should call is HDR_INR.

Invocation:

```
CALL HDR_IN[x]( PARAM, XNAME, ITEM, COMP, VALUE, STATUS )
```

Arguments:

PARAM = CHARACTER * (*) (Given)
Parameter name of the image (case insensitive).

XNAME = CHARACTER * (*) (Given)
Source of the header information ('FITS' or ' ' for FITS headers).

ITEM = CHARACTER * (*) (Given)
Name of the header item.

COMP = INTEGER (Given)
The component of a multiple FITS header item ('HISTORY' and 'COMMENT' items often have many occurrences). The number of occurrences may be queried using the HDR_NUMB subroutine.

VALUE = ? (Given and Returned)
The value. This is unmodified if the item doesn't exist.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- This subroutine will directly access an image if an association has not already been made. Note that it will be opened for read-only access and you will not be able to write or delete any header items. If you need to be able to do this then access the image first, either with one of the IMG_MOD[n][x] subroutines (if you intend to process the image data), or the HDR_MOD subroutine.
- Item names from any source may be hierarchical (i.e. ING.DETHEAD gets the value of the FITS header "ING DETHEAD"; BOUNDS.MAXX gets the value of the MAXX component of BOUNDS in a non-FITS source).

- This subroutine may be used to read the value of an item from the same source of more than one image dataset at a time by using multiple parameter names. Multiple parameter names are provided as a comma separated list (i.e. 'IN1,IN2,IN3'). The header source specified by XNAME must exist in all images and the argument VALUE must be declared as a dimension of size at least the number of parameters in the list; if this option is used.
- If a header item is not found its associated element of the VALUE argument will remain unchanged. It is therefore important that suitable defaults are assigned to VALUE before calling this subroutine. The source must exist.

HDR_MOD
Open an image allowing modification of any header items

Description:

This subroutine opens an image and allows the header information to be modified by other HDR routines.

Invocation:

```
CALL HDR_MOD( PARAM, STATUS )
```

Arguments:

PARAM = CHARACTER * (*) (Given)

Parameter name of the image (case insensitive).

STATUS = INTEGER (Given and Returned)

The global status.

Notes:

- This subroutine may be used to obtain modification access to more than one image at a time by using multiple parameter names. Multiple parameter names are provided as a comma separated list (i.e. 'IN1,IN2,IN3').
- This subroutine should only be used in programs that only modify header items. If you intend to also modify the image data then use an IMG_MOD[n][x] subroutine.

HDR_NAME
Return a header item name

Description:

This subroutine returns the name of a header item using an index of its relative position within a named source. By incrementing index N all the names in a source may be queried.

Invocation:

```
CALL HDR_NAME( PARAM, XNAME, N, ITEM, STATUS )
```

Arguments:

PARAM = CHARACTER * (*) (Given)
Parameter name of the image (case insensitive).

XNAME = CHARACTER * (*) (Given)
Source of the header information ('FITS' or ' ' for FITS headers).

N = INTEGER (Given)
The index of the item.

ITEM = CHARACTER * (*) (Returned)
The name of the header item (blank when no item with the given index exists).

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- This subroutine will directly access an image if an association has not already been made. Note that it will be opened for read-only access and you will not be able to write or delete any header items. If you need to be able to do this then access the image first, either with one of the IMG_MOD[n][x] subroutines (if you intend to process the image data), or the HDR_MOD subroutine.
- The order in which header item names are returned may change if the source is modified (by deletion or by writing).
- The header item name will be returned in a hierarchical format if necessary.
- The total number of items from a source can be found using HDR_NUMB.

HDR_NUMB
Return a header item count

Description:

This subroutine returns the number of header items or the number of occurrences of a specific item. The numbers returned can be used as limits when indexing using HDR_NAME or when deleting/reading items with more than one occurrence (such as FITS 'HISTORY' or 'COMMENTS') or to test the existence of an item.

Invocation:

```
CALL HDR_NUMB( PARAM, XNAME, ITEM, N, STATUS )
```

Arguments:

PARAM = CHARACTER * (*) (Given)
Parameter name of the image (case insensitive).

XNAME = CHARACTER * (*) (Given)
Source of the header information ('FITS' or ' ' for FITS headers).

ITEM = CHARACTER * (*) (Given)
The name of an item or '*'. If this is '*' then a count of all the items is returned, otherwise the number of occurrences of the named item is returned.

N = INTEGER (Returned)
The number of header items or occurrences of an item.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- This subroutine will directly access an image if an association has not already been made. Note that it will be opened for read-only access and you will not be able to write or delete any header items. If you need to be able to do this then access the image first, either with one of the IMG_MOD[n][x] subroutines (if you intend to process the image data), or the HDR_MOD subroutine.
- This subroutine may be used to query the number of items or occurrences of an item in the same source of more than one image dataset at a time by using multiple parameter names. Multiple parameter names are provided as a comma separated list (i.e. 'IN1,IN2,IN3'). The source must exist in all images and the argument N will be returned as the maximum number of header items or occurrences located (from each source).
- If the number of header items is zero then this will be returned, no error will be reported.

- The number of occurrences of an item will usually be 1, except for the special FITS items 'COMMENT', 'HISTORY' and ' '. This ability is therefore most useful for testing the existence of items (0 is returned if an item isn't found) or for manipulation of the occurrences of the special FITS items.

HDR_OUT[x]
Write a header item

Description:

This subroutine writes a header item.

The [x] part of the name is optional. If used [x] indicates the data type (one of R, D, I, L or C, the default is C) of the item. So for instance if you want an item with type REAL, the subroutine that you should call is HDR_OUTR.

Invocation:

```
CALL HDR_OUT[x]( PARAM, XNAME, ITEM, COMMEN, VALUE, STATUS )
```

Arguments:

PARAM = CHARACTER * (*) (Given)
Parameter name of the image (case insensitive).

XNAME = CHARACTER * (*) (Given)
Source of the header information ('FITS' or ' ' for FITS headers).

ITEM = CHARACTER * (*) (Given)
Name of the header item.

COMMEN = CHARACTER * (*) (Given)
If XNAME is 'FITS' then this is used as a comment, otherwise it is not used.

VALUE = ? (Given)
The value.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- This subroutine will directly access an image if an association has not already been made. If an association has already been made then the image must be opened for write or modification access (by IMG_MOD[n][x] or IMG_OUT[x], if you also want to process the image data and HDR_MOD otherwise).
- Item names from any source may be hierarchical (i.e. ING.DETHEAD writes the FITS header "ING DETHEAD"; BOUNDS.MAXX the value of the MAXX component of BOUNDS in a non-FITS source). Writing hierarchical records in FITS records is strongly discouraged (as it violates the standard).
- This subroutine may be used to write the value of items in the same source of more than one image dataset at a time by using multiple parameter names. Multiple parameter names are provided as a comma separated list (i.e. 'IN1,IN2,IN3'). Note that the argument VALUE must be declared as a dimension of size at least the number of parameters in the list, if this option is used.

C USING IMG FROM C

C functions, that are equivalent to the Fortran subroutines, are available for those who prefer this language (indeed using IMG from C can seem more natural as IMG returns pointers to data). The major changes in the interface (compared with the Fortran specifications) are that the names of functions change from `img_something` or `hdr_something` to `imgSomething` and `hdrSomething` and that "strings" which are returned or can be passed as arrays also have a length argument.

C.1 IMG C functions

`void imgCancl (char *param, int *status)`
Cancel an image/parameter association

`void imgDelet (char *param, int *status)`
Delete an image

`void imgFree (char *param, int *status)`
Free an image

`void imgIn[N][X] (char *param, int *nx, [int *ny,] [int *nz,] TYPE **ip, int *status)`
Access an existing image for reading

`void imgIndf (char *param, int *indf, int *status)`
Obtain an NDF identifier for an image

`void imgMod[N][X] (char *param, int *nx, [int *ny,] [int nz,] TYPE **ip, int *status)`
Access an image for modification

`void imgName[N][X] (char *param, char *value, int value_length, int *status)`
Return the image name

`void imgNew[N][X] (char *param, int nx, [int ny,] [int nz,] TYPE **ip, int *status)`
Create a new image

`void imgOut[X] (char *param1, char *param2, TYPE **ip, int *status)`
Create an output image

`void imgTmp[N][X] (char *param, int nx, [int ny,] [int nz,] TYPE **ip, int *status)`
Create a temporary image

Where the parts in `"[]"` are optional or not always available. `[N]` indicates the number of dimensions of the data you want to access and the necessary `nx`, `ny` and `nz` arguments should be passed.

Number of dimensions `[N]`	`nx`, `ny` & `nz` required
1	`nx`
2	`nx` & `ny`
3	`nx`, `ny` & `nz`

The default number of dimensions are two (in which case do not replace `[N]` by anything).

The ordering of array dimensions is different in C to Fortran, so for instance the Fortran array `ARRAY(NX,NY,NZ)` is equivalent to the C array `ARRAY[NZ][NY][NX]`, but usually this is of little

importance since images are not of fixed sizes (so you're unlikely to hard code any array dimensions) and you're far more likely to use the data via a pointer to the first element. Since C arrays also always start at 0 (rather than 1 as is the default in Fortran) the index of an element say `[y][x]` is `nx` × `y` + `x`. In 3-D this becomes `[z][y][x]` — `nx` × `ny` × `z` + `nx` × `y` + `x`.

The data type of the image is specified by replacing the `[X]` part of the name by one of the following character codes:

Character code `[X]`	C data type (`TYPE`)	Description
`F`	`float`	
`D`	`double`	
`I`	`int`	
`S`	`short`	
`US`	`unsigned short`	
`B`	`signed char`	Signed byte
`UB`	`unsigned char`	Unsigned byte

You will also need to replace `TYPE` by the related data type. The default data type is `float` (in which case do not replace `[X]` by anything and set `TYPE` to `float`).

C.2 HDR C functions

```
void hdrCopy ( char *param1, char *xname1, char *param2, char *xname2, int *status )
```
Copy header information from one image to another

```
void hdrDelet (char *param, char *xname, char *item, int comp, int *status)
```
Delete a header item

```
void hdrIn (char *param, char *xname, char *item, int comp, char *value,
            int value_length, int *status)
```
Read a character header item

```
void hdrInC (char *param, char *xname, char *item, int comp, char *value,
            int value_length, int *status)
```
Read a character header item

```
void hdrIn[X] (char *param, char *xname, char *item, int comp, TYPE *value,
            int *status)
```
Read a header item

```
void hdrMod (char *param, int *status)
```
Open an image allowing modification of any header items

```
void hdrName (char *param, char *xname, int n, char *item, int item_length,
            int *status)
```
Return a header item name

```
void hdrNumb (char *param, char *xname, char *item, int *n, int *status)
```
Return a header item count

```
void hdrOut (char *param, char *xname, char *item, char *commen, char *value,
            int value_length, int *status)
```
Write a character header item

```
void hdrOutC (char *param, char *xname, char *item, char *commen, char *value,
           int value_length, int *status)
```
Write a character header item

```
void hdrOut[X] (char *param, char *xname, char *item, char *commen, TYPE *value,
           int *status)
```
Write a header item

Where the "[X]" indicate the data type of the header items. The possible values of [X] are:

Character code[X]	C data type (TYPE)	Description
C	char	Character string
D	double	
I	int	
L	int	Boolean
F	float	

C.3 Examples of using IMG from C

C.3.1 Accessing an existing image

This section shows a C version of the mean program from §2.1.

```
#include <stdio.h>
#include "img.h"                                                    [1]

void mean_(int *istat)                                              [2]
{
  /* Local variables: */
  float *ip, sum;
  int nx, ny, i;

  /*  Access the input image. */
  imgIn( "in", &nx, &ny, &ip, istat );

  /*  Derive the mean and write it out. */
  sum = 0.0f;
  for( i=0; i < nx*ny; i++ ) sum += ip[i];
  printf ("Mean value = %f\n", sum/(nx*ny) );

  /*  Free the input image. */
  imgFree( "in", istat );
}
```

The following notes refer to the numbered statements:

(1) The file img.h contains prototypes of all the IMG and HDR functions.

(2) As in the Fortran case this program needs to be created as a function with an int * argument. This is a replacement for the main function. Note that the function is named

the same as the eventual program *except* for the trailing underscore. The underscore is required as this function will be called from a Fortran subroutine (you should also note that this "trick" may not be the same from machine to machine, to do this portably you should use the CNF (SUN/209) macros as in the full example of `mean.c`).

As before an interface file - `mean.ifl` - is also required to complete the program.

C.3.2 Creating a new image

This example (`flat.c`) shows how to create a new image. It also sets all the elements of the image to 1.0.

```
#include <stdio.h>
#include "img.h"

void flat_(int *istat)
{


  /*  Local variables: */
  float *ip;
  int i;

  /*  Create a new image. */
  imgNew( "out", 416, 578, &ip, istat );

  /*  Set all its elements to the value 1.0. */
  for( i=0; i < 416*578; i++ ) {
    ip[i] = 1.0f;
  }

  /*  Free the new image. */
  imgFree( "out", istat );
}
```

C.3.3 Modifying an image

This example shows how to access an existing image and modify its values.

```
  /* Access an existing image. */
  imgMod( "in", &nx, &ny, &ip, istat );

  /* Set all elements with a value above 32000 to 32000. */
  for( i=0; i <nx*ny; i++ ) {
      if ( ip[i] > 32000.0f ) {
         ip[i] = 32000.0f;
      }
  }

  /* Free the image */
  imgFree( "in", istat );
```

C.3.4 Modifying a copy of an image

This example (`add.c`) copies the input image and then modifies the copy.

```
/*  Access an existing image */
imgIn( "in", &nx, &ny, &ptrIn, status );

/*  Copy this to an output image */
imgOut( "in", "out", &ptrOut, status );

/*  Get the value to add. */
printf("Value to add to image: ");
scanf( "%f", &value );

/*  And do the work. */
for( i=0; i <nx*ny; i++ ) {
  ptrOut[i] = value + ptrIn[i];
}

/*  Free the input and output images. */
imgFree( "*", status );
```

C.3.5 Getting images as workspace

This example (`shadow.c`) shows how to create temporary images. The HDS (SUN/92) manual describes some reasons why you might consider using temporary images rather than "malloc'd" memory in your programs.

```
/*  Access the input image. */
imgIn( "in", &nx, &ny, &ipin, istat );

/*  Get a same sized temporary image as workspace. */
imgTmp( "temp", nx, ny, &iptemp, istat );

/*  Free all images (this deletes the temporary image). */
imgFree( "*", istat );
```

C.3.6 Using "images" which are not 2-dimensional

This section just shows some example calls that access spectra and data-cubes.

```
imgIn1( "spectrum", &nx, &ip, istat );
imgIn3( "cube", &nx, &ny, &nz, &ip, istat );
```

C.3.7 Accessing images using different data types

So far all the example shown access the image data assuming a type of `float`. IMG can also access data in other useful types. A list of these is shown in §C.1. Among the possible calls are:

```
double *dPtr;
imgInD( "image", &nx, &ny, &dPtr, istat );
float *fPtr;
```

```
imgInF( "image", &nx, &ny, &fPtr, istat );
short *sPtr;
imgInS( "image", &nx, &ny, &sPtr, istat );
int *iPtr;
imgInI( "image", &nx, &ny, &iPtr, istat );
```

Requirements for data types and dimensionalities can be mixed as in:

```
imgIn1D( 'spectrum', &nx, &ip, istat )
imgMod2I( 'image', &nx, &ny, &ip, istat )
imgNew3S( 'cube', &nx, &ny, &nz, ip, istat )
imgIn2F( 'image', &nx, &ny, &ip, istat )
```

C.3.8 Accessing multiple images

This example (`proc.c`) shows how IMG can access more than one image per function call:

```
/*  Declare pointers for images. */
float *ip[3], *ipproc, *ipbias, *ipflat, *ipraw ;              [1]
int nx, ny, i;

/*  Access input images. */
imgIn( "bias,flat,raw", &nx, &ny, ip, istat );                 [2]
ipbias = ip[0];
ipflat = ip[1];
ipraw = ip[2];

/*  Create a new output image by copying the raw input image. */
imgOut( "raw", "proc", &ipproc, istat );

/*  Debias and flatfield data. */
for( i=0; i <nx*ny; i++ ) {
   *ipproc++ = ( *ipraw++ - *ipbias++ ) / *ipflat++;
}

/*  Free all the images.*/
imgFree( "*", istat );
```

The following notes refer to the numbered statements:

(1) Note that the variable `ip` is declared as an array of pointers to `float`.

(2) After this call each of the elements of `ip` is set to point to a different image. The advantage of such a call is that all the images are guaranteed to be the same shape and data type.

C.3.9 Handling "bad" data

The way that you handle "bad" values in your data is similar to that used in Fortran, see §4.1.6, however, the bad data values themselves are now defined in the `img.h` file and have names to reflect the C data types.

Bad value macro	C data type
VAL__BADF	float
VAL__BADD	double
VAL__BADI	int
VAL__BADS	short
VAL__BADUS	unsigned short
VAL__BADB	signed char
VAL__BADUB	unsigned char

C.4 Examples of using HDR from C

C.4.1 Reading header items

This example (`hdrread.c`) shows how to read the value of a header item.

```
/*  Local Variables: */
char item[] = "OBSERVER";
char value[40];                                  [1]

/*  Try to read the value. */
hdrIn( "in", " ", item, 1, value, 40, istat );  [2]

/*  And write it out */
printf( "This data was taken by %s.\n", value);

/*  Free the image */
imgFree( "in", istat );
```

The following notes refer to the numbered statements:

(1) A character "string" is declared to contain the value of the header item.

(2) The size of the string is passed as an additional argument when calling `hdrIn` (this argument is not present in the Fortran version).

C.4.2 Writing header items

This example (`hdrwrite.c`) shows how to write a header item to an image.

```
/*  Write the header item. */
hdrOut( "out", " ", "OBSERVER", "The observer.", "Fred Bloggs", 0, istat ); [1]

/*  Free the image. */
imgFree( "out", istat );
```

The following notes refer to the numbered statements:

(1) Note the extra argument which is set to 0. This indicates that a single string (`"Fred Bloggs"`) has been passed. This could also have been set to the value 12, the string length. This argument is of more significance when dealing with multiple images.

C.4.3 Accessing header items using different data types

All the possible types of header items that can be accessed are described in §C.2. Among the possible calls are:

```
float bscale;
hdrInF( "in", " ", "bscale", 1, &bscale, istat );
int ibfact;
hdrInI( "in", " ", "binned", 1, &ibfact, istat );
#define TRUE 1
hdrOutL( "out", " ", "checked",  "Data checked for C/Rs", TRUE, istat );
```

C.4.4 Accessing header items by index

This example (`hdrlist.c`) shows how to access header items by index.

```
/*  Local Variables: */
char value[80], item[30];
int nitem, i;

/*  See how many items are present (this also accesses the image). */
hdrNumb( "in", " ", "*", &nitem, istat );
for( i=1; i<=nitem; i++ ) {

  /*  Get the name of the I'th header item. */
  hdrName( "in", source, i, item, 30, istat );                    [1]

  /*  Get its value and print out the item name and value. */
  hdrInC( "in", source, item, 1, value, 80, istat );              [1]
  printf( "%s = %s \n", item, value );
}
```

The following notes refer to the numbered statements:

(1) Note the extra arguments (`30` & `80`) that pass the maximum length of the output strings.

C.4.5 Reading and writing header items from/to many images

It is possible to read and write header items from/to many images in one call and some examples follow. Note that arrays of strings can only be passed using fixed string length. It is not possible to use arrays of pointers to `char` forming a ragged array. When using a character array you should pass a pointer to the first element not the actual array (as the HDR routines expect to see `char *`).

This example shows how to read the same header item from two images at the same time.

```
char RA[2][80];
char DEC[2][80];
hdrIn( "IN1,IN2", " ", "RA", 1, (char *) RA, 80, status );        [1]
hdrIn( "IN1,IN2", " ", "DEC", 1, (char *) DEC, 80, status );      [1]
printf( "The RA and DEC of IN1 are: %s, %s\n", RA[0], DEC[0] );
printf( "and the RA and DEC of IN2 are: %s, %s\n", RA[1], DEC[1] );
imgFree( "*", status );
```

The following notes refer to the numbered statements:

(1) The address to the first element of the RA and DEC arrays is passed. This could have been written as &RA[0][0] and &DEC[0][0].

This example shows how to write values to the same header item of two images.

```
char obstype[2][5] = {"BIAS","FLAT"};
hdrOut( "BIAS,FLAT", " ", "OBSTYPE", "Type of observation",
        (char *) obstype, 5, istat );
imgFree( "*", istat );
```

C.5 Compiling and linking C programs

Compiling and linking a C program is very similar to the methods described (in §6), except that you must compile the source code before the linking stage.

If your program source file is named prog.c, and it only depends on IMG, then the commands:

```
% cc -c -I/star/include prog.c
% alink prog.o -L/star/lib ‘img_link_adam‘
```

will compile the source and link the program.

D Changes in release (1.1)

This version introduces and documents the C interface.

E Changes in release (1.2)

In this release several bugs in the Linux version of IMG have been corrected. These affected the release of headers when using the IMG_CANCL routine and counting of header items by HDR_NUMB. A new routine has also been added – IMG_NAME. This returns the name of the image data file for use when producing logfile information etc.

F Changes in release (1.3)

The documentation has been updated to reinforce the fact that NDF identifiers obtained using IMG_INDF are not released by any IMG calls (you must do this for yourself using NDF_ANNUL).

A new routine for copying sources of header information (HDR_COPY) has been added.

A bug in the FITS header writer that added NULL characters after the END card has been corrected. This should have only affected output FITS headers that contained some initially empty cards.

G Changes in this release (1.3-1)

A bug that returned all images as the same size, even from different calls to `IMG_IN`, has been fixed.

FITS character strings may now be free-format (i.e. the terminating quote may be before the 20th column of a header card).