SUN/11.6

Starlink Project
Starlink User Note 11.6

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

4th December 2017

ARY

A Subroutine Library for Accessing ARRAY Data Structures Version 2.0 Reference Manual

Abstract

The ARY library is a set of routines for accessing Starlink ARRAY data structures built using the Hierarchical Data System (HDS).

Contents

1 Introduction

This is a preliminary document describing a set of routines for accessing Starlink ARRAY data structures built using the Hierarchical Data System HDS (SUN/92). Details of these structures and the design philosophy behind them can be found in SGP/38, although familiarity with that document is not necessarily required in order to use the routines described here.

This document currently lacks a descriptive section outlining the philosophy behind the use of the ARY_ routines. It is nevertheless being made available in this form because the ARY_ system constitutes an essential sub-component of the NDF_ system, which contains routines for accessing Starlink NDF (N-Dimensional Data Format) structures. These NDF_ routines are described in SUN/33.

The most likely reason for needing to use the ARY_ routines directly at present is for accessing ARRAY structures stored in NDF extensions. Since most ARY_ routines closely resemble the equivalent NDF_ routine, SUN/33 should initially form an adequate introduction to their use, in conjunction with the detailed routine descriptions contained in Appendix C of this document.

Note, version 2.0 of the ARY system is based on a complete re-write of the original Fortran system in C.

2 Bounds, Dimensions and Pixel Counts

The API for version 2.0 of the ARY system has been changed to allow very large arrays to be handled - that is, arrays that have more elements than can be counted in a single 4-byte integer. This has been achieved in different ways in the C and Fortran interfaces.

2.1 The Fortran API

Routines that have integer arguments that represent either the index of a pixel along a pixel axis or the number of elements in an array now have two APIs - one that supports use with very large arrays and a legacy API that does not (included for the benefit of old code). The “large array” and “legacy” routines have the same names except that the “large array” versions have the character “K” appended to the end. For instance, the legacy API includes ARY_DIM, which returns the number of pixels along each pixel axis of an array:

```
INTEGER STATUS
INTEGER IARY
INTEGER NDIM
INTEGER DIM( 3 )
...
CALL ARY\_DIM( IARY, 3, DIM, NDIM, STATUS )
```

The ‘large array” API includes ARY_DIMK, which behaves exactly like ARY_DIM, but returns the pixel counts in 8 byte integers:

```
INTEGER STATUS
INTEGER IARY
INTEGER NDIM
INTEGER*8 DIM( 3 )
...
CALL ARY\_DIMK( IARY, 3, DIM, NDIM, STATUS )
```

The list of routines for which "K" versions exist is as follows:

- ARY_BOUND
- ARY_DIM
- ARY_MAP
- ARY_MAPZ
- ARY_NEW
- ARY_NEWP
- ARY_OFFS
- ARY_SBND
- ARY_SECT
- ARY_SHIFT
- ARY_SIZE

2.2 The C API

The C API does not include alternate functions for the two APIS. Instead, all arguments that refer to pixel indices or counts are declared with a special data type of "hdsdim" (defined in include file "star/hds.h"). The particular form of integer corresponding to "hdsdim" is determiend by the version fo the HDS library with which the application is linked. Local variables used to store pixel indices or counts should also be declared with type "hdsdim":

```
Ary *ary;
int status;
int ndim;
hdsdim dim[ 3 ];
...
aryDim( ary, 3, dim, ndim, &status );
```

3 Array Storage Forms

Note that at present, the ARY_ system provides full support only for the "primitive" and "simple" forms of the ARRAY data structure.

Some support is also provided for two additional forms:

SCALED - the "scaled" form described in SGP/38. This form is the same as the "simple" form except that two extra scalar values are included that describe a linear scaling from the stored array values to the data values of interest to an external user. These two scalars are referred to as SCALE and ZERO. The external (unscaled) data values are derived from the stored (scaled) data values as follows:

unscaled = SCALE*scaled + ZERO

DELTA - this form is not currently described in SGP/38. Delta form provides a lossless compression scheme designed for arrays of integers in which there is at least one pixel axis along which the array value changes only slowly. For further details, see §3.1.

The following points should be noted:

(1) Scaled and delta arrays are "read-only". An error will be reported if an attempt is made to map a scaled or delta array for WRITE or UPDATE access. When mapped for READ access, the pointer returned by ARY_MAP provides access to the *original* data values - that is, the mapped values are the result of (for scaled arrays) applying the scale and zero terms to the stored values, or (for delta arrays) uncompressing the compressed values.

Currently, the internal stored (i.e. scaled or compressed) data values cannot be accessed directly.

(2) The result of copying a scaled or delta array (using ARY_COPY) will be an equivalent simple array.

(3) Scaled and delta arrays cannot be created directly. Instead, a simple array must first be created (using ARY_NEW), and this can then be converted to a scaled or delta array as follows:

SCALED - storing scale and zero values in the simple array using ARY_PTSZ<T>. A typical program would create a simple array, map it for write access, store the scaled data values in the mapped simple array, unmap the array, and then associate scale and zero values with the array, thus converting it to a scaled array.

DELTA - copying the simple array using ARY_DELTA. The copy will be a compressed array stored in delta form. A typical program would create a simple array, map it for write access, store the uncompressed data values in the mapped simple array, unmap the array, and then copy it using ARY_DELTA. The original simple array could then be deleted if it is no longer needed.

(4) Scaled and delta arrays cannot have complex data types. An error will be reported if an attempt is made to to import an HDS structure describing a complex scaled or delta array, or to use ARY_PTSZ<T> or ARY_DELTA on an array with complex data values.

(5) When applied to a scaled or delta array, the ARY_TYPE and ARY_FTYPE routines return the data type of the external (i.e. unscaled or uncompressed) values. In practice, this means that for a scaled array they return the data type of the SCALE and ZERO constants, rather than the data type of the array holding the stored (scaled) data values. For a delta array they return the data type of the original uncompressed values.

3.1 Delta Compressed Array Form

The DELTA storage form provides lossless compression for integer arrays. It uses two methods to achieve compression:

- Differences between adjacent data values are stored, rather than the full data values themselves. For many forms of astronomical data, the differences between adjacent data values have a much smaller range than the data values themselves. This means that they can be represented in fewer bits. For instance, if the data values are _INTEGER, then the differences between adjacent values may fit into the range of a _WORD (-32767 to +32767) or even a _BYTE (-127 to +127). This use of a shorter data type usually provides the majority of the compression. However, it is not necessary for all differences to be small - if the difference between two adjacent data values is too large for the compressed data type, the second of the two data values will be stored explicitly using the full data type of the original uncompressed data. Obviously, the more values that need to be stored in full in this way, the lower will be the compression.

  In the above description, the term "adjacent" means "adjacent along a specified pixel axis". The pixel axis along which differences are taken is referred to as the "compression axis". It may be specified explicitly by the calling application when ARY_DELTA is called, or it may be left unspecified in which case ARY_DELTA will choose the axis that gives the best compression.

- If the uncompressed array contains runs of more than three identical values along the compression axis, then the run of identical values is replaced by a single value (stored in full, not as a difference) and a repetition count.

3.1.1 Creating a Delta Array

To create a DELTA array, first store the uncompressed integer values in a simple array, and then copy the array using ARY_DELTA. The copy produced by ARY_DELTA will be stored in DELTA form.

Arrays of floating point values may be compressed by first storing the floating point values in a SCALED array, and then using ARY_DELTA to create a delta compressed copy of the scaled array. Note, the scaled array must use an integer data type to store the internal (i.e. scaled) values. The use of the scaled array means that the compression is not lossless, since some information will have been lost in scaling the floating point values into integers.

3.1.2 The HDS Structure of a Delta Array

The HDS structure of a DELTA array is similar to the SIMPLE array, in that it will contain VARIANT, DATA and ORIGIN components. In addition they can contain SCALE and ZERO terms, which, if present, are used to scale the uncompressed integers as in a SCALED array. Uncompression happens first, producing an array of uncompressed integers, which are then unscaled if required using SCALE and ZERO to produce the final uncompressed, unscaled, array.

DELTA arrays cannot be used to hold complex values and so no IMAGINARY_DATA component will be present. Also, DELTA arrays have an implicit value of .TRUE. for their bad pixel flags, and so no BAD_PIXEL component will be present in the HDS structure.

Information is stored within a DELTA array that allows sub-sections of the compressed array to be uncompressed without needing to uncompress the whole array.

A DELTA array is stored in an HDS structure with type DELTA_ARRAY, and contains the following components:

DATA - This is a one-dimensional integer array holding the differences between adjacent uncompressed integer data values along the compression axis. Its data type will be eother _INTEGER, _WORD or _BYTE and is specified when ARY_DELTA is called to create the array. A few integer values (all near the maximum value allowed by the data type) are reserved for use as flags to indicate one of the following conditions (where "MAX" represents the largest positive integer value that can be represented using the data type of the DATA array):

- The value MAX is reserved to indicate that the next element of the uncompressed array is good, but could not be expressed as a difference from the previous element because the difference would not fit into the available data range of the DATA array. Instead, the full uncompressed value is stored in the next element of the VALUE array.
- The value (MAX-1) is reserved to indicate that the next element of the uncompressed array is good and is exactly equal to the following (N-1) elements. The full uncompressed value is stored in the next element of the VALUE array. The value of N is stored in the next element of the REPEAT array.
- The value (MAX-2) is reserved to indicate that the next element of the uncompressed array is bad, as are the following (N-1) elements. The full uncompressed value of the next good value following the bad values is stored in the next element of the VALUE array. The value of N is stored in the next element of the REPEAT array.
- The value (MAX-3) is reserved to indicate that the next element of the uncompressed array is bad, but the following element is good and its full uncompressed value is stored in the next element of the VALUE array.
- The value (MAX-4) is reserved to indicate that the next N elements of the uncompressed array are good but cannot be expressed as differences from the previous element because the differences would not fit into the available data range of the DATA array. Instead, the full uncompressed values are stored in the next N elements of the VALUE array. The value of N is stored in the next element of the REPEAT array.
- Any other value is taken to be (NEXT - PREVIOUS) - the difference between the next uncompressed value and the previous uncompressed value.

Notes:

(1) The "available data range" in DATA is reduced to leave room for the above flags.

(2) The first element in each row of pixels parallel to the compression axis is always represented using one of these flag values. This allows each row of pixel values to be uncompressed without reference to any earlier values.

(3) Repeated runs of good or bad value are always contained within a single row of pixels parallel to the compression axis. Runs of repeated values that cross the boundary between adjacent rows are split into two repeated runs - one for each row.

FIRST_DATA - This is an _INTEGER array with (NDIM-1) axes which have the same order and size as the axes of the uncompressed array, but omitting the compression axis (NDIM is the number of axes in the uncompressed array). It holds the zero-based index into the DATA array at which the first element of the corresponding row of values is stored.

For instance, if the uncompressed array is a cube with bounds (1:10,1:5,1:7), and the compression axis is axis number 2, then the FIRST_DATA array will be two-dimensional with bounds (1:10,1:7). Element (2,3) of this array (for instance) will hold the integer index of the DATA array element that gives the full value for pixel (2,1,3) in the uncompressed array. Elements (2,2,3), (2,3,3), (2,4,3) and (2,5,3) of the uncompressed array are then derived from the following values in the DATA array.

FIRST_REPEAT - This is an array with the same shape as the FIRST_DATA array. It holds the zero-based index of the first value of the REPEAT array to be used whilst uncompressing the corresponding row of pixels. This component will only be present in the DELTA_ARRAY structure if the REPEAT component is present. The data type of this array will be one of _INTEGER, _UWORD or _UBYTE, depending on the largest value stored in it.

FIRST_VALUE - This is an array with the same shape as the FIRST_DATA array. It holds the zero-based index of the first value of the VALUE array to be used whilst uncompressing the corresponding row of pixels. The data type of this array will be one of _INTEGER, _UWORD or _UBYTE, depending on the largest value stored in it.

ORIGIN - A one-dimensional _INTEGER array holding the pixel indices of the first element of the uncompressed array. This component is optional - an origin of (1,1,1...) is assumed if the component is not present in the DELTA_ARRAY structure.

REPEAT - A one-dimensional _INTEGER array holding the number of repetitions for each value associated with an occurrence of (MAX-1), (MAX-2) or (MAX-4) in the DATA array. The data type of this array will be one of _INTEGER, _UWORD or _UBYTE, depending on the largest value stored in it. This array will not be present if there are no runs in the uncompressed data array.

SCALE - An optional component giving a scale factor to apply to the uncompressed integer values. It can be of any data type. If present the uncompressed array is treated like a SCALED array. In particular, the data type of the uncompressed array will be the same as the data type of the SCALE component, if present. If not present, the data type of the uncompressed array is given by the data type of the VALUE array.

VALUE - A one-dimensional array with the same data type as the uncompressed array (_INTEGER, _WORD, _UWORD, _BYTE or _UBYTE) prior to scaling by SCALE and ZERO.

It holds full uncompressed integer values for the elements that are flagged with any of the special values listed under "DATA" above. Note, if SCALE and ZERO components are present in the DELTA array, the VALUE array holds internal scaled values, rather than external unscaled values.

VARIANT - The storage form of the array. This will always be set to "DELTA".

ZAXIS - A scalar _INTEGER value giving the index of the ecompression axis - that is, the pixel axis index within the uncompressed array along which differences were taken. Care should be taken in the choice of ZAXIS since it can affect the degree of compression achieved. If ZAXIS is not specified when compressing an array, it defaults to the axis that gives the greatest compression. Note, the ZAXIS value is one-based, not zero-based.

ZDIM - A scalar _INTEGER holding the length of the compression axis within the uncompressed array. The other dimensions of the uncompressed array are given by the shape of the FIRST_DATA array.

ZERO - An optional component giving a zero offset to add to the uncompressed integer values. It can be of any data type. If present the uncompressed array is treated like a SCALED array.

ZRATIO - A scalar _REAL holding the compression factor - that is, the ratio of the uncompressed array size to the compressed array size. This is approximate as it does not include the effects of the metadata needed to describe the extra components of a DELTA array (i.e. the space needed to hold the HDS component names, types, dimensions, etc).

4 Compiling and Linking

ADAM applications which call ARY_ routines may be linked with the link script ary_link_adam, which should be specified on the linker command line. For example, to compile and link an application called `adamprog` using the `alink` command, the following might be used:

```
% alink adamprog.f -o adamprog ‘ary_link_adam‘
```

A "stand-alone" (*i.e.* non-ADAM) version of the ARY_ system is also available and should be used by those applications which do not use ADAM facilities. This version may be obtained by specifying the options file ary_link on the linker command line. For example, to compile and link a stand-alone C application called `prog`, the following might be used:

```
% gcc  prog.c -o prog ‘ary_link‘
```

Both versions of the ARY_ system contain the same set of user-callable routines.

A Alphabetical list of Routines

ARY_ANNUL(IARY, STATUS)
Annul an array identifier

ARY_BAD(IARY, CHECK, BAD, STATUS)
Determine if an array may contain bad pixels

ARY_BASE(IARY1, IARY2, STATUS)
Obtain an identifier for a base array

ARY_BOUND(IARY, NDIMX, LBND, UBND, NDIM, STATUS)
Enquire the pixel-index bounds of an array

ARY_BOUNDK(IARY, NDIMX, LBND, UBND, NDIM, STATUS)
Enquire the pixel-index bounds of an array

ARY_CLONE(IARY1, IARY2, STATUS)
Clone an array identifier

ARY_CMPLX(IARY, CMPLX, STATUS)
Determine whether an array holds complex values

ARY_COPY(IARY1, PLACE, IARY2, STATUS)
Copy an array to a new location

ARY_DELET(IARY, STATUS)
Delete an array

ARY_DIM(IARY, NDIMX, DIM, NDIM, STATUS)
Enquire the dimension sizes of an array

ARY_DIMK(IARY, NDIMX, DIM, NDIM, STATUS)
Enquire the dimension sizes of an array

ARY_DUPE(IARY1, PLACE, IARY2, STATUS)
Duplicate an array

ARY_FIND(LOC, NAME, IARY, STATUS)
Find an array in an HDS structure and import it into the ARY_ system

ARY_FORM(IARY, FORM, STATUS)
Obtain the storage form of an array

ARY_FTYPE(IARY, FTYPE, STATUS)
Obtain the full data type of an array

ARY_GTSZx(IARY, SCALE, ZERO, STATUS)
Get the scale and zero values for a scaled array

ARY_IMPRT(LOC, IARY, STATUS)
Import an array into the ARY_ system from HDS

ARY_ISACC(IARY, ACCESS, ISACC, STATUS)
Determine whether a specified type of array access is available

ARY_ISBAS(IARY, BASE, STATUS)
Enquire if an array is a base array

ARY_ISMAP(IARY, MAPPED, STATUS)
Determine if an array is currently mapped

ARY_ISTMP(IARY, TEMP, STATUS)
Determine if an array is temporary

ARY_MAP(IARY, TYPE, MMOD, PNTR, EL, STATUS)
Obtain mapped access to an array

ARY_MAPK(IARY, TYPE, MMOD, PNTR, EL, STATUS)
Obtain mapped access to an array

ARY_MAPZ(IARY, TYPE, MMOD, RPNTR, IPNTR, EL, STATUS)
Obtain complex mapped access to an array

ARY_MAPZK(IARY, TYPE, MMOD, RPNTR, IPNTR, EL, STATUS)
Obtain complex mapped access to an array

ARY_MSG(TOKEN, IARY)
Assign the name of an array to a message token

ARY_NDIM(IARY, NDIM, STATUS)
Enquire the dimensionality of an array

ARY_NEW(FTYPE, NDIM, LBND, UBND, PLACE, IARY, STATUS)
Create a new simple array

ARY_NEWK(FTYPE, NDIM, LBND, UBND, PLACE, IARY, STATUS)
Create a new simple array

ARY_NEWP(FTYPE, NDIM, UBND, PLACE, IARY, STATUS)
Create a new primitive array

ARY_NEWPK(FTYPE, NDIM, UBND, PLACE, IARY, STATUS)
Create a new primitive array

ARY_NOACC(ACCESS, IARY, STATUS)
Disable a specified type of access to an array

ARY_OFFS(IARY1, IARY2, MXOFFS, OFFS, STATUS)
Obtain the pixel offset between two arrays

ARY_OFFSK(IARY1, IARY2, MXOFFS, OFFS, STATUS)
Obtain the pixel offset between two arrays

ARY_PLACE(LOC, NAME, PLACE, STATUS)
Obtain an array placeholder

ARY_PTSZ*x*(IARY, SCALE, ZERO, STATUS)
Set new scale and zero values for a scaled array

ARY_RESET(IARY, STATUS)
Reset an array to an undefined state

ARY_SAME(IARY1, IARY2, SAME, ISECT, STATUS)
Enquire if two arrays are part of the same base array

ARY_SBAD(BAD, IARY, STATUS)
Set the bad-pixel flag for an array

ARY_SBND(NDIM, LBND, UBND, IARY, STATUS)
Set new pixel-index bounds for an array

ARY_SBNDK(NDIM, LBND, UBND, IARY, STATUS)
Set new pixel-index bounds for an array

ARY_SCTYP(IARY, TYPE, STATUS)
Obtain the numeric type of a scaled array

ARY_SECT(IARY1, NDIM, LBND, UBND, IARY2, STATUS)
Create an array section

ARY_SECTK(IARY1, NDIM, LBND, UBND, IARY2, STATUS)
Create an array section

ARY_SHIFT(NSHIFT, SHIFT, IARY, STATUS)
Apply pixel-index shifts to an array

ARY_SHIFTK(NSHIFT, SHIFT, IARY, STATUS)
Apply pixel-index shifts to an array

ARY_SIZE(IARY, NPIX, STATUS)
Determine the size of an array

ARY_SIZEK(IARY, NPIX, STATUS)
Determine the size of an array

ARY_SSECT(IARY1, IARY2, IARY3, STATUS)
Produce a similar array section to an existing one

ARY_STATE(IARY, STATE, STATUS)
Determine the state of an array (defined or undefined)

ARY_STYPE(FTYPE, IARY, STATUS)
Set a new type for an array

ARY_TEMP(PLACE, STATUS)
Obtain a placeholder for a temporary array

ARY_TRACE(NEWFLG, OLDFLG)
Set the internal ARY_ system error-tracing flag

ARY_TYPE(IARY, TYPE, STATUS)
Obtain the numeric type of an array

ARY_UNMAP(IARY, STATUS)
Unmap an array

ARY_VALID(IARY, VALID, STATUS)
Determine whether an array identifier is valid

ARY_VERFY(IARY, STATUS)
Verify that an array's data structure is correctly constructed

B Classified list of Routines

B.1 Access to Existing Arrays

ARY_FIND(LOC, NAME, IARY, STATUS)
Find an array in an HDS structure and import it into the ARY_ system

ARY_IMPRT(LOC, IARY, STATUS)
Import an array into the ARY_ system from HDS

B.2 Enquiring Array Attributes

ARY_BAD(IARY, CHECK, BAD, STATUS)
Determine if an array may contain bad pixels

ARY_BOUND(IARY, NDIMX, LBND, UBND, NDIM, STATUS)
Enquire the pixel-index bounds of an array

ARY_CMPLX(IARY, CMPLX, STATUS)
Determine whether an array holds complex values

ARY_DIM(K)(IARY, NDIMX, DIM, NDIM, STATUS)
Enquire the dimension sizes of an array

ARY_FORM(IARY, FORM, STATUS)
Obtain the storage form of an array

ARY_FTYPE(IARY, FTYPE, STATUS)
Obtain the full data type of an array

ARY_ISACC(IARY, ACCESS, ISACC, STATUS)
Determine whether a specified type of array access is available

ARY_ISMAP(IARY, MAPPED, STATUS)
Determine if an array is currently mapped

ARY_ISBAS(IARY, BASE, STATUS)
Enquire if an array is a base array

ARY_ISTMP(IARY, TEMP, STATUS)
Determine if an array is temporary

ARY_NDIM(IARY, NDIM, STATUS)
Enquire the dimensionality of an array

ARY_OFFS(K)(IARY1, IARY2, MXOFFS, OFFS, STATUS)
Obtain the pixel offset between two arrays

ARY_SAME(IARY1, IARY2, SAME, ISECT, STATUS)
Enquire if two arrays are part of the same base array

ARY_SIZE(K)(IARY, NPIX, STATUS)
Determine the size of an array

ARY_STATE(IARY, STATE, STATUS)
Determine the state of an array (defined or undefined)

ARY_TYPE(IARY, TYPE, STATUS)
Obtain the numeric type of an array

ARY_VALID(IARY, VALID, STATUS)
Determine whether an array identifier is valid

ARY_VERFY(IARY, STATUS)
Verify that an array's data structure is correctly constructed

B.3 Creating and Deleting Arrays

ARY_DELET(IARY, STATUS)
Delete an array

ARY_DUPE(IARY1, PLACE, IARY2, STATUS)
Duplicate an array

ARY_NEW(K)(FTYPE, NDIM, LBND, UBND, PLACE, IARY, STATUS)
Create a new simple array

ARY_NEWP(K)(FTYPE, NDIM, UBND, PLACE, IARY, STATUS)
Create a new primitive array

B.4 Setting Array Attributes

ARY_NOACC(ACCESS, IARY, STATUS)
Disable a specified type of access to an array

ARY_RESET(IARY, STATUS)
Reset an array to an undefined state

ARY_SBAD(BAD, IARY, STATUS)
Set the bad-pixel flag for an array

ARY_SHIFT(K)(NSHIFT, SHIFT, IARY, STATUS)
Apply pixel-index shifts to an array

ARY_STYPE(FTYPE, IARY, STATUS)
Set a new type for an array

B.5 Access to Array Values

ARY_MAP(K)(IARY, TYPE, MMOD, PNTR, EL, STATUS)
Obtain mapped access to an array

ARY_MAPZ(K)(IARY, TYPE, MMOD, RPNTR, IPNTR, EL, STATUS)
Obtain complex mapped access to an array

ARY_UNMAP(IARY, STATUS)
Unmap an array

B.6 Creation and Control of Identifiers

ARY_ANNUL(IARY, STATUS)
Annul an array identifier

ARY_BASE(IARY1, IARY2, STATUS)
Obtain an identifier for a base array

ARY_CLONE(IARY1, IARY2, STATUS)
Clone an array identifier

ARY_SECT(K)(IARY1, NDIM, LBND, UBND, IARY2, STATUS)
Create an array section

ARY_SSECT(IARY1, IARY2, IARY3, STATUS)
Produce a similar array section to an existing one

ARY_VALID(IARY, VALID, STATUS)
Determine whether an array identifier is valid

B.7 Message System Routines

ARY_MSG(TOKEN, IARY)
Assign the name of an array to a message token

B.8 Creating Placeholders

ARY_PLACE(LOC, NAME, PLACE, STATUS)
Obtain an array placeholder

ARY_TEMP(PLACE, STATUS)
Obtain a placeholder for a temporary array

B.9 Copying Arrays

ARY_COPY(IARY1, PLACE, IARY2, STATUS)
Copy an array to a new location

ARY_DUPE(IARY1, PLACE, IARY2, STATUS)
Duplicate an array

B.10 Miscellaneous

ARY_TRACE(NEWFLG, OLDFLG)
Set the internal ARY_ system error-tracing flag

C Fortran Routine Descriptions

ARY_ANNUL
Annul an array identifier

Description:

The routine annuls the array identifier supplied so that it is no longer recognised as a valid identifier by the ARY_ routines. Any resources associated with it are released and made available for re-use. If the array is mapped for access, then it is automatically unmapped by this routine.

Invocation:

```
CALL ARY_ANNUL( IARY, STATUS )
```

Arguments:

IARY = INTEGER (Given and Returned)
The array identifier to be annulled. A value of ARY__NOID is returned (as defined in the include file ARY_PAR).

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

Notes:

- This routine 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. In particular, it will fail if the identifier supplied is not initially valid, but this will only be reported if STATUS is set to SAI__OK on entry.
- An error will result if an attempt is made to annul the last remaining identifier associated with an array which is in an undefined state (unless it is a temporary array, in which case it will be deleted at this point).

ARY_BAD
Determine if an array may contain bad pixels

Description:

The routine returns a logical value indicating whether an array may contain bad pixels for which checks must be made when its values are processed. Only if the returned value is .FALSE. can such checks be omitted. If the CHECK argument to this routine is set .TRUE., then it will perform an explicit check (if necessary) to see whether bad pixels are actually present.

Invocation:

```
CALL ARY_BAD( IARY, CHECK, BAD, STATUS )
```

Arguments:

IARY = INTEGER (Given)
Array identifier.

CHECK = LOGICAL (Given)
Whether to perform an explicit check to see if bad pixels are actually present.

BAD = LOGICAL (Returned)
Whether it is necessary to check for bad pixels when processing the array's values.

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

Notes:

- If CHECK is set .FALSE., then the returned value of BAD will indicate whether bad pixels might be present and should therefore be checked for during subsequent processing. However, even if BAD is returned .TRUE. in such circumstances, it is still possible that there may not actually be any bad pixels present (for instance, in an array section, the region of the base array accessed might happen to avoid all the bad pixels).
- If CHECK is set .TRUE., then an explicit check will be made, if necessary, to ensure that BAD is only returned .TRUE. if bad pixels are actually present.
- If the array is mapped for access through the identifier supplied, then the value of BAD will refer to the actual mapped values. It may differ from its original (unmapped) value if conversion errors occurred during the mapping process, or if an initialisation option of '/ZERO' was specified for an array which was initially undefined, or if the mapped values have subsequently been modified.
- The BAD argument will always return a value of .TRUE. if the array is in an undefined state.

ARY_BASE
Obtain an identifier for a base array

Description:

The routine returns an identifier for the base array with which an array section is associated.

Invocation:

```
CALL ARY_BASE( IARY1, IARY2, STATUS )
```

Arguments:

IARY1 = INTEGER (Given)
Identifier for an existing array section (the routine will also work if this is already a base array).

IARY2 = INTEGER (Returned)
Identifier for the base array with which the section is associated.

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

Notes:

- If this routine is called with STATUS set, then a value of ARY__NOID will be returned for the IARY2 argument, although no further processing will occur. The same value will also be returned if the routine should fail for any reason. The ARY__NOID constant is defined in the include file ARY_PAR.

ARY_BOUND
Enquire the pixel-index bounds of an array

Description:

The routine returns the lower and upper pixel-index bounds of each dimension of an array, together with the total number of dimensions.

Invocation:

```
CALL ARY_BOUND( IARY, NDIMX, LBND, UBND, NDIM, STATUS )
```

Arguments:

IARY = INTEGER (Given)
Array identifier.

NDIMX = INTEGER (Given)
Maximum number of pixel-index bounds to return (i.e. the declared size of the LBND and UBND arguments).

LBND(NDIMX) = INTEGER (Returned)
Lower pixel-index bounds for each dimension.

UBND(NDIMX) = INTEGER (Returned)
Upper pixel-index bounds for each dimension.

NDIM = INTEGER (Returned)
Total number of array dimensions.

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

Notes:

- If the array has fewer than NDIMX dimensions, then any remaining elements of the LBND and UBND arguments will be filled with 1's.
- If the array has more than NDIMX dimensions, then the NDIM argument will return the actual number of dimensions. In this case only the first NDIMX sets of bounds will be returned, and an error will result if the size of any of the remaining dimensions exceeds 1.
- The symbolic constant ARY__MXDIM may be used to declare the size of the LBND and UBND arguments so that they will be able to hold the maximum number of array bounds that this routine can return. This constant is defined in the include file ARY_PAR.

ARY_BOUNDK
Enquire the pixel-index bounds of an array

Description:

The routine returns the lower and upper pixel-index bounds of each dimension of an array, together with the total number of dimensions.

Invocation:

```
CALL ARY_BOUNDK( IARY, NDIMX, LBND, UBND, NDIM, STATUS )
```

Arguments:

IARY = INTEGER (Given)
Array identifier.

NDIMX = INTEGER (Given)
Maximum number of pixel-index bounds to return (i.e. the declared size of the LBND and UBND arguments).

LBND(NDIMX) = INTEGER*8 (Returned)
Lower pixel-index bounds for each dimension.

UBND(NDIMX) = INTEGER*8 (Returned)
Upper pixel-index bounds for each dimension.

NDIM = INTEGER (Returned)
Total number of array dimensions.

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

Notes:

- If the array has fewer than NDIMX dimensions, then any remaining elements of the LBND and UBND arguments will be filled with 1's.
- If the array has more than NDIMX dimensions, then the NDIM argument will return the actual number of dimensions. In this case only the first NDIMX sets of bounds will be returned, and an error will result if the size of any of the remaining dimensions exceeds 1.
- The symbolic constant ARY__MXDIM may be used to declare the size of the LBND and UBND arguments so that they will be able to hold the maximum number of array bounds that this routine can return. This constant is defined in the include file ARY_PAR.

ARY_CLONE
Clone an array identifier

Description:

The routine produces a "cloned" copy of an array identifier (i.e. it produces a new identifier describing an array with identical attributes to the original).

Invocation:

```
CALL ARY_CLONE( IARY1, IARY2, STATUS )
```

Arguments:

IARY1 = INTEGER (Given)
Array identifier to be cloned.

IARY2 = INTEGER (Returned)
Cloned identifier.

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

Notes:

- If this routine is called with STATUS set, then a value of ARY__NOID will be returned for the IARY2 argument, although no further processing will occur. The same value will also be returned if the routine should fail for any reason. The ARY__NOID constant is defined in the include file ARY_PAR.

ARY_CMPLX
Determine whether an array holds complex values

Description:

The routine returns a logical value indicating whether an array holds complex values.

Invocation:

```
CALL ARY_CMPLX( IARY, CMPLX, STATUS )
```

Arguments:

IARY = INTEGER (Given)
Array identifier.

CMPLX = LOGICAL (Returned)
Whether the array holds complex values.

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

ARY_COPY
Copy an array to a new location

Description:

The routine copies an array to a new location and returns an identifier for the resulting new base array.

Invocation:

```
CALL ARY_COPY( IARY1, PLACE, IARY2, STATUS )
```

Arguments:

IARY1 = INTEGER (Given)
Identifier for the array (or array section) to be copied.

PLACE = INTEGER (Given and Returned)
An array placeholder (e.g. generated by the ARY_PLACE routine) which indicates the position in the data system where the new array will reside. The placeholder is annulled by this routine, and a value of ARY__NOPL will be returned (as defined in the include file ARY_PAR).

IARY2 = INTEGER (Returned)
Identifier for the new array.

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

Notes:

- The result of copying a scaled or delta array will be an equivalent simple array.
- If this routine is called with STATUS set, then a value of ARY__NOID will be returned for the IARY2 argument, although no further processing will occur. The same value will also be returned if the routine should fail for any reason. In either event, the placeholder will still be annulled. The ARY__NOID constant is defined in the include file ARY_PAR.

ARY_DELET
Delete an array

Description:

The routine deletes the specified array. If this is a base array, then the associated data object is erased and all array identifiers which refer to it (or to sections derived from it) become invalid. If the array is mapped for access, then it is first unmapped. If an array section is specified, then this routine is equivalent to calling ARY_ANNUL.

Invocation:

```
CALL ARY_DELET( IARY, STATUS )
```

Arguments:

IARY = INTEGER (Given and Returned)
Identifier for the array to be deleted. A value of ARY__NOID is returned.

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

Notes:

- This routine 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.
- A value of ARY__NOID is always returned for the IARY argument, even if the routine should fail. This constant is defined in the include file ARY_PAR.

ARY_DELTA
Compress an array using delta compression

Description:

The routine creates a copy of the supplied array stored in DELTA form, which provides a lossless compression scheme for integer data. This scheme assumes that adjacent integer values in the input array tend to be close in value, and so differences between adjacent values can be represented in fewer bits than the absolute values themselves. The differences are taken along a nominated pixel axis within the supplied array (specified by argument ZAXIS).

In practice, the scheme is limited currently to representing differences between adjacent values using a HDS integer data type (specified by argyument TYPE) - that is, arbitrary bit length is not yet supported. So for instance an _INTEGER input array can be compressed by storing differences as _WORD or _BYTE values, but a _WORD input array can only be compressed by storing differences as _BYTE values.

Any input value that differs from its earlier neighbour by more than the data range of the selected data type is stored explicitly using the data type of the input array.

Further compression is achieved by replacing runs of equal input values by a single occurrence of the value with a correspsonding repetition count.

It should be noted that the degree of compression achieved is dependent on the nature of the data, and it is possible for the compressed array to occupy more space than the uncompressed array. The compression factor actually achieved is returned in argument ZRATIO (the ratio of the supplied array size to the compressed array size). A minmum allowed compression ratio may be specified via argument MINRAT. If the compression ratio is less than this value, then the returned copy is left uncompressed.

Invocation:

```
CALL ARY_DELTA( IARY1, ZAXIS, TYPE, MINRAT, PLACE, ZRATIO, IARY2, STATUS )
```

Arguments:

IARY1 = INTEGER (Given)

The input array identifier. This can be stored in any form. If it is already stored in DELTA form, it is uncompressed and then re-compressed using the supplied compression parameters. If is is stored in SCALED form, the internal integer values are compressed and the scale and zero terms are copied into the DELTA array.

ZAXIS = INTEGER (Given)

The index of the pixel axis along which differences are to be taken. If this is zero, a default value will be selected that gives the greatest compression. An error will be reported if a value less than zero or greater than the number of axes in the input array is supplied.

TYPE = CHARACTER $*$ ($*$) (Given)

The data type in which to store the differences between adjacent input values. This must be one of `'_BYTE'`, `'_WORD'` or `'_INTEGER'`. Additionally, a blank string may be supplied in which case a default value will be selected that gives the greatest compression.

MINRAT = REAL (Given)
The minimum allowed ZRATIO value. If compressing the input array results in a ZRATIO value smaller than or equal to MINRAT, then the returned array is left uncompressed. If the supplied value is zero or negative, then the array will be compressed regardless of the compression ratio.

PLACE = INTEGER (Given and Returned)
An array placeholder (e.g. generated by the ARY_PLACE routine) which indicates the position in the data system where the new array will reside. The placeholder is annulled by this routine, and a value of ARY__NOPL will be returned (as defined in the include file ARY_PAR).

ZRATIO = REAL (Returned)
The compression factor actually achieved (the ratio of the supplied array size to the compressed array size). Genuine compressions are represented by values more than 1.0, but values less than 1.0 may be returned if the input data is not suited to delta compression (i.e. if the "compression" actually expands the array storage). Note, the returned value of ZRATIO may be smaller than MINRAT, in which case the supplied array is left unchanged. The returned compression factor is approximate as it does not take into account the space occupied by the HDS metadata describing the extra components of a DELTA array (i.e. the component names, data types, dimensions, etc). This will only be significant for very small arrays.

IARY2 = INTEGER (Returned)
Identifier for the new DELTA array.

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

Notes:

- An error will be reported if the supplied array does not hold integer values. In the case of a SCALED array, the internal (scaled) values must be integers, but the external (unscaled) values can be of any data type.
- The compression axis and compressed data type actually used can be determined by passing the returned array to ARY_GTDLT.
- An error will result if the array, or any part of it, is currently mapped for access (e.g. through another identifier).
- An error will result if the array holds complex values.

ARY_DIM
Enquire the dimension sizes of an array

Description:

The routine returns the size in pixels of each dimension of an array, together with the total number of dimensions (the size of a dimension is the difference between that dimension's upper and lower pixel-index bounds + 1).

Invocation:

```
CALL ARY_DIM( IARY, NDIMX, DIM, NDIM, STATUS )
```

Arguments:

IARY = INTEGER (Given)
Array identifier.

NDIMX = INTEGER (Given)
Maximum number of dimension sizes to return (i.e. the declared size of the DIM argument).

DIM(NDIMX) = INTEGER (Returned)
Size of each dimension in pixels.

NDIM = INTEGER (Returned)
Total number of array dimensions.

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

Notes:

- If the array has fewer than NDIMX dimensions, then any remaining elements of the DIM argument will be filled with 1's.
- If the array has more than NDIMX dimensions, then the NDIM argument will return the actual number of dimensions. In this case only the first NDIMX dimension sizes will be returned, and an error will result if the size of any of the excluded dimensions exceeds 1.
- The symbolic constant ARY__MXDIM may be used to declare the size of the DIM argument so that it will be able to hold the maximum number of array dimension sizes that this routine can return. This constant is defined in the include file ARY_PAR.

ARY_DIMK
Enquire the dimension sizes of an array

Description:

The routine returns the size in pixels of each dimension of an array, together with the total number of dimensions (the size of a dimension is the difference between that dimension's upper and lower pixel-index bounds + 1).

Invocation:

```
CALL ARY_DIMK( IARY, NDIMX, DIM, NDIM, STATUS )
```

Arguments:

IARY = INTEGER (Given)
Array identifier.

NDIMX = INTEGER (Given)
Maximum number of dimension sizes to return (i.e. the declared size of the DIM argument).

DIM(NDIMX) = INTEGER*8 (Returned)
Size of each dimension in pixels.

NDIM = INTEGER (Returned)
Total number of array dimensions.

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

Notes:

- If the array has fewer than NDIMX dimensions, then any remaining elements of the DIM argument will be filled with 1's.
- If the array has more than NDIMX dimensions, then the NDIM argument will return the actual number of dimensions. In this case only the first NDIMX dimension sizes will be returned, and an error will result if the size of any of the excluded dimensions exceeds 1.
- The symbolic constant ARY__MXDIM may be used to declare the size of the DIM argument so that it will be able to hold the maximum number of array dimension sizes that this routine can return. This constant is defined in the include file ARY_PAR.

ARY_DUPE
Duplicate an array

Description:

The routine duplicates an array, creating a new base array with the same attributes as an existing array (or array section). The new array is left in an undefined state.

Invocation:

```
CALL ARY_DUPE( IARY1, PLACE, IARY2, STATUS )
```

Arguments:

IARY1 = INTEGER (Given)
Identifier for the array to be duplicated.

PLACE = INTEGER (Given and Returned)
An array placeholder (e.g. generated by the ARY_PLACE routine) which indicates the position in the data system where the new array will reside. The placeholder is annulled by this routine, and a value of ARY__NOPL will be returned (as defined in the include file ARY_PAR).

IARY2 = INTEGER (Returned)
Identifier for the new duplicate array.

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

Notes:

- Duplicating a scaled or delta array produces and equivalent simple array.
- If this routine is called with STATUS set, then a value of ARY__NOID will be returned for the IARY2 argument, although no further processing will occur. The same value will also be returned if the routine should fail for any reason. In either event, the placeholder will still be annulled. The ARY__NOID constant is defined in the include file ARY_PAR.

ARY_FIND
Find an array in an HDS structure and import it into the ARY_ system

Description:

The routine finds a named array within an HDS structure, imports it into the ARY_ system and issues an identifier for it. The imported array may then be manipulated by the ARY_ routines.

Invocation:

```
CALL ARY_FIND( LOC, NAME, IARY, STATUS )
```

Arguments:

LOC = CHARACTER $*$ ($*$) (Given)
Locator to the enclosing HDS structure.

NAME = CHARACTER $*$ ($*$) (Given)
Name of the HDS structure component to be imported.

IARY = INTEGER (Returned)
Array identifier.

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

Notes:

- If this routine is called with STATUS set, then a value of ARY__NOID will be returned for the IARY argument, although no further processing will occur. The same value will also be returned if the routine should fail for any reason. The ARY__NOID constant is defined in the include file ARY_PAR.

ARY_FORM
Obtain the storage form of an array

Description:

The routine returns the storage form of an array as an upper-case character string (e.g. 'SIMPLE').

Invocation:

```
CALL ARY_FORM( IARY, FORM, STATUS )
```

Arguments:

IARY = INTEGER (Given)
Array identifier.

FORM = CHARACTER ∗ (∗) (Returned)
Storage form of the array.

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

Notes:

- The symbolic constant ARY__SZFRM may be used for declaring the length of a character variable to hold the storage form of an array. This constant is defined in the include file ARY_PAR.
- At present, the ARY_ routines only support "primitive", "scaled", "simple" and "delta" arrays, so only the values 'PRIMITIVE', 'SCALED' 'DELTA' and 'SIMPLE' can be returned.

ARY_FTYPE
Obtain the full data type of an array

Description:

The routine returns the full data type of an array as an upper-case character string (e.g. '_REAL' or 'COMPLEX_BYTE').

Invocation:

```
CALL ARY_FTYPE( IARY, FTYPE, STATUS )
```

Arguments:

IARY = INTEGER (Given)
Array identifier.

FTYPE = CHARACTER * (*) (Returned)
Full data type of the array.

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

Notes:

- The symbolic constant ARY__SZFTP may be used for declaring the length of a character variable to hold the full data type of an array. This constant is defined in the include file ARY_PAR.
- For "Scaled" arrays, the data type returned by this function is the data type of the SCALE and ZERO terms, rather than the data type of the stored array.

ARY_GTDLT
Get the compressed axis and data type for a DELTA array

Description:

The routine returns the details of the compression used to produce an array stored in DELTA form. If the array is not stored in DELTA form, then null values are returned as listed below, but no error is reported.

A DELTA array is compressed by storing only the differences between adjacent array values along a nominated compression axis, rather than the full array values. The differences are stored using a smaller data type than the original absolute values. The compression is lossless because any differences that will not fit into the smaller data type are stored explicitly in an extra array with a larger data type. Additional compression is achieved by replacing runs of equal values by a single value and a repeat count.

Invocation:

```
CALL ARY_GTDLT( IARY, ZAXIS, ZTYPE, ZRATIO, STATUS )
```

Arguments:

IARY = INTEGER (Given)
Array identifier.

ZAXIS = INTEGER (Returned)
The index of the pixel axis along which compression occurred. The first axis has index 1. Zero is returned if the array is not stored in DELTA form.

ZTYPE = CHARACTER $*$ ($*$) (Returned)
The data type in which the differences between adjacent array values are stored. This will be one of '_BYTE', '_WORD' or '_INTEGER'. The data type of the array itself is returned if the supplid array is not stored in DELTA form.

ZRATIO = REAL (Returned)
The compression factor - the ratio of the uncompressed array size to the compressed array size. This is approximate as it does not include the effects of the metadata needed to describe the extra components of a DELTA array (i.e. the space needed to hold the component names, types, dimensions, etc). A value of 1.0 is returned if the supplid array is not stored in DELTA form.

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

ARY_GTSZB
Get the scale and zero values for an array

Description:

The routine returns the scale and zero values associated with an array. If the array is not stored in scaled form, then values of 1.0 and 0.0 are returned.

Invocation:

```
CALL ARY_GTSZB( IARY, SCALE, ZERO, STATUS )
```

Arguments:

IARY = INTEGER (Given)
Array identifier.

SCALE = BYTE (Returned)
The scaling factor.

ZERO = BYTE (Returned)
The zero offset.

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

ARY_IMPRT
Import an array into the ARY_ system from HDS

Description:

The routine imports an array into the ARY_ system from HDS and issues an identifier for it. The array may then be manipulated by the ARY_ routines.

Invocation:

```
CALL ARY_IMPRT( LOC, IARY, STATUS )
```

Arguments:

LOC = CHARACTER ∗ (∗) (Given)
HDS locator to an array structure.

IARY = INTEGER (Returned)
Array identifier.

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

Notes:

- The locator supplied as input to this routine may later be annulled without affecting the subsequent behaviour of the ARY_ system.
- If this routine is called with STATUS set, then a value of ARY__NOID will be returned for the IARY argument, although no further processing will occur. The same value will also be returned if the routine should fail for any reason. The ARY__NOID constant is defined in the include file ARY_PAR.

ARY_ISACC
Determine whether a specified type of array access is available

Description:

The routine determines whether a specified type of access to an array is available, or whether it has been disabled. If access is not available, then any attempt to access the array in this way will fail.

Invocation:

```
CALL ARY_ISACC( IARY, ACCESS, ISACC, STATUS )
```

Arguments:

IARY = INTEGER (Given)
Array identifier.

ACCESS = CHARACTER $*$ ($*$) (Given)
The type of array access required: 'BOUNDS', 'DELETE', 'SHIFT', 'TYPE' or 'WRITE' (see the Notes section for details).

ISACC = LOGICAL (Returned)
Whether the specified type of access is available.

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

Notes:

The valid access types control the following operations on the array:

- 'BOUNDS' permits the pixel-index bounds of a base array to be altered.
- 'DELETE' permits deletion of the array.
- 'SHIFT' permits pixel-index shifts to be applied to a base array.
- 'TYPE' permits the data type of the array to be altered.
- 'WRITE' permits new values to be written to the array, or the array's state to be reset.

ARY_ISBAS
Enquire if an array is a base array

Description:

The routine returns a logical value indicating whether the array whose identifier is supplied is a base array (as opposed to an array section).

Invocation:

```
CALL ARY_ISBAS( IARY, BASE, STATUS )
```

Arguments:

IARY = INTEGER (Given)
Array identifier.

BASE = LOGICAL (Returned)
Whether the array is a base array.

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

ARY_ISMAP
Determine if an array is currently mapped

Description:

The routine returns a logical value indicating whether an array is currently mapped for access through the identifier supplied.

Invocation:

```
CALL ARY_ISMAP( IARY, MAPPED, STATUS )
```

Arguments:

IARY = INTEGER (Given)
Array identifier.

MAPPED = LOGICAL (Returned)
Whether the array is mapped for access through the IARY identifier.

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

ARY_ISTMP
Determine if an array is temporary

Description:

The routine returns a logical value indicating whether the specified array is temporary. Temporary arrays are deleted once the last identifier which refers to them is annulled.

Invocation:

```
CALL ARY_ISTMP( IARY, TEMP, STATUS )
```

Arguments:

IARY = INTEGER (Given)
Array identifier.

TEMP = LOGICAL (Returned)
Whether the array is temporary.

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

ARY_LOC
Obtain an HDS locator for an array

Description:

The routine returns an HDS locator for the data object referred to by the supplied ARY identifier.

Invocation:

```
CALL ARY_LOC( IARY, LOC, STATUS )
```

Arguments:

IARY = INTEGER (Given)

Array identifier.

LOC = CHARACTER $*$ (DAT__SZLOC) (Returned)

The HDS locator. It should be annulled using DAT_ANNUL when no longer needed. A value of DAT__NOLOC will be returned if an error occurs.

STATUS = INTEGER (Given and Returned)

The global status.

ARY_MAP
Obtain mapped access to an array

Description:

The routine obtains mapped access an array, returning a pointer to the mapped values and a count of the number of elements mapped.

Invocation:

```
CALL ARY_MAP( IARY, TYPE, MMOD, PNTR, EL, STATUS )
```

Arguments:

IARY = INTEGER (Given)
Array identifier.

TYPE = CHARACTER ∗ (∗) (Given)
The numerical data type required for access (e.g. '_REAL').

MMOD = CHARACTER ∗ (∗) (Given)
The mapping mode for access to the array: 'READ', 'UPDATE' or 'WRITE', with an optional initialisation mode '/BAD' or '/ZERO' appended.

PNTR = INTEGER (Returned)
Pointer to the mapped values.

EL = INTEGER (Returned)
Number of elements mapped.

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

Notes:

- If the array is a scaled array, the returned mapped values will be the stored array values multiplied by the scale factor and shifted by the zero term.
- If the array is a delta (i.e. compressed) array, the returned mapped values will be the uncompressed array values.
- Currently, only READ access is available for scaled and compressed arrays. An error will be reported if an attempt is made to get WRITE or UPDATE access to a scaled or compressed array.

ARY_MAPK
Obtain mapped access to an array

Description:

The routine obtains mapped access an array, returning a pointer to the mapped values and a count of the number of elements mapped.

Invocation:

```
CALL ARY_MAPK( IARY, TYPE, MMOD, PNTR, EL, STATUS )
```

Arguments:

IARY = INTEGER (Given)
Array identifier.

TYPE = CHARACTER $*$ ($*$) (Given)
The numerical data type required for access (e.g. '_REAL').

MMOD = CHARACTER $*$ ($*$) (Given)
The mapping mode for access to the array: 'READ', 'UPDATE' or 'WRITE', with an optional initialisation mode '/BAD' or '/ZERO' appended.

PNTR = INTEGER (Returned)
Pointer to the mapped values.

EL = INTEGER*8 (Returned)
Number of elements mapped.

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

Notes:

- If the array is a scaled array, the returned mapped values will be the stored array values multiplied by the scale factor and shifted by the zero term.
- If the array is a delta (i.e. compressed) array, the returned mapped values will be the uncompressed array values.
- Currently, only READ access is available for scaled and compressed arrays. An error will be reported if an attempt is made to get WRITE or UPDATE access to a scaled or compressed array.

ARY_MAPZ
Obtain complex mapped access to an array

Description:

The routine obtains complex mapped access to an array, returning pointers to the real and imaginary values and a count of the number of elements mapped.

Invocation:

```
CALL ARY_MAPZ( IARY, TYPE, MMOD, RPNTR, IPNTR, EL, STATUS )
```

Arguments:

IARY = INTEGER (Given)
Array identifier.

TYPE = CHARACTER ∗ (∗) (Given)
The numerical data type required for accessing the array (e.g. '_REAL').

MMOD = CHARACTER ∗ (∗) (Given)
The mapping mode for access to the array: 'READ', 'UPDATE' or 'WRITE', with an optional initialisation mode '/BAD' or '/ZERO' appended.

RPNTR = INTEGER (Returned)
Pointer to the mapped real (i.e. non-imaginary) values.

IPNTR = INTEGER (Returned)
Pointer to the mapped imaginary values.

EL = INTEGER (Returned)
Number of elements mapped.

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

ARY_MAPZK
Obtain complex mapped access to an array

Description:

The routine obtains complex mapped access to an array, returning pointers to the real and imaginary values and a count of the number of elements mapped.

Invocation:

```
CALL ARY_MAPZK( IARY, TYPE, MMOD, RPNTR, IPNTR, EL, STATUS )
```

Arguments:

IARY = INTEGER (Given)
Array identifier.

TYPE = CHARACTER ∗ (∗) (Given)
The numerical data type required for accessing the array (e.g. '_REAL').

MMOD = CHARACTER ∗ (∗) (Given)
The mapping mode for access to the array: 'READ', 'UPDATE' or 'WRITE', with an optional initialisation mode '/BAD' or '/ZERO' appended.

RPNTR = INTEGER (Returned)
Pointer to the mapped real (i.e. non-imaginary) values.

IPNTR = INTEGER (Returned)
Pointer to the mapped imaginary values.

EL = INTEGER*8 (Returned)
Number of elements mapped.

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

ARY_MSG
Assign the name of an array to a message token

Description:

The routine assigns the name of an array to a message token (in a form which a user will understand) for use in constructing messages with the MSG_ and ERR_ routines (see SUN/104).

Invocation:

```
CALL ARY_MSG( TOKEN, IARY )
```

Arguments:

TOKEN = CHARACTER * (*) (Given)
Name of the message token.

IARY = INTEGER (Given)
Array identifier.

Notes:

- This routine has no STATUS argument and performs no error checking. If it should fail, then no assignment to the message token will be made and this will be apparent in the final message.

ARY_NDIM
Enquire the dimensionality of an array

Description:

The routine determines the number of dimensions which an array has.

Invocation:

```
CALL ARY_NDIM( IARY, NDIM, STATUS )
```

Arguments:

IARY = INTEGER (Given)
Array identifier.

NDIM = INTEGER (Returned)
Number of array dimensions.

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

ARY_NEW
Create a new simple array

Description:

The routine creates a new simple array and returns an identifier for it. The array may subsequently be manipulated with the ARY_ routines.

Invocation:

```
CALL ARY_NEW( FTYPE, NDIM, LBND, UBND, PLACE, IARY, STATUS )
```

Arguments:

FTYPE = CHARACTER $*$ ($*$) (Given)
Full data type of the array.

NDIM = INTEGER (Given)
Number of array dimensions.

LBND(NDIM) = INTEGER (Given)
Lower pixel-index bounds of the array.

UBND(NDIM) = INTEGER (Given)
Upper pixel-index bounds of the array.

PLACE = INTEGER (Given and Returned)
An array placeholder (e.g. generated by the ARY_PLACE routine) which indicates the position in the data system where the new array will reside. The placeholder is annulled by this routine, and a value of ARY__NOPL will be returned (as defined in the include file ARY_PAR).

IARY = INTEGER (Returned)
Identifier for the new array.

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

Notes:

- If this routine is called with STATUS set, then a value of ARY__NOID will be returned for the IARY argument, although no further processing will occur. The same value will also be returned if the routine should fail for any reason. In either event, the placeholder will still be annulled. The ARY__NOID constant is defined in the include file ARY_PAR.

ARY_NEWK
Create a new simple array

Description:

The routine creates a new simple array and returns an identifier for it. The array may subsequently be manipulated with the ARY_ routines.

Invocation:

```
CALL ARY_NEWK( FTYPE, NDIM, LBND, UBND, PLACE, IARY, STATUS )
```

Arguments:

FTYPE = CHARACTER $*$ ($*$) (Given)
Full data type of the array.

NDIM = INTEGER (Given)
Number of array dimensions.

LBND(NDIM) = INTEGER*8 (Given)
Lower pixel-index bounds of the array.

UBND(NDIM) = INTEGER*8 (Given)
Upper pixel-index bounds of the array.

PLACE = INTEGER (Given and Returned)
An array placeholder (e.g. generated by the ARY_PLACE routine) which indicates the position in the data system where the new array will reside. The placeholder is annulled by this routine, and a value of ARY__NOPL will be returned (as defined in the include file ARY_PAR).

IARY = INTEGER (Returned)
Identifier for the new array.

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

Notes:

- If this routine is called with STATUS set, then a value of ARY__NOID will be returned for the IARY argument, although no further processing will occur. The same value will also be returned if the routine should fail for any reason. In either event, the placeholder will still be annulled. The ARY__NOID constant is defined in the include file ARY_PAR.

ARY_NEWP
Create a new primitive array

Description:

The routine creates a new primitive array and returns an identifier for it. The array may subsequently be manipulated with the ARY_ routines.

Invocation:

```
CALL ARY_NEWP( FTYPE, NDIM, UBND, PLACE, IARY, STATUS )
```

Arguments:

FTYPE = CHARACTER ∗ (∗) (Given)
Data type of the array (e.g. '_REAL'). Note that complex types are not allowed for primitive arrays.

NDIM = INTEGER (Given)
Number of array dimensions.

UBND(NDIM) = INTEGER (Given)
Upper pixel-index bounds of the array (the lower bound of each dimension is taken to be 1).

PLACE = INTEGER (Given and Returned)
An array placeholder (e.g. generated by the ARY_PLACE routine) which indicates the position in the data system where the new array will reside. The placeholder is annulled by this routine, and a value of ARY__NOPL will be returned (as defined in the include file ARY_PAR).

IARY = INTEGER (Returned)
Identifier for the new array.

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

Notes:

- If this routine is called with STATUS set, then a value of ARY__NOID will be returned for the IARY argument, although no further processing will occur. The same value will also be returned if the routine should fail for any reason. In either event, the placeholder will still be annulled. The ARY__NOID constant is defined in the include file ARY_PAR.

ARY_NEWPK
Create a new primitive array

Description:

The routine creates a new primitive array and returns an identifier for it. The array may subsequently be manipulated with the ARY_ routines.

Invocation:

```
CALL ARY_NEWPK( FTYPE, NDIM, UBND, PLACE, IARY, STATUS )
```

Arguments:

FTYPE = CHARACTER ∗ (∗) (Given)
Data type of the array (e.g. '_REAL'). Note that complex types are not allowed for primitive arrays.

NDIM = INTEGER (Given)
Number of array dimensions.

UBND(NDIM) = INTEGER*8 (Given)
Upper pixel-index bounds of the array (the lower bound of each dimension is taken to be 1).

PLACE = INTEGER (Given and Returned)
An array placeholder (e.g. generated by the ARY_PLACE routine) which indicates the position in the data system where the new array will reside. The placeholder is annulled by this routine, and a value of ARY__NOPL will be returned (as defined in the include file ARY_PAR).

IARY = INTEGER (Returned)
Identifier for the new array.

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

Notes:

- If this routine is called with STATUS set, then a value of ARY__NOID will be returned for the IARY argument, although no further processing will occur. The same value will also be returned if the routine should fail for any reason. In either event, the placeholder will still be annulled. The ARY__NOID constant is defined in the include file ARY_PAR.

ARY_NOACC
Disable a specified type of access to an array

Description:

The routine disables the specified type of access to an array, so that any subsequent attempt to access it in that way will fail. Access restrictions imposed on an array identifier by this routine will be propagated to any new identifiers derived from it, and cannot be revoked.

Invocation:

```
CALL ARY_NOACC( ACCESS, IARY, STATUS )
```

Arguments:

ACCESS = CHARACTER $*$ ($*$) (Given)
The type of access to be disabled: 'BOUNDS', 'DELETE', 'MODIFY', 'SCALE', 'SHIFT', 'TYPE' or 'WRITE'.

IARY = INTEGER (Given)
Array identifier.

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

Notes:

Disabling each type of access imposes the following restrictions on an array:

- 'BOUNDS' prevents the pixel-index bounds of a base array from being altered.
- 'DELETE' prevents the array being deleted.
- 'MODIFY' prevents any form of modification to the array (i.e. it disables all the other access types).
- 'SCALE' prevents the scale and zero values from being changed.
- 'SHIFT' prevents pixel-index shifts from being applied to a base array.
- 'TYPE' prevents the data type of the array from being altered.
- 'WRITE' prevents new values from being written to the array, or the array's state from being reset.

ARY_OFFS
Obtain the pixel offset between two arrays

Description:

The routine returns the pixel offset for each requested dimension between two arrays. These values are the offsets which should be added to the pixel indices of the first array to obtain the indices of the corresponding pixel in the second array.

Invocation:

```
CALL ARY_OFFS( IARY1, IARY2, MXOFFS, OFFS, STATUS )
```

Arguments:

IARY1 = INTEGER (Given)
First array identifier.

IARY2 = INTEGER (Given)
Second array identifier.

MXOFFS = INTEGER (Given)
Maximum number of pixel offsets to return (i.e. the declared size of the OFFS argument).

OFFS(MXOFFS) = INTEGER (Returned)
Array of pixel offsets for each dimension.

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

Notes:

- The two array identifiers supplied need not refer to the same base array (although they may often do so). If they do not, then the offset between the pixels in each array is determined by matching the pixel indices of their respective base arrays.
- Note that non-zero pixel offsets may exist even for dimensions which exceed the dimensionality of either of the two arrays supplied. The symbolic constant ARY__MXDIM may be used to declare the size of the OFFS argument so that it will be able to hold the maximum number of non-zero offsets that this routine can return.

ARY_OFFSK
Obtain the pixel offset between two arrays

Description:

The routine returns the pixel offset for each requested dimension between two arrays. These values are the offsets which should be added to the pixel indices of the first array to obtain the indices of the corresponding pixel in the second array.

Invocation:

```
CALL ARY_OFFSK( IARY1, IARY2, MXOFFS, OFFS, STATUS )
```

Arguments:

IARY1 = INTEGER (Given)
First array identifier.

IARY2 = INTEGER (Given)
Second array identifier.

MXOFFS = INTEGER (Given)
Maximum number of pixel offsets to return (i.e. the declared size of the OFFS argument).

OFFS(MXOFFS) = INTEGER*8 (Returned)
Array of pixel offsets for each dimension.

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

Notes:

- The two array identifiers supplied need not refer to the same base array (although they may often do so). If they do not, then the offset between the pixels in each array is determined by matching the pixel indices of their respective base arrays.
- Note that non-zero pixel offsets may exist even for dimensions which exceed the dimensionality of either of the two arrays supplied. The symbolic constant ARY__MXDIM may be used to declare the size of the OFFS argument so that it will be able to hold the maximum number of non-zero offsets that this routine can return.

ARY_PLACE
Obtain an array placeholder

Description:

The routine returns an array placeholder. A placeholder is used to identify a position in the underlying data system (HDS) and may be passed to other routines (e.g. ARY_NEW) to indicate where a newly created array should be positioned.

Invocation:

```
CALL ARY_PLACE( LOC, NAME, PLACE, STATUS )
```

Arguments:

LOC = CHARACTER $*$ ($*$) (Given)
HDS locator to the structure to contain the new array.

NAME = CHARACTER $*$ ($*$) (Given)
Name of the new structure component (i.e. the array).

PLACE = INTEGER (Returned)
Array placeholder identifying the nominated position in the data system.

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

Notes:

- Placeholders are intended only for local use within an application and only a limited number of them are available simultaneously. They are always annulled as soon as they are passed to another routine to create a new array, where they are effectively exchanged for an array identifier.
- If this routine is called with STATUS set, then a value of ARY__NOPL will be returned for the PLACE argument, although no further processing will occur. The same value will also be returned if the routine should fail for any reason. The ARY__NOPL constant is defined in the include file ARY_PAR.

ARY_PTSZB
Set new scale and zero values for a scaled array

Description:

The routine sets new values for the scale and zero values associated with an array. If the array is stored in simple form, then the storage form is changed to scaled.

Invocation:

```
CALL ARY_PTSZB( IARY, SCALE, ZERO, STATUS )
```

Arguments:

IARY = INTEGER (Given)
Array identifier.

SCALE = BYTE (Given)
The new value for the scaling factor.

ZERO = BYTE (Given)
The new value for the zero offset.

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

Notes:

- This routine may only be used to change the type of a base array. If it is called with an array which is not a base array, then it will return without action. No error will result.
- An error will result if the array, or any part of it, is currently mapped for access (e.g. through another identifier).

ARY_RESET
Reset an array to an undefined state

Description:

The routine resets an array so that its values become undefined. Its use is advisable before making format changes to an array if retention of the existing values is not required (e.g. before changing its data type with the ARY_STYPE routine); this will avoid the cost of converting the existing values.

Invocation:

```
CALL ARY_RESET( IARY, STATUS )
```

Arguments:

IARY = INTEGER (Given)
Array identifier.

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

Notes:

- This routine may only be used to reset the state of a base array. If an array section is supplied, then it will return without action. No error will result.
- An array cannot be reset while it is mapped for access. This routine will fail if this is the case.

ARY_SAME
Enquire if two arrays are part of the same base array

Description:

The routine determines whether two array identifiers refer to parts of the same base array. If so, it also determines whether they intersect.

Invocation:

```
CALL ARY_SAME( IARY1, IARY2, SAME, ISECT, STATUS )
```

Arguments:

IARY1 = INTEGER (Given)
Identifier for the first array (or array section).

IARY2 = INTEGER (Given)
Identifier for the second array (or array section).

SAME = LOGICAL (Returned)
Whether the identifiers refer to parts of the same base array.

ISECT = LOGICAL (Returned)
Whether the arrays intersect.

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

Notes:

- Two arrays (or array sections) are counted as intersecting if (i) they both refer to the same base array and (ii) altering values in one of the arrays can result in the values in the other array changing in consequence.

ARY_SBAD
Set the bad-pixel flag for an array

Description:

The routine sets the value of the bad-pixel flag for an array. A call to this routine with BAD set to .TRUE. declares that the specified array may contain bad pixel values for which checks must be made by algorithms which subsequently processes its values. A call with BAD set to .FALSE. declares that there are definitely no bad values present and that subsequent checks for such values may be omitted.

Invocation:

```
CALL ARY_SBAD( BAD, IARY, STATUS )
```

Arguments:

BAD = LOGICAL (Given)
Bad-pixel flag value to be set.

IARY = INTEGER (Given)
Array identifier.

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

Notes:

- If the array is mapped for access when this routine is called, then the bad-pixel flag will be associated with the mapped values. This information will only be transferred to the actual data object when the array is unmapped (but only if it was mapped for UPDATE or WRITE access). The value transferred may be modified if conversion errors occur during the unmapping process.

ARY_SBND
Set new pixel-index bounds for an array

Description:

The routine sets new pixel-index bounds for an array (or array section). The number of array dimensions may also be changed. If a base array is specified, then a permanent change is made to the actual data object and this will be apparent through any other array identifiers which refer to it. However, if an identifier for an array section is specified, then its bounds are altered without affecting other arrays.

Invocation:

```
CALL ARY_SBND( NDIM, LBND, UBND, IARY, STATUS )
```

Arguments:

NDIM = INTEGER (Given)
New number of array dimensions.

LBND(NDIM) = INTEGER (Given)
New lower pixel-index bounds of the array.

UBND(NDIM) = INTEGER (Given)
New upper pixel-index bounds of the array,

IARY = INTEGER (Given)
Array identifier.

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

Notes:

- The bounds of an array section cannot be altered while it is mapped for access through the identifier supplied to this routine.
- The bounds of a base array cannot be altered while any part of it is mapped for access (i.e. even through another identifier).
- The array's pixel values (if defined) will be retained for those pixels which lie within both the old and new bounds. Any pixels lying outside the new bounds will be lost (and cannot later be recovered by further changes to the array's bounds). Any new pixels introduced where the new bounds extend beyond the old ones will be assigned the "bad" value, and the subsequent value of the bad-pixel flag will reflect this.
- If the bounds of a base array are to be altered and retention of the existing pixel values is not required, then a call to ARY_RESET should be made before calling this routine. This will eliminate any processing which might otherwise be needed to retain the existing values. This step is not necessary with an array section, where no processing of pixel values takes place.

ARY_SBNDK
Set new pixel-index bounds for an array

Description:

The routine sets new pixel-index bounds for an array (or array section). The number of array dimensions may also be changed. If a base array is specified, then a permanent change is made to the actual data object and this will be apparent through any other array identifiers which refer to it. However, if an identifier for an array section is specified, then its bounds are altered without affecting other arrays.

Invocation:

```
CALL ARY_SBNDK( NDIM, LBND, UBND, IARY, STATUS )
```

Arguments:

NDIM = INTEGER (Given)
New number of array dimensions.

LBND(NDIM) = INTEGER*8 (Given)
New lower pixel-index bounds of the array.

UBND(NDIM) = INTEGER*8 (Given)
New upper pixel-index bounds of the array,

IARY = INTEGER (Given)
Array identifier.

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

Notes:

- The bounds of an array section cannot be altered while it is mapped for access through the identifier supplied to this routine.
- The bounds of a base array cannot be altered while any part of it is mapped for access (i.e. even through another identifier).
- The array's pixel values (if defined) will be retained for those pixels which lie within both the old and new bounds. Any pixels lying outside the new bounds will be lost (and cannot later be recovered by further changes to the array's bounds). Any new pixels introduced where the new bounds extend beyond the old ones will be assigned the "bad" value, and the subsequent value of the bad-pixel flag will reflect this.
- If the bounds of a base array are to be altered and retention of the existing pixel values is not required, then a call to ARY_RESET should be made before calling this routine. This will eliminate any processing which might otherwise be needed to retain the existing values. This step is not necessary with an array section, where no processing of pixel values takes place.

ARY_SCTYP
Obtain the numeric type of a scaled array

Description:

The routine returns the numeric type of a scaled array as an upper-case character string (e.g. '_REAL'). The returned type describes the values stored in the array, before they are unscaled using the associated scale and zero values. Use ARY_TYPE if you need the data type of the array after it has been unscaled.

Invocation:

```
CALL ARY_SCTYP( IARY, TYPE, STATUS )
```

Arguments:

IARY = INTEGER (Given)
Array identifier.

TYPE = CHARACTER $*$ ($*$) (Returned)
Numeric type of the array.

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

Notes:

- If the array is not stored in SCALED form, then this routine returns the same type as the ARY_TYPE routine.
- The symbolic constant ARY__SZTYP may be used for declaring the length of a character variable which is to hold the numeric type of an array. This constant is defined in the include file ARY_PAR.

ARY_SECT
Create an array section

Description:

The routine creates a new array section which refers to a selected region of an existing array (or array section). The section may be larger or smaller in extent than the original array.

Invocation:

```
CALL ARY_SECT( IARY1, NDIM, LBND, UBND, IARY2, STATUS )
```

Arguments:

IARY = INTEGER (Given)
Identifier for the initial array.

NDIM = INTEGER (Given)
Number of dimensions for new section.

LBND(NDIM) = INTEGER (Given)
Lower pixel-index bounds for the new section.

UBND(NDIM) = INTEGER (Given)
Upper pixel-index bounds for the new section.

IARY2 = INTEGER (Returned)
Identifier for the new section.

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

Notes:

- The number of section dimensions need not match the number of dimensions in the initial array. Pixel-index bounds will be padded with 1's as necessary to identify the pixels to which the new section should refer.
- Note that sections which extend beyond the pixel-index bounds of the initial array will be padded with bad pixels.
- If this routine is called with STATUS set, then a value of ARY__NOID will be returned for the IARY2 argument, although no further processing will occur. The same value will also be returned if the routine should fail for any reason. The ARY__NOID constant is defined in the include file ARY_PAR.

ARY_SECTK
Create an array section

Description:

The routine creates a new array section which refers to a selected region of an existing array (or array section). The section may be larger or smaller in extent than the original array.

Invocation:

```
CALL ARY_SECTK( IARY1, NDIM, LBND, UBND, IARY2, STATUS )
```

Arguments:

IARY = INTEGER (Given)
Identifier for the initial array.

NDIM = INTEGER (Given)
Number of dimensions for new section.

LBND(NDIM) = INTEGER*8 (Given)
Lower pixel-index bounds for the new section.

UBND(NDIM) = INTEGER*8 (Given)
Upper pixel-index bounds for the new section.

IARY2 = INTEGER (Returned)
Identifier for the new section.

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

Notes:

- The number of section dimensions need not match the number of dimensions in the initial array. Pixel-index bounds will be padded with `1's` as necessary to identify the pixels to which the new section should refer.
- Note that sections which extend beyond the pixel-index bounds of the initial array will be padded with bad pixels.
- If this routine is called with STATUS set, then a value of ARY__NOID will be returned for the IARY2 argument, although no further processing will occur. The same value will also be returned if the routine should fail for any reason. The ARY__NOID constant is defined in the include file ARY_PAR.

ARY_SHIFT
Apply pixel-index shifts to an array

Description:

The routine applies pixel-index shifts to an array. An integer shift is applied to each dimension so that its pixel-index bounds, and the indices of each pixel, change by the amount of shift applied to the corresponding dimension. The array's pixels retain their values and none are lost.

Invocation:

```
CALL ARY_SHIFT( NSHIFT, SHIFT, IARY, STATUS )
```

Arguments:

NSHIFT = INTEGER (Given)
Number of dimensions to which shifts are to be applied. This must not exceed the number of array dimensions. If fewer shifts are supplied than there are dimensions in the array, then the extra dimensions will not be shifted.

SHIFT(NSHIFT) = INTEGER (Given)
The pixel-index shifts to be applied to each dimension.

IARY = INTEGER (Given)
Array identifier.

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

Notes:

- Pixel-index shifts applied to a base array will affect the appearance of that array as seen by all base-array identifiers associated with it. However, array sections derived from that base array will remain unchanged (as regards both pixel-indices and data content).
- Pixel-index shifts cannot be applied to a base array while any part of it is mapped for access (i.e. even through another identifier).
- Pixel-index shifts applied to an array section only affect that section itself, and have no effect on other array identifiers.
- Pixel-index shifts cannot be applied to an array section while it is mapped for access through the identifier supplied to this routine.

ARY_SHIFTK
Apply pixel-index shifts to an array

Description:

The routine applies pixel-index shifts to an array. An integer shift is applied to each dimension so that its pixel-index bounds, and the indices of each pixel, change by the amount of shift applied to the corresponding dimension. The array's pixels retain their values and none are lost.

Invocation:

```
CALL ARY_SHIFTK( NSHIFT, SHIFT, IARY, STATUS )
```

Arguments:

NSHIFT = INTEGER (Given)
Number of dimensions to which shifts are to be applied. This must not exceed the number of array dimensions. If fewer shifts are supplied than there are dimensions in the array, then the extra dimensions will not be shifted.

SHIFT(NSHIFT) = INTEGER*8 (Given)
The pixel-index shifts to be applied to each dimension.

IARY = INTEGER (Given)
Array identifier.

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

Notes:

- Pixel-index shifts applied to a base array will affect the appearance of that array as seen by all base-array identifiers associated with it. However, array sections derived from that base array will remain unchanged (as regards both pixel-indices and data content).
- Pixel-index shifts cannot be applied to a base array while any part of it is mapped for access (i.e. even through another identifier).
- Pixel-index shifts applied to an array section only affect that section itself, and have no effect on other array identifiers.
- Pixel-index shifts cannot be applied to an array section while it is mapped for access through the identifier supplied to this routine.

ARY_SIZE
Determine the size of an array

Description:

The routine returns the number of pixels in the array whose identifier is supplied (i.e. the product of its dimensions).

Invocation:

```
CALL ARY_SIZE( IARY, NPIX, STATUS )
```

Arguments:

IARY = INTEGER (Given)
Array identifier.

NPIX = INTEGER (Returned)
Number of pixels in the array.

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

ARY_SIZEK
Determine the size of an array

Description:

The routine returns the number of pixels in the array whose identifier is supplied (i.e. the product of its dimensions).

Invocation:

```
CALL ARY_SIZEK( IARY, NPIX, STATUS )
```

Arguments:

IARY = INTEGER (Given)
Array identifier.

NPIX = INTEGER*8 (Returned)
Number of pixels in the array.

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

ARY_SSECT
Create a similar array section to an existing one

Description:

The routine creates a new array section, using an existing section as a template. The new section bears the same relationship to its base array as the template section does to its own base array. Allowance is made for pixel-index shifts which may have been applied so that the pixel-indices of the new section match those of the template. The number of dimensions of the input and template arrays may differ.

Invocation:

```
CALL ARY_SSECT( IARY1, IARY2, IARY3, STATUS )
```

Arguments:

IARY1 = INTEGER (Given)
Identifier for the input array from which the section is to be derived. This may be a base array or an array section.

IARY2 = INTEGER (Given)
Identifier for the template section (this may also be a base array or an array section).

IARY3 = INTEGER (Returned)
Identifier for the new array section.

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

Notes:

- This routine normally generates an array section. However, if both input arrays are base arrays with identical pixel-index bounds, then there is no need to create a section in order to access the required part of the first array. In this case a base array identifier will be returned instead.
- The new section created by this routine will have the same number of dimensions as the array (or array section) from which it is derived. If the template (IARY2) array has fewer dimensions than this, then the bounds of any additional input dimensions are preserved unchanged in the new array. If the template (IARY2) array has more dimensions, then the excess ones are ignored.
- This routine takes account of the regions of each base array to which the input array sections have access. It may therefore restrict the region accessible to the new section (and pad with "bad" pixels) so as not to grant access to regions of the base array which were not previously accessible through the input arrays.

- If this routine is called with STATUS set, then a value of ARY__NOID will be returned for the IARY3 argument, although no further processing will occur. The same value will also be returned if the routine should fail for any reason. The ARY__NOID constant is defined in the include file ARY_PAR.

ARY_STATE
Determine the state of an array (defined or undefined)

Description:

The routine returns a logical value indicating whether an array's pixel values are currently defined.

Invocation:

```
CALL ARY_STATE( IARY, STATE, STATUS )
```

Arguments:

IARY = INTEGER (Given)
Array identifier.

STATE = LOGICAL (Returned)
Whether the array's pixel values are defined.

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

ARY_STYPE
Set a new type for an array

Description:

The routine sets a new full type for an array, causing its data storage type to be changed. If the array's pixel values are defined, then they will be converted from the old type to the new one. If they are undefined, then no conversion will be necessary. Subsequent enquiries will reflect the new type. Conversion may be performed between any types supported by the ARY_ routines, including from a non-complex type to a complex type (and vice versa).

Invocation:

```
CALL ARY_STYPE( FTYPE, IARY, STATUS )
```

Arguments:

FTYPE = CHARACTER $*$ ($*$) (Given)
The new full type specification for the array (e.g. '_REAL' or 'COMPLEX_INTEGER').

IARY = INTEGER (Given)
Array identifier.

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

Notes:

- This routine may only be used to change the type of a base array. If it is called with an array which is not a base array, then it will return without action. No error will result.
- An error will result if the array, or any part of it, is currently mapped for access (e.g. through another identifier).
- If the type of an array is to be changed without its pixel values being retained, then a call to ARY_RESET should be made beforehand. This will avoid the cost of converting all the values.

ARY_TEMP
Obtain a placeholder for a temporary array

Description:

The routine returns an array placeholder which may be used to create a temporary array (i.e. one which will be deleted automatically once the last identifier associated with it is annulled). The placeholder returned by this routine may be passed to other routines (e.g. ARY_NEW or ARY_COPY) to produce a temporary array in the same way as a new permanent array would be created.

Invocation:

```
CALL ARY_TEMP( PLACE, STATUS )
```

Arguments:

PLACE = INTEGER (Returned)
Placeholder for a temporary array.

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

Notes:

- Placeholders are intended only for local use within an application and only a limited number of them are available simultaneously. They are always annulled as soon as they are passed to another routine to create a new array, where they are effectively exchanged for an array identifier.
- If this routine is called with STATUS set, then a value of ARY__NOPL will be returned for the PLACE argument, although no further processing will occur. The same value will also be returned if the routine should fail for any reason. The ARY__NOPL constant is defined in the include file ARY_PAR.

ARY_TRACE
Set the internal ARY_ system error-tracing flag

Description:

The routine sets an internal flag in the ARY_ system which enables or disables error-tracing messages. If this flag is set to .TRUE., then any error occurring within the ARY_ system will be accompanied by error messages indicating which internal routines have exited prematurely as a result. If the flag is set to .FALSE., this internal diagnostic information will not appear and only standard error messages will be produced.

Invocation:

```
CALL ARY_TRACE( NEWFLG, OLDFLG )
```

Arguments:

NEWFLG = LOGICAL (Given)

The new value to be set for the error-tracing flag.

OLDFLG = LOGICAL (Returned)

The previous value of the flag.

Notes:

- By default, the error tracing flag is set to .FALSE., so no internal diagnostic information will be produced.

ARY_TYPE
Obtain the numeric type of an array

Description:

The routine returns the numeric type of an array as an upper-case character string (e.g. '_REAL').

Invocation:

```
CALL ARY_TYPE( IARY, TYPE, STATUS )
```

Arguments:

IARY = INTEGER (Given)
Array identifier.

TYPE = CHARACTER * (*) (Returned)
Numeric type of the array.

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

Notes:

- The symbolic constant ARY__SZTYP may be used for declaring the length of a character variable which is to hold the numeric type of an array. This constant is defined in the include file ARY_PAR.

ARY_UNMAP
Unmap an array

Description:

The routine unmaps an array which has previously been mapped for READ, UPDATE or WRITE access.

Invocation:

```
CALL ARY_UNMAP( IARY, STATUS )
```

Arguments:

IARY = INTEGER (Given)
Array identifier.

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

Notes:

- This routine 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.
- An error will result if the array has not previously been mapped for access.

ARY_VALID
Determine whether an array identifier is valid

Description:

Determine whether an array identifier is valid (i.e. associated with an array).

Invocation:

```
CALL ARY_VALID( IARY, VALID, STATUS )
```

Arguments:

IARY = INTEGER (Given)
Identifier to be tested.

VALID = LOGICAL (Returned)
Whether the identifier is valid.

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

ARY_VERFY
Verify that an array's data structure is correctly constructed

Description:

The routine checks that the data structure containing an array is correctly constructed and that the array's pixel values are defined. It also checks for the presence of any "rogue" components in the data structure. If an anomaly is found, then an error results. Otherwise, the routine returns without further action.

Invocation:

```
CALL ARY_VERFY( IARY, STATUS )
```

Arguments:

IARY = INTEGER (Given)
Array identifier.

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

D C Function Descriptions

The C API differs from the Fortran API in the following ways:

(1) Arrays are identified using pointers of type “Ary *” rather than integer values.

(2) Values representing pixel indices or counts are stored in variables of type “hdsdim”. This will be a 4 or 8 byte integer type, depending on the version of the installed HDS library.

(3) HDS locators are passed as pointers of type “HDSLoc *” rather than as character strings.

aryAnnul
Annul an array pointer

Description:

This function annuls the array pointer supplied so that it is no longer recognised as a valid pointer by the ARY_ routines. Any resources associated with it are released and made available for re-use. If the array is mapped for access, then it is automatically unmapped by this routine.

Invocation:

```
void aryAnnul( Ary **ary, int *status )
```

Notes:

- This routine 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. In particular, it will fail if the pointer supplied is not initially valid, but this will only be reported if ' status' is set to SAI__OK on entry.
- An error will result if an attempt is made to annul the last remaining pointer associated with an array which is in an undefined state (unless it is a temporary array, in which case it will be deleted at this point).

Parameters

ary Address of the array pointer to be annulled. A value of NULL is returned in place of the supplied pointer.

status
The global status.

aryBad
Determine if an array may contain bad pixels

Description:

This function returns a boolean value indicating whether an array may contain bad pixels for which checks must be made when its values are processed. Only if the returned value is zero can such checks be omitted. If the " check" argument to this function is set non-zero, then it will perform an explicit check (if necessary) to see whether bad pixels are actually present.

Invocation:

```
void aryBad( Ary *ary, int check, int *bad, int *status )
```

Notes:

- If " check" is set to zero, then the returned value of " bad" will indicate whether bad pixels might be present and should therefore be checked for during subsequent processing. However, even if " bad" is returned non-zero in such circumstances, it is still possible that there may not actually be any bad pixels present (for instance, in an array section, the region of the base array accessed might happen to avoid all the bad pixels).
- If " check" is set non-zero, then an explicit check will be made, if necessary, to ensure that " bad" is only returned non-zero if bad pixels are actually present.
- If the array is mapped for access through the identifier supplied, then the value of " bad" will refer to the actual mapped values. It may differ from its original (unmapped) value if conversion errors occurred during the mapping process, or if an initialisation option of ' /ZERO' was specified for an array which was initially undefined, or if the mapped values have subsequently been modified.
- The " bad" argument will always be returned holding a non-zero value if the array is in an undefined state.

Parameters

ary Array identifier.

check
If non-zero, an explicit check is performed to see if bad pixels are actually present.

bad
Returned holding a flag indicating whether it is necessary to check for bad pixels when processing the array' s values.

status
The global status.

aryBase
Obtain an identifier for a base array

Description:

This function returns an identifier for the base array with which an array section is associated.

Invocation:

```
void aryBase( Ary *ary1, Ary**ary2, int *status )
```

Notes:

- If this routine is called with " status" set, then a value of NULL will be returned for the " ary2" argument, although no further processing will occur. The same value will also be returned if the routine should fail for any reason.

Parameters

ary1

Identifier for an existing array section (the function will also work if this is already a base array).

ary2

Returned holding an identifier for the base array with which the section is associated.

status

The global status.

aryBound
Enquire the pixel-index bounds of an array

Description:

This function returns the lower and upper pixel-index bounds of each dimension of an array, together with the total number of dimensions.

Invocation:

```
void aryBound( Ary *ary, int ndimx, hdsdim *lbnd, hdsdim *ubnd, int *ndim, int
*status )
```

Notes:

- If the array has fewer than " ndimx" dimensions, then any remaining elements of the " lbnd" and " ubnd" arguments will be filled with 1' s.
- If the array has more than " ndimx" dimensions, then the " ndim" argument will return the actual number of dimensions. In this case only the first " ndimx" sets of bounds will be returned, and an error will result if the size of any of the remaining dimensions exceeds 1.
- The symbolic constant ARY__MXDIM may be used to declare the size of the " lbnd" and " ubnd" arguments so that they will be able to hold the maximum number of array bounds that this routine can return. This constant is defined in the header file " ary.h" .

Parameters

ary Array identifier.

ndimx
Maximum number of pixel-index bounds to return (i.e. the declared size of the " lbnd" and " ubnd" arguments).

lbnd
Returned holding the lower pixel-index bounds for each dimension.

ubnd
Returned holding the upper pixel-index bounds for each dimension.

ndim
Returned holding the total number of array dimensions.

status
The global status.

aryClone
Clone an array identifier

Description:

This function produces a " cloned" copy of an array identifier (i.e. it produces a new identifier describing an array with identical attributes to the original).

Invocation:

```
void aryClone( Ary *ary1, Ary **ary2, int *status )
```

Notes:

- If this routine is called with " status" set, then a value of NULL will be returned for the " ary2" argument, although no further processing will occur. The same value will also be returned if the routine should fail for any reason.

Parameters

ary1

Array identifier to be cloned.

ary2

Returned holding the cloned identifier.

status

The global status.

aryCmplx
Determine whether an array holds complex values

Description:

This function returns a logical value indicating whether an array holds complex values.

Invocation:

```
void aryCmplx( Ary *ary, int *cmplx, int *status )
```

Parameters

ary Array identifier.

cmplx
Returned holding a flag indicating whether the array holds complex values.

status
The global status.

aryCopy
Copy an array to a new location

Description:

This function copies an array to a new location and returns an identifier for the resulting new base array.

Invocation:

```
void aryCopy( Ary *ary1, AryPlace **place, Ary **ary2, int *status )
```

Arguments:

ary1

Identifier for the array (or array section) to be copied.

place

An array placeholder (e.g. generated by the aryPlace function) which indicates the position in the data system where the new array will reside. The placeholder is annulled by this function, and a value of NULL will be returned.

ary2

Returned holding the identifier for the new array.

status

The global status.

Notes:

- The result of copying a scaled or delta array will be an equivalent simple array.
- If this routine is called with " status" set, then a value of NULL will be returned for the " ary" argument, although no further processing will occur. The same value will also be returned if the function should fail for any reason. In either event, the placeholder will still be annulled.

aryDelet
Delete an array

Description:

This function deletes the specified array. If this is a base array, then the associated data object is erased and all array identifiers which refer to it (or to sections derived from it) become invalid. If the array is mapped for access, then it is first unmapped. If an array section is specified, then this function is equivalent to calling aryAnnul.

Invocation:

```
void aryDelet( Ary **ary, int *status )
```

Notes:

- This function 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.
- A value of NULL is always returned for the " ary" argument, even if the function should fail.

Parameters

ary Identifier for the array to be deleted. A value of NULL is returned.

status
The global status.

aryDelta
Compress an array using delta compression

Description:

This function creates a copy of the supplied array stored in DELTA form, which provides a lossless compression scheme for integer data. This scheme assumes that adjacent integer values in the input array tend to be close in value, and so differences between adjacent values can be represented in fewer bits than the absolute values themselves. The differences are taken along a nominated pixel axis within the supplied array (specified by argument ZAXIS).

In practice, the scheme is limited currently to representing differences between adjacent values using a HDS integer data type (specified by argyument TYPE) - that is, arbitrary bit length is not yet supported. So for instance an _INTEGER input array can be compressed by storing differences as _WORD or _BYTE values, but a _WORD input array can only be compressed by storing differences as _BYTE values.

Any input value that differs from its earlier neighbour by more than the data range of the selected data type is stored explicitly using the data type of the input array.

Further compression is achieved by replacing runs of equal input values by a single occurrence of the value with a correspsonding repetition count.

It should be noted that the degree of compression achieved is dependent on the nature of the data, and it is possible for the compressed array to occupy more space than the uncompressed array. The compression factor actually achieved is returned in argument " zratio" (the ratio of the supplied array size to the compressed array size). A minmum allowed compression ratio may be specified via argument " minrat" . If the compression ratio is less than this value, then the returned copy is left uncompressed.

Invocation:

```
void aryDelta( Ary *ary1, int zaxis, const char *type, float minrat, AryPlace
**place, float *zratio, Ary **ary2, int *status )
```

Notes:

- An error will be reported if the supplied array does not hold integer values. In the case of a SCALED array, the internal (scaled) values must be integers, but the external (unscaled) values can be of any data type.
- The compression axis and compressed data type actually used can be determined by passing the returned array to aryGtdlt.
- An error will result if the array, or any part of it, is currently mapped for access (e.g. through another identifier).
- An error will result if the array holds complex values.

Parameters

ary1

The input array identifier. This can be stored in any form. If it is already stored in DELTA form, it is uncompressed and then re-compressed using the supplied compression parameters. If is is stored in SCALED form, the internal integer values are compressed and the scale and zero terms are copied into the DELTA array.

zaxis

The one-based index of the pixel axis along which differences are to be taken. If this is zero, a default value will be selected that gives the greatest compression. An error will be reported if a value less than zero or greater than the number of axes in the input array is supplied.

type

The data type in which to store the differences between adjacent input values. This must be one of ' _BYTE' , ' _WORD' or ' _INTEGER' . Additionally, a blank string may be supplied in which case a default value will be selected that gives the greatest compression.

minrat

The minimum allowed ZRATIO value. If compressing the input array results in a ZRATIO value smaller than or equal to MINRAT, then the returned array is left uncompressed. If the supplied value is zero or negative, then the array will be compressed regardless of the compression ratio.

place

Address of an array placeholder pointer (e.g. generated by the aryPlace function), which indicates the position in the data system where the new array will reside. The placeholder is annulled by this function, and a value of NULL will be returned.

zratio

Returned holding the compression factor actually achieved (the ratio of the supplied array size to the compressed array size). Genuine compressions are represented by values more than 1.0, but values less than 1.0 may be returned if the input data is not suited to delta compression (i.e. if the " compression" actually expands the array storage). Note, the returned value of ZRATIO may be smaller than MINRAT, in which case the supplied array is left unchanged. The returned compression factor is approximate as it does not take into account the space occupied by the HDS metadata describing the extra components of a DELTA array (i.e. the component names, data types, dimensions, etc). This will only be significant for very small arrays.

ary2

Returned holding a pointer to the new DELTA array.

status

The global status.

aryDim
Enquire the dimension sizes of an array

Description:

This function returns the size in pixels of each dimension of an array, together with the total number of dimensions (the size of a dimension is the difference between that dimension' s upper and lower pixel-index bounds + 1).

Invocation:

```
void aryDim( Ary *ary, int ndimx, hdsdim *dim, int *ndim, int *status )
```

Notes:

- If the array has fewer than " ndimx" dimensions, then any remaining elements of the " dim" argument will be filled with 1' s.
- If the array has more than " ndimx" dimensions, then the " ndim" argument will return the actual number of dimensions. In this case only the first " ndimx" dimension sizes will be returned, and an error will result if the size of any of the excluded dimensions exceeds 1.
- The symbolic constant ARY__MXDIM may be used to declare the size of the " dim" argument so that it will be able to hold the maximum number of array dimension sizes that this routine can return. This constant is defined in the header file ary.h.

Parameters

ary Array identifier.

ndimx
Maximum number of dimension sizes to return (i.e. the size of the " dim" array).

dim
An array returned holding the size of each dimension in pixels.

ndim
Returned holding the total number of array dimensions.

status
The global status.

aryDupe
Duplicate an array

Description:

This function duplicates an array, creating a new base array with the same attributes as an existing array (or array section). The new array is left in an undefined state.

Invocation:

```
void aryDupe( Ary *ary1, AryPlace **place, Ary **ary2, int *status )
```

Notes:

- Duplicating a scaled or delta array produces an equivalent simple array.
- If this routine is called with " status" set, then a value of NULL will be returned for the " ary" argument, although no further processing will occur. The same value will also be returned if the function should fail for any reason. In either event, the placeholder will still be annulled.

Parameters

ary1

Identifier for the array to be duplicated.

place

An array placeholder (e.g. generated by the aryPlace routine) which indicates the position in the data system where the new array will reside. The placeholder is annulled by this routine, and a value of NULL will be returned.

ary2

Returned holding the identifier for the new duplicate array.

status

The global status.

aryFind
Find an array in an HDS structure and import it into the ARY_ system

Description:

This function finds a named array within an HDS structure, imports it into the ARY_ system and issues an identifier for it. The imported array may then be manipulated by the ARY_ routines.

Invocation:

```
void aryFind( HDSLoc *loc, const char *name, Ary **ary, int *status )
```

Notes:

- If this routine is called with " status" set, then a NNULL pointer will be returned for the " ary" argument, although no further processing will occur. The same value will also be returned if the routine should fail for any reason.

Parameters

loc Locator to the enclosing HDS structure.

name
Name of the HDS structure component to be imported.

ary Address of variable in which to return the Array identifier.

status
The global status.

aryForm
Obtain the storage form of an array

Description:

This function returns the storage form of an array as an upper-case character string (e.g. ' SIMPLE').

Invocation:

```
void aryForm( Ary *ary, char form[ARY__SZFRM+1], int *status )
```

Notes:

- The symbolic constant ARY__SZFRM should be used for declaring the length of a character variable to hold the storage form of an array. This constant is defined in the header file ary.h.
- At present, the ARY_ routines only support " primitive" , " scaled" , " simple" and " delta" arrays, so only the values ' PRIMITIVE' , ' SCALED' ' DELTA' and ' SIMPLE' can be returned.

Parameters

ary Array identifier.

form
Returned holding the storage form of the array.

status
The global status.

aryFtype
Obtain the full data type of an array

Description:

This function returns the full data type of an array as an upper-case character string (e.g. ' _REAL' or ' COMPLEX_BYTE').

Invocation:

```
void aryFtype( Ary *ary, char ftype[ARY__SZFTP+1], int *status )
```

Notes:

- The symbolic constant ARY__SZFTP should be used for declaring the length of a character variable to hold the full data type of an array. This constant is defined in the header file ary.h.
- For " Scaled" arrays, the data type returned by this function is the data type of the SCALE and ZERO terms, rather than the data type of the stored array.

Parameters

ary Array identifier.

ftype
Returned holding the full data type of the array.

status
The global status.

aryGtdlt
Get the compressed axis and data type for a DELTA array

Description:

This function returns the details of the compression used to produce an array stored in DELTA form. If the array is not stored in DELTA form, then null values are returned as listed below, but no error is reported.

A DELTA array is compressed by storing only the differences between adjacent array values along a nominated compression axis, rather than the full array values. The differences are stored using a smaller data type than the original absolute values. The compression is lossless because any differences that will not fit into the smaller data type are stored explicitly in an extra array with a larger data type. Additional compression is achieved by replacing runs of equal values by a single value and a repeat count.

Invocation:

```
void aryGtdlt( Ary *ary, int *zaxis, char ztype[DAT__SZTYPE+1], float *zratio,
int *status )
```

Parameters

ary Array identifier.

zaxis
Returned holding the index of the pixel axis along which compression occurred. The first axis has index 1. Zero is returned if the array is not stored in DELTA form.

ztype
Returned holding the data type in which the differences between adjacent array values are stored. This will be one of '_BYTE', '_WORD' or '_INTEGER'. The data type of the array itself is returned if the supplid array is not stored in DELTA form.

zratio
Returned holding the compression factor - the ratio of the uncompressed array size to the compressed array size. This is approximate as it does not include the effects of the metadata needed to describe the extra components of a DELTA array (i.e. the space needed to hold the component names, types, dimensions, etc). A value of 1.0 is returned if the supplid array is not stored in DELTA form.

status
The global status.

aryGtsz<T>
Get the scale and zero values for an array

Description:

This function returns the scale and zero values associated with an array. If the array is not stored in scaled form, then values of 1.0 and 0.0 are returned.

Invocation:

```
void aryGtsz<T>( Ary *ary, CGEN_TYPE *scale, CGEN_TYPE *zero, int *status )
```

Parameters

ary Array identifier.

scale
Returned holding the scaling factor.

zero
Returned holding the zero offset.

status
The global status.

aryImprt
Import an array into the ARY_ system from HDS

Description:

This function imports an array into the ARY_ system from HDS and issues an identifier for it. The array may then be manipulated by the ARY_ routines.

Invocation:

```
void aryImprt( HDSLoc *loc, Ary **ary, int *status )
```

Notes:

- The locator supplied as input to this routine may later be annulled without affecting the subsequent behaviour of the ARY_ system.
- If this routine is called with " status" set, then a value of NULL will be returned for the " ary" argument, although no further processing will occur. The same value will also be returned if the routine should fail for any reason.

Parameters

loc HDS locator to an array structure.

ary Returned holding an array identifier for the array structure.

status
The global status.

aryIsacc
Determine whether a specified type of array access is available

Description:

This function determines whether a specified type of access to an array is available, or whether it has been disabled. If access is not available, then any attempt to access the array in this way will fail.

Invocation:

```
void aryIsacc( Ary *ary, const char access[ARY__SZACC+1], int *isacc, int *status
)
```

Notes:

The valid access types control the following operations on the array:

- ' BOUNDS' permits the pixel-index bounds of a base array to be altered.
- ' DELETE' permits deletion of the array.
- ' SHIFT' permits pixel-index shifts to be applied to a base array.
- ' TYPE' permits the data type of the array to be altered.
- ' WRITE' permits new values to be written to the array, or the array' s state to be reset.

Parameters

ary Array identifier.

access
The type of array access required: ' BOUNDS' , ' DELETE' , ' SHIFT' , ' TYPE' or ' WRITE' (see the Notes section for details).

isacc
Returned holding a flag indicating whether the specified type of access is available.

status
The global status.

aryIsbas
Enquire if an array is a base array

Description:

This function returns a boolean value indicating whether the array whose identifier is supplied is a base array (as opposed to an array section).

Invocation:

```
void aryIsbas( Ary *ary, int *base, int *status )
```

Parameters

ary Array identifier.

base
Returned holding a flag indicating whether the array is a base array.

status
The global status.

aryIsmap
Determine if an array is currently mapped

Description:

This function returns a boolean value indicating whether an array is currently mapped for access through the identifier supplied.

Invocation:

```
void aryIsmap( Ary *ary, int *mapped, int *status )
```

Parameters

ary Array identifier.

mapped
Returned holding a flag indicating whether the array is mapped for access through the ARY identifier.

status
The global status.

aryIstmp
Determine if an array is temporary

Description:

This function returns a boolean value indicating whether the specified array is temporary. Temporary arrays are deleted once the last identifier which refers to them is annulled.

Invocation:

```
void aryIstmp( Ary *ary, int *temp, int *status )
```

Parameters

ary Array identifier.

temp
Returned holding a flag indicating whether the array is temporary.

status
The global status.

aryLoc
Obtain an HDS locator for an array

Description:

This function returns an HDS locator for the data object referred to by the supplied ARY identifier.

Invocation:

```
void aryLoc( Ary *ary, HDSLoc **loc, int *status )
```

Parameters

ary Array identifier.

loc Returned holding the HDS locator. It should be annulled using datAnnul when no longer needed. A value of NULL will be returned if an error occurs.

status
The global status.

aryLock
Lock an array for exclusive use by the current thread

Description:

This function locks an ARY array for use by the current thread. An array can be locked for read-only access or read-write access. Multiple threads can lock an array simultaneously for read-only access, but only one thread can lock an array for read-write access at any one time. Use of any ARY function that may modify any aspect of the array - either the data values stored in the array or the meta-data describing the whole array - will fail with an error unless the thread has locked the array for read-write access. Use of an ARY function that cannot modify the array will fail with an error unless the thread has locked the array (in this case the lock can be either for read-only or read-write access).

If " readonly" is zero (indicating the current thread wants to modify some aspect of the array), this function will report an error if any other thread currently has a lock (read-only or read-write) on the array.

If " readonly" is non-zero (indicating the current thread wants read-only access to the array), this function will report an error only if another thread currently has a read-write lock on the array.

The current thread must unlock the array using datUnlock before it can be locked for use by another thread. All arrays are initially locked by the current thread when they are created or opened. The type of access available to the array (" Read" , " Write" or " Update") determines the type of the initial lock. For pre-existing arrays, this is determined by the access mode specified when it is first opened. For new and temporary arrays, the initial lock is always a read-write lock.

Invocation:

```
aryLock( Ary *ary, int readonly, int *status );
```

Notes:

- If the version of HDS being used does not support object locking, this function will return without action unless the HDS tuning parameter V4LOCKERROR is set to a non-zero value, in which case an error will be reported.
- An error will be reported if the supplied array is currently locked by another thread.
- The majority of ARY functions will report an error if the array supplied to the function has not been locked for use by the calling thread. The exceptions are the functions that manage these locks - aryLock, datUnlock and aryLocked.
- Attempting to lock an array that is already locked by the current thread will change the type of lock (read-only or read-write) if the lock types differ, but will otherwise have no effect.

Parameters

ary Pointer to the array that is to be locked.

readonly
If non-zero, the array is locked for read-only access. Otherwise it is locked for read-write access.

status = int$*$ (Given and Returned)
Pointer to global status.

aryLocked
See if an array is locked

Description:

This function returns a value that indicates if the supplied ARY array has been locked for use by one or more threads. A thread can lock an array either for read-only access or for read-write access. The lock management functions (aryLock and aryUnlock) will ensure that any thread that requests and is given a read-write lock will have exclusive access to the array - no other locks of either type will be issued to other threads until the first thread releases the read-write lock using aryUnlock. If a thread requests and is given a read-only lock, the lock management functions may issue read-only locks to other threads, but it will also ensure that no other thread is granted a read-write lock until all read-only locks have been released.

Invocation:

```
locked = aryLocked( const Ary *ary, int *status );
```

Notes:

- Zero is returned as the function value if an error has already occurred, or if an error occurs in this function.

Parameters

ary Pointer to the array to be checked.

status
Pointer to global status.

Returned function value :

A value indicating the status of the supplied array:

- 1: The application is is linked with a version of HDS that does not support object locking.

0: the supplied array is unlocked. This is the condition that must be met for the current thread to be able to lock the supplied array for read-write access using function aryLock. This condition can be achieved by releasing any existing locks using aryUnlock.

1: the supplied array is locked for reading and writing by the current thread. This is the condition that must be met for the current thread to be able to use the supplied array in any ARY function that might modify the array (except for the locking and unlocking functions - see below). This condition can be achieved by calling aryLock.

2: the supplied array is locked for reading and writing by a different thread. An error will be reported if the current thread attempts to use the array in any other ARY function.

3: the supplied array is locked read-only by the current thread (and maybe other threads as well). This is the condition that must be met for the current thread to be able to use the supplied array in any ARY function that cannot modify the array. An error will be reported if the current thread attempts to use the array in any ARY function that could modify the array. This condition can be achieved by calling aryLock.

4: the supplied array is not locked by the current thread, but is locked read-only by one or more other threads. An error will be reported if the current thread attempts to use the array in any other ARY function.

aryMap
Obtain mapped access to an array

Description:

This function obtains mapped access an array, returning a pointer to the mapped values and a count of the number of elements mapped.

Invocation:

```
void aryMap( Ary *ary, const char *type, const char *mmod, void **pntr, size_t
*el, int *status )
```

Notes:

- If the array is a scaled array, the returned mapped values will be the stored array values multiplied by the scale factor and shifted by the zero term.
- If the array is a delta (i.e. compressed) array, the returned mapped values will be the uncompressed array values.
- Currently, only READ access is available for scaled and compressed arrays. An error will be reported if an attempt is made to get WRITE or UPDATE access to a scaled or compressed array.

Parameters

ary Array identifier.

type
The numerical data type required for access (e.g. "_REAL").

mmod
The mapping mode for access to the array: "READ", "UPDATE" or "WRITE", with an optional initialisation mode "/BAD" or "/ZERO" appended.

pntr
Returned holding a pointer to the mapped values.

el Returned holding the number of elements mapped.

status
The global status.

aryMapz
Obtain complex mapped access to an array

Description:

This function obtains complex mapped access to an array, returning pointers to the real and imaginary values and a count of the number of elements mapped.

Invocation:

```
void aryMapz( Ary *ary, const char *type, const char *mmod, void **rpntr, void
**ipntr, size_t *el, int *status )
```

Parameters

ary Array identifier.

type
The numerical data type required for accessing the array (e.g. '_REAL').

mmod
The mapping mode for access to the array: 'READ', 'UPDATE' or 'WRITE', with an optional initialisation mode '/BAD' or '/ZERO' appended.

rpntr
Returned holding a pointer to the mapped real (i.e. non-imaginary) values.

ipntr
Returned holding a pointer to the mapped imaginary values.

el Returned holding the number of elements mapped.

status
The global status.

aryMsg
Assign the name of an array to a message token

Description:

This function assigns the name of an array to a message token (in a form which a user will understand) for use in constructing messages with the MSG_ and ERR_ routines (see SUN/104).

Invocation:

```
void aryMsg( const char *token, Ary *ary )
```

Notes:

- This routine has no " status" argument and performs no error checking. If it should fail, then no assignment to the message token will be made and this will be apparent in the final message.

Parameters

token
Name of the message token.

ary Array identifier.

aryNdim
Enquire the dimensionality of an array

Description:

This routine determines the number of dimensions which an array has.

Invocation:

```
void aryNdim( Ary *ary, int *ndim, int *status )
```

Parameters

ary Array identifier.

ndim
Returned holding the number of array dimensions.

status
The global status.

aryNew
Create a new simple array

Description:

This function creates a new simple array and returns an identifier for it. The array may subsequently be manipulated with other Ary functions.

Invocation:

```
void aryNew( const char *ftype, int ndim, const hdsdim *lbnd, const hdsdim *ubnd,
AryPlace **place, Ary **ary, int *status )
```

Notes:

- If this routine is called with " status" set, then a value of NULL will be returned for the " ary" argument, although no further processing will occur. The same value will also be returned if the routine should fail for any reason. In either event, the placeholder will still be annulled.

Parameters

ftype
Full data type of the array.

ndim
Number of array dimensions.

lbnd
Lower pixel-index bounds of the array.

ubnd
Upper pixel-index bounds of the array.

place
On entry, holds an array placeholder (e.g. generated by the aryPlace function) which indicates the position in the data system where the new array will reside. The placeholder is annulled by this function, and a value of NULL will be returned on exit.

ary Returned holding an identifier for the new array.

status
The global status.

aryNewp
Create a new primitive array

Description:

This function creates a new primitive array and returns an identifier for it. The array may subsequently be manipulated with other Ary routines.

Invocation:

```
void aryNewp( const char *ftype, int ndim, const hdsdim *ubnd, AryPlace **place,
Ary **ary, int *status )
```

Notes:

- If this routine is called with " status" set, then a value of NULL will be returned for the " ary" argument, although no further processing will occur. The same value will also be returned if the routine should fail for any reason. In either event, the placeholder will still be annulled.

Parameters

ftype
Data type of the array (e.g. ' _REAL'). Note that complex types are not allowed for primitive arrays.

ndim
Number of array dimensions.

ubnd
Upper pixel-index bounds of the array (the lower bound of each dimension is taken to be 1).

place
On entry, holds an array placeholder (e.g. generated by the aryPlace function) which indicates the position in the data system where the new array will reside. The placeholder is annulled by this function, and a value of NULL will be returned on exit.

ary Returned holding an identifier for the new array.

status
The global status.

aryNoacc
Disable a specified type of access to an array

Description:

This function disables the specified type of access to an array, so that any subsequent attempt to access it in that way will fail. Access restrictions imposed on an array identifier by this routine will be propagated to any new identifiers derived from it, and cannot be revoked.

Invocation:

```
void aryNoacc( const char *access, Ary *ary, int *status )
```

Notes:

Disabling each type of access imposes the following restrictions on an array:

- ' BOUNDS' prevents the pixel-index bounds of a base array from being altered.
- ' DELETE' prevents the array being deleted.
- ' MODIFY' prevents any form of modification to the array (i.e. it disables all the other access types).
- ' SCALE' prevents the scale and zero values from being changed.
- ' SHIFT' prevents pixel-index shifts from being applied to a base array.
- ' TYPE' prevents the data type of the array from being altered.
- ' WRITE' prevents new values from being written to the array, or the array' s state from being reset.

Parameters

access
The type of access to be disabled: ' BOUNDS' , ' DELETE' , ' MODIFY' , ' SCALE' , ' SHIFT' , ' TYPE' or ' WRITE' .

ary Array identifier.

status
The global status.

aryOffs
Obtain the pixel offset between two arrays

Description:

This function returns the pixel offset for each requested dimension between two arrays. These values are the offsets which should be added to the pixel indices of the first array to obtain the indices of the corresponding pixel in the second array.

Invocation:

```
void aryOffs( Ary *ary1, Ary *ary2, int mxoffs, hdsdim *offs, int *status )
```

Notes:

- The two array identifiers supplied need not refer to the same base array (although they may often do so). If they do not, then the offset between the pixels in each array is determined by matching the pixel indices of their respective base arrays.
- Note that non-zero pixel offsets may exist even for dimensions which exceed the dimensionality of either of the two arrays supplied. The symbolic constant ARY__MXDIM may be used to declare the size of the OFFS argument so that it will be able to hold the maximum number of non-zero offsets that this routine can return.

Parameters

iary1
First array identifier.

iary2
Second array identifier.

mxoffs
Maximum number of pixel offsets to return (i.e. the declared size of the supplied " offs" array).

offs
Returned holding an array of pixel offsets for each dimension.

status
The global status.

aryPlace
Obtain an array placeholder

Description:

This function returns an array placeholder. A placeholder is used to identify a position in the underlying data system (HDS) and may be passed to other routines (e.g. aryNew) to indicate where a newly created array should be positioned.

Invocation:

```
void aryPlace( HDSLoc *loc, const char *name, AryPlace **place, int *status )
```

Notes:

- Placeholders are intended only for local use within an application and only a limited number of them are available simultaneously. They are always annulled as soon as they are passed to another routine to create a new array, where they are effectively exchanged for an array identifier.
- If this routine is called with " status" set, then a value of NULL will be returned for the " place" argument, although no further processing will occur. The same value will also be returned if the routine should fail for any reason.

Parameters

loc HDS locator to the structure to contain the new array.

name
Name of the new structure component (i.e. the array).

place
Returned holding an array placeholder identifying the nominated position in the data system.

status
The global status.

aryPtsz<T>
Set new scale and zero values for a scaled array

Description:

This function sets new values for the scale and zero values associated with an array. If the array is stored in simple form, then the storage form is changed to scaled.

Invocation:

```
void aryPtsz<T>( Ary *ary, CGEN_TYPE scale, CGEN_TYPE zero, int *status )
```

Notes:

- This routine may only be used to change the type of a base array. If it is called with an array which is not a base array, then it will return without action. No error will result.
- An error will result if the array, or any part of it, is currently mapped for access (e.g. through another identifier).

Parameters

ary Array identifier.

scale
The new value for the scaling factor.

zero
The new value for the zero offset.

status
The global status.

aryReset
Reset an array to an undefined state

Description:

This function resets an array so that its values become undefined. Its use is advisable before making format changes to an array if retention of the existing values is not required (e.g. before changing its data type with the aryStype function); this will avoid the cost of converting the existing values.

Invocation:

```
void aryReset( Ary *ary, int *status )
```

Notes:

- This function may only be used to reset the state of a base array. If an array section is supplied, then it will return without action. No error will result.
- An array cannot be reset while it is mapped for access. This routine will fail if this is the case.

Parameters

ary Array identifier.

status
The global status.

aryRound
Set the internal ARY_ system rouding flag

Description:

This function sets an internal flag in the ARY_ system that controls how floating point values are converted to integer values when doing array type conversion. If the flag is non-zero floating point values are rounded to the nearest integer value. Otherwise, floating point values are truncated towards zero.

Invocation:

```
int aryRound( int newflg )
```

Notes:

- By default, the rounding flag is set to zero, so floating point values are truncated towards zero when being converted to integers.

Parameters

newflg

The new value to be set for the rounding flag. If a negative value is supplied, the existing value is left unchanged.

arySame
Enquire if two arrays are part of the same base array

Description:

This function determines whether two array identifiers refer to parts of the same base array. If so, it also determines whether they intersect.

Invocation:

```
void arySame( Ary *ary1, Ary *ary2, int *same, int *isect, int *status )
```

Notes:

- Two arrays (or array sections) are counted as intersecting if (i) they both refer to the same base array and (ii) altering values in one of the arrays can result in the values in the other array changing in consequence.

Parameters

ary1
Identifier for the first array (or array section).

ary2
Identifier for the second array (or array section).

same
Returned holding a boolean flag indicating whether the identifiers refer to parts of the same base array.

isect
Returned holding a boolean flag indicating whether the arrays intersect.

status
The global status.

arySbad
Set the bad-pixel flag for an array

Description:

This function sets the value of the bad-pixel flag for an array A call to this routine with " bad" set non-zero declares that the specified array may contain bad pixel values for which checks must be made by algorithms which subsequently processes its values. A call with " bad" set to zero declares that there are definitely no bad values present and that subsequent checks for such values may be omitted.

Invocation:

```
void arySbad( int bad, Ary *ary, int *status )
```

Notes:

- If the array is mapped for access when this routine is called, then the bad-pixel flag will be associated with the mapped values. This information will only be transferred to the actual data object when the array is unmapped (but only if it was mapped for UPDATE or WRITE access). The value transferred may be modified if conversion errors occur during the unmapping process.

Parameters

bad
Bad-pixel flag value to be set.

ary Array identifier.

status
The global status.

arySbnd
Set new pixel-index bounds for an array

Description:

This function sets new pixel-index bounds for an array (or array section). The number of array dimensions may also be changed. If a base array is specified, then a permanent change is made to the actual data object and this will be apparent through any other array identifiers which refer to it. However, if an identifier for an array section is specified, then its bounds are altered without affecting other arrays.

Invocation:

```
void arySbnd( int ndim, const hdsdim *lbnd, const hdsdim *ubnd, Ary *ary, int
*status )
```

Notes:

- The bounds of an array section cannot be altered while it is mapped for access through the identifier supplied to this routine.
- The bounds of a base array cannot be altered while any part of it is mapped for access (i.e. even through another identifier).
- The array' s pixel values (if defined) will be retained for those pixels which lie within both the old and new bounds. Any pixels lying outside the new bounds will be lost (and cannot later be recovered by further changes to the array' s bounds). Any new pixels introduced where the new bounds extend beyond the old ones will be assigned the " bad" value, and the subsequent value of the bad-pixel flag will reflect this.
- If the bounds of a base array are to be altered and retention of the existing pixel values is not required, then a call to aryReset should be made before calling this routine. This will eliminate any processing which might otherwise be needed to retain the existing values. This step is not necessary with an array section, where no processing of pixel values takes place.

Parameters

ndim
New number of array dimensions.

lbnd
Array holding the new lower pixel-index bounds of the array.

ubnd
Array holding the new upper pixel-index bounds of the array.

ary Array identifier.

status

The global status.

arySctyp
Obtain the numeric type of a scaled array

Description:

This function returns the numeric type of a scaled array as an upper-case character string (e.g. '_REAL'). The returned type describes the values stored in the array, before they are unscaled using the associated scale and zero values. Use aryType if you need the data type of the array after it has been unscaled.

Invocation:

```
void arySctyp( Ary *ary, char type[ARY__SZTYP+1], int *status )
```

Notes:

- If the array is not stored in SCALED form, then this routine returns the same type as the aryType function.
- The symbolic constant ARY__SZTYP should be used to declare the length of a character variable which is to hold the numeric type of an array. This constant is defined in the header file ary_par.h.

Parameters

ary Array identifier.

type
Numeric type of the array.

status
The global status.

arySect
Create an array section

Description:

This function creates a new array section which refers to a selected region of an existing array (or array section). The section may be larger or smaller in extent than the original array.

Invocation:

```
void arySect( Ary *ary1, int ndim, const hdsdim *lbnd, const hdsdim *ubnd, Ary
**ary2, int *status )
```

Notes:

- The number of section dimensions need not match the number of dimensions in the initial array. Pixel-index bounds will be padded with 1' s as necessary to identify the pixels to which the new section should refer.
- Note that sections which extend beyond the pixel-index bounds of the initial array will be padded with bad pixels.
- If this routine is called with " Status" set, then a value of NULL will be returned for the " ary2" argument, although no further processing will occur. The same value will also be returned if the routine should fail for any reason.

Parameters

ary Identifier for the initial array.

ndim
Number of dimensions for new section. This is the length of the " lbnd" and Ubnd" arrays.

lbnd
Lower pixel-index bounds for the new section.

ubnd
Upper pixel-index bounds for the new section.

ary2
Address of a variable in which to return an identifier for the new section.

status
The global status.

aryShift
Apply pixel-index shifts to an array

Description:

This function applies pixel-index shifts to an array. An integer shift is applied to each dimension so that its pixel-index bounds, and the indices of each pixel, change by the amount of shift applied to the corresponding dimension. The array's pixels retain their values and none are lost.

Invocation:

```
void aryShift( int nshift, const hdsdim *shift, Ary *ary, int *status )
```

Notes:

- Pixel-index shifts applied to a base array will affect the appearance of that array as seen by all base-array identifiers associated with it. However, array sections derived from that base array will remain unchanged (as regards both pixel-indices and data content).
- Pixel-index shifts cannot be applied to a base array while any part of it is mapped for access (i.e. even through another identifier).
- Pixel-index shifts applied to an array section only affect that section itself, and have no effect on other array identifiers.
- Pixel-index shifts cannot be applied to an array section while it is mapped for access through the identifier supplied to this routine.

Parameters

nshift
Number of dimensions to which shifts are to be applied (i.e. the length of array " shift"). This must not exceed the number of array dimensions. If fewer shifts are supplied than there are dimensions in the array, then the extra dimensions will not be shifted.

shift
An array holding the pixel-index shifts to be applied to each dimension.

ary Array identifier.

status
The global status.

arySize
Determine the size of an array

Description:

This function returns the number of pixels in the array whose identifier is supplied (i.e. the product of its dimensions).

Invocation:

```
void arySize( Ary *ary, int npix, int *status )
```

Parameters

ary Array identifier.

npix
Returned holding the number of pixels in the array.

status
The global status.

arySsect
Create a similar array section to an existing one

Description:

This function creates a new array section, using an existing section as a template. The new section bears the same relationship to its base array as the template section does to its own base array. Allowance is made for pixel-index shifts which may have been applied so that the pixel-indices of the new section match those of the template. The number of dimensions of the input and template arrays may differ.

Invocation:

```
void arySsect( Ary *ary1, Ary *ary2, Ary **ary3, int *status )
```

Notes:

- This routine normally generates an array section. However, if both input arrays are base arrays with identical pixel-index bounds, then there is no need to create a section in order to access the required part of the first array. In this case a base array identifier will be returned instead.
- The new section created by this routine will have the same number of dimensions as the array (or array section) from which it is derived. If the template (ary2) array has fewer dimensions than this, then the bounds of any additional input dimensions are preserved unchanged in the new array. If the template (ary2) array has more dimensions, then the excess ones are ignored.
- This routine takes account of the regions of each base array to which the input array sections have access. It may therefore restrict the region accessible to the new section (and pad with " bad" pixels) so as not to grant access to regions of the base array which were not previously accessible through the input arrays.
- If this routine is called with " status" set, then a value of NULL will be returned for the " ary3" argument, although no further processing will occur. The same value will also be returned if the routine should fail for any reason.

Parameters

ary1
Identifier for the input array from which the section is to be derived. This may be a base array or an array section.

ary2
Identifier for the template section (this may also be a base array or an array section).

ary3
Returned holding the identifier for the new array section.

status

The global status.

aryState
Determine the state of an array (defined or undefined)

Description:

This function returns a flag indicating whether an array' s pixel values are currently defined.

Invocation:

```
void aryState( Ary *ary, int *state, int *status )
```

Parameters

ary Array identifier.

state
Returned hoplding a flag indicating whether the array' s pixel values are defined.

status
The global status.

aryStype
Set a new type for an array

Description:

This function sets a new full type for an array, causing its data storage type to be changed. If the array's pixel values are defined, then they will be converted from the old type to the new one. If they are undefined, then no conversion will be necessary. Subsequent enquiries will reflect the new type. Conversion may be performed between any types supported by the ary_ routines, including from a non-complex type to a complex type (and vice versa).

Invocation:

```
void aryStype( const char *ftype, Ary *ary, int *status )
```

Notes:

- This function may only be used to change the type of a base array. If it is called with an array which is not a base array, then it will return without action. No error will result.
- An error will result if the array, or any part of it, is currently mapped for access (e.g. through another identifier).
- If the type of an array is to be changed without its pixel values being retained, then a call to aryReset should be made beforehand. This will avoid the cost of converting all the values.

Parameters

ftype
The new full type specification for the array (e.g. '_REAL' or 'COMPLEX_INTEGER').

ary Array identifier.

status
The global status.

aryTemp
Obtain a placeholder for a temporary array

Description:

This function returns an array placeholder which may be used to create a temporary array (i.e. one which will be deleted automatically once the last identifier associated with it is annulled). The placeholder returned by this routine may be passed to other routines (e.g. aryNew or aryCopy) to produce a temporary array in the same way as a new permanent array would be created.

Invocation:

```
void aryTemp( AryPlace **place, int *status )
```

Notes:

- Placeholders are intended only for local use within an application and only a limited number of them are available simultaneously. They are always annulled as soon as they are passed to another routine to create a new array, where they are effectively exchanged for an array identifier.
- If this routine is called with STATUS set, then a value of NULL will be returned for the " place" argument, although no further processing will occur. The same value will also be returned if the routine should fail for any reason.

Parameters

place
Returned holding a placeholder for a temporary array.

status
The global status.

aryTrace
Set the internal ARY_ system error-tracing flag

Description:

This function sets an internal flag in the ARY_ system which enables or disables error-tracing messages. If this flag is set non-zero, then any error occurring within the ARY_ system will be accompanied by error messages indicating which internal routines have exited prematurely as a result. If the flag is set to zero, this internal diagnostic information will not appear and only standard error messages will be produced.

Invocation:

```
char aryTrace( char newflg )
```

Notes:

- By default, the error tracing flag is set to zero, so no internal diagnostic information will be produced.

Parameters

newflg

The new value to be set for the error-tracing flag. If a negative value is supplied, the existing value is left unchanged.

aryType
Obtain the numeric type of an array

Description:

This function returns the numeric type of an array as an upper-case character string (e.g. ' _REAL').

Invocation:

```
void aryType( Ary *ary, char type[ARY__SZTYP+1], int *status )
```

Notes:

- The symbolic constant ARY__SZTYP should be used for declaring the length of a character array which is to hold the numeric type of an array. This constant is defined in the header file ary_par.h

Parameters

ary Array identifier.

type
Numeric type of the array.

status
The global status.

aryUnlock
Unlock an array so that it can be locked by a different thread

Description:

This function ensures that the current thread does not have a lock of any type on the supplied array. See aryLock.

The array must be locked again, using aryLock, before it can be used by any other ARY function. All arrays are initially locked by the current thread when they are first created or opened.

Invocation:

```
aryUnlock( Ary *ary, int *status );
```

Notes:

- If the version of HDS being used does not support object locking, this function will return without action unless the HDS tuning parameter V4LOCKERROR is set to a non-zero value, in which case an error will be reported.
- No error is reported if the supplied array is currently locked for read-only or read-write access by another thread.
- The majority of ARY functions will report an error if the array supplied to the function has not been locked for use by the calling thread. The exceptions are the functions that manage these locks - aryLock, aryUnlock and aryLocked.
- Attempting to unlock an array that is not locked by the current thread has no effect, and no error is reported. The aryLocked function can be used to determine if the current thread has a lock on the array.

Parameters

ary Pointer to to the array to be unlocked.

status
Pointer to global status.

aryUnmap
Unmap an array

Description:

This function unmaps an array which has previously been mapped for READ, UPDATE or WRITE access.

Invocation:

```
void aryUnmap( Ary *ary, int *status )
```

Notes:

- This function 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.
- An error will result if the array has not previously been mapped for access.

Parameters

ary Array identifier.

status
The global status.

aryValid
Determine whether an array identifier is valid

Description:

Determine whether an array identifier is valid (i.e. associated with an array).

Invocation:

```
int aryValid( Ary *ary, int *status )
```

Returned Value:

A flag indicating if the identifier is valid.

Parameters

ary Array identifier to be tested.

status
The global status.

aryVerfy
Verify that an array’ s data structure is correctly constructed

Description:

This function checks that the data structure containing an array is correctly constructed and that the array’ s pixel values are defined. It also checks for the presence of any " rogue" components in the data structure. If an anomaly is found, then an error results. Otherwise, the routine returns without further action.

Invocation:

```
void aryVerfy( Ary *ary, int *status )
```

Parameters

ary Array identifier.

status
The global status.

E Changes and new features in V1.1

Only relatively minor changes have taken place since the previous version (V1.0) of the ARY_ system. The most significant of these are as follows:

(1) An obscure bug resulting from an un-annulled HDS locator has been fixed. This could occasionally result in corrupted files if the ARY_STYPE routine was called repeatedly from successive invocations of an application in an ADAM monolith.

(2) Two new routines have been introduced, primarily to provide facilities required by the NDF_ system:

- **ARY_NDIM(IARY, NDIM, STATUS)**
  Enquire the dimensionality of an array
- **ARY_OFFS(IARY1, IARY2, MXOFFS, OFFS, STATUS)**
  Obtain the pixel offset between two arrays

(3) A stand-alone (non-ADAM) version of the ARY_ library has been added and new linker options files have been provided to allow linking with either version.

(4) The encoding of ARY_ system identifiers has been changed to improve the chance of detecting erroneous identifier values.

(5) Messages about array data structures now contain the full HDS object name, including the full container file name.

(6) A few minor documentation errors have been corrected.

No changes to existing applications should be required, neither should any re-compilation or re-linking be necessary.

F Changes and new features in V1.3

The most significant changes in version (V1.3) of the ARY_ system were as follows:

(1) A new array storage form called “DELTA” has been introduced. This provides lossless compression for integer arrays. The new routine ARY_DELTA creates a copy of a supplied array, stored in DELTA form.

(2) A new routine called ARY_LOC returns a locator for the HDS data object referred to by the supplied ARY identifier.

G Changes and new features in V1.4

The most significant changes in version (V1.4) of the ARY_ system are as follows:

(1) A bug in ARY_DUPE has been fixed. This bug resulted in the wrong data type for the output array if the input array was compressed.

H Changes and new features in V2.0

The most significant changes in version (V2.0) of the ARY_ system are as follows:

(1) This is a complete new re-write of the orignial Fortran code in C.

(2) The C Interface is thread-safe, including new facilities for locking and unlocking arrays for exclusive use by the current thread.

(3) The Fortran API has been expanded ito include versions of routines that allow pixel index and count values to be given and returned in 8 byte integers.

No changes to existing applications should be required, neither should any re-compilation or re-linking be necessary (so long as you do not need to use any of the new features of course).