SUN/92.14

Starlink Project
Starlink User Note 92.14

R.F. Warren-Smith
M.D. Lawden
B.K. McIlwrath
T. Jenness
P.W. Draper
D.S. Berry

6th February 2018

HDS
Hierarchical Data System
Version 6.0
Programmer's Manual

Abstract

HDS is a library of functions used for accessing file-based hierarchical data and supports the storage of a wide variety of information. It is particularly suited to the storage of large multi-dimensional arrays (with their ancillary data) where efficient access is needed.

Historically the HDS library used a unique format for disk-file storage developed by the UK Starlink project in the 1980's. However it now supports both this historical Starlink format and also the popular HDF5 format.

HDS organises data into hierarchies, broadly similar to the directory structure of a hierarchical filing system, but contained within a single HDS *container file.* The structures stored in these files are self-describing and flexible; HDS supports modification and extension of structures previously created, as well as deletion, copying, renaming, *etc.*

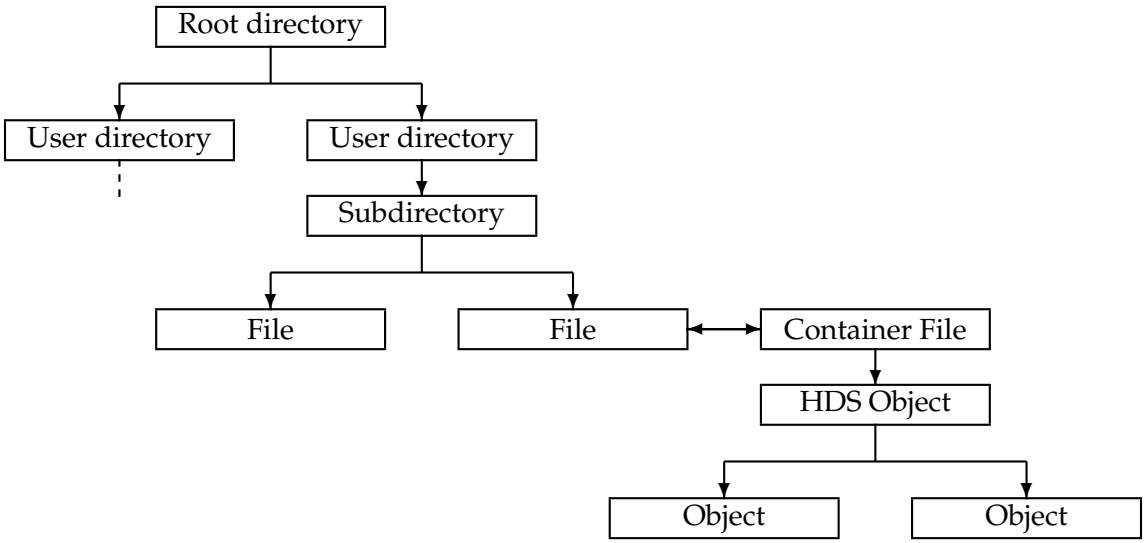
All information stored in HDS files is portable between the machines on which HDS is implemented. Thus, there are no format conversion problems when moving between machines.

The routines described in this document may be used to perform operations on any HDS data. In addition, HDS forms a toolkit for the construction of higher level (more specialised) data structures and the software which accesses them. HDS routines are therefore invoked indirectly by many other items of Starlink software.

Contents

List of Figures

Figure 1: The relationship between a computer's filing system and HDS.



1 INTRODUCTION

This document is intended for those people who intend to write programs using HDS. It is tutorial in style and assumes no prior knowledge of HDS, although some knowledge of a hierarchical filing system will be helpful. Anyone who has used UNIX should understand the analogies that are drawn between directories and HDS structures.

This note mainly deals with HDS as a "stand-alone" software package. Routines which interface HDS with a particular software environment (*i.e.* the HDS/ADAM parameter system routines) are not considered here.[1]

Reference information is presented in the appendices. Appendix A gives a list of the calling sequences for the HDS routines; this will be useful to experienced programmers who just want to be reminded of the parameters required, while Appendix B gives full routine specifications for all of HDS. Appendix E describes the HDS error codes.

HDS stands for "Hierarchical Data System". It is a flexible system for storing and retrieving data and takes over from a computer's filing system at the level of an individual file (see Fig 1). A conventional file effectively contains an 1-dimensional sequence of data elements, whereas an HDS file can contain a more complex structure. There are many parallels between the hierarchical way HDS stores data within files and the way that a filing system organises the files themselves. These analogies will be helpful in what follows.

The advantage of HDS is that it allows many different kinds of data to be stored in a consistent

[1]See ADAM Programmer Note 7 for a description of these routines.

and logical fashion. It is also very flexible, in that objects can be added or deleted whilst retaining the logical structure. HDS also provides portability of data, so that the same data objects may be accessed from different types of computer despite the fact that each may actually format its files and data in different ways.

2 DISK FORMATS AND HDF5

Historically, the term "HDS" was used to refer both to a subroutine library and also to an on-disk data format. Both were developed by the UK Starlink project in the 1980's. Since then the subroutine library has been used extensively - mainly within the Starlink Software Collection but also outside. However the size of typical astronomical data sets has increased enormously since the 1980's (as has the typical astronomer's desktop computing power) and the original disk format upon which the HDS library was based no longer looks like a good match to current needs. Solving this problem by upgrading the original disk format was deemed impractical as documentation on the internals of the HDS disk format is sparse and the original developers are no longer available. Instead, it was decided to move to the popular and well supported HDF5 library and corresponding disk format - see `https://support.hdfgroup.org/HDF5/`

However, given the extensive use of the HDS library within the Starlink Software Collection, simply changing all the HDS calls within starlink application code to equivalent HDF5 calls would be prohibitive, both from the point of view of the effort involved and also the likelihood of bugs being introduced.

Instead it was decided to re-implement the routines within the HDS library using calls to the routines provided by the HDF5 library, using the HDF5 disk format in a way that mimics as closely as possible the original HDS disk format. In this way no changes would be needed to application code since all HDS routines would still be available and would behave in the same way. In addition, data files created by the new HDS library would be compatible with the wide range of publicly available tools that already exist for examining HDF5 files.

For further information about the motivation behind this change and the technicalities involved see "Re-implementing the Hierarchical Data System using HDF5" (T. Jenness, Astronomy and Computing, Vol.12, 2015 - `https://arxiv.org/abs/1502.04029`).

Note, this document has not yet been fully updated to refer to the new library and disk format. If in doubt, send a query to the Starlink support mailing list or consult the code (held in various repositories within the Starlink project on github).

2.1 Choosing the disk format version for new HDS files

The new HDF5-based disk format is called "HDS version 5" (the previous Starlink-specific disk format was version 4). Perhaps confusingly, the HDS library and data format have always had independent version numbers. So where-as the new HDF5-based *data format* is "HDS version 5", the new *library* is "HDS version 6.0".

An obvious requirement was that version 6.0 of the HDS library should be able to read and write both data formats (version 4 and version 5). This is necessary for it to be possible to pass data files to and from sites that still use version 5 of the HDS library (i.e. can only access files

that use the Starlink-specific HDS disk format). However, when creating a new disk file, the HDS library (version 6) needs to known which disk format (version 4 or version 5) to use. By default, the original disk format (version 4) is used so that files created by version 6 of the HDS library can be read by starlink systems that still use version 5. However, this can be changed by setting the environment variable "`HDS_VERSION`" to "5", or by using the HDS_TUNE routine to set tuning parameter "`VERSION`" to 5.

All data files created by version 6 of the HDS library still have the traditional extension - "`.sdf`" - ("Starlink Data File"), regardless of the disk format in use.

Note, in a future release the default disk format will be changed from version 4 to version 5.

3 OBJECTS

HDS files are known as *container files* and by default have the extension '.sdf'. HDS files contain *data objects* which will often be referred to simply as *objects*. An object is an entity which contains data or other objects. This is the basis of the hierarchical nature of HDS and is analogous to the concepts of *file* and *directory* – a directory can contain files and directories which can themselves contain files and directories and so on (see Fig 1).

An HDS object possesses a number of attributes, each of which is described in more detail below:

3.1 Name

The primary way of identifying an HDS object is by its *name*, which must be unique within its own container object. The name of an object is a character string which may contain any printing characters; white space is ignored and alphabetic characters are capitalised. The maximum length of an HDS name is 15 characters.

There are no special rules governing the first character (*i.e.* it can be numeric), so HDS itself allows great freedom in specifying names (and also types – see below). In practice, however, some restrictions will be imposed by considerations of portability of data and applications, and of possible syntax conflicts with the environment within which HDS is used.

3.2 Type

The *type* of an HDS object falls into two *classes*:

- Structure
- Primitive

Structure objects contain other objects called *components*. Primitive objects contain only numeric, character, or logical values. Objects in the different classes are referred to as *structures* and *primitives* while the more general term *object* refers to either a *structure* or a *primitive*. Structures are analogous to the directories in a filing system – they can contain a part of the hierarchy below

them. Primitives are analogous to files – they are at the bottom of any branch of the structure and contain the actual data.

In HDS, structure types are represented by character strings with the same rules of formation as *name,* except that a structure type may not start with an underscore character '_' (a structure type may also be completely blank). Examples of structure types are 'IMAGE', 'SPECTRUM', 'INSTR_RESP', *etc.* These do not begin with an underscore, so they are easily distinguished from the primitive types, which do.

Special rules apply to the primitive types, which all begin with an underscore and are "pre-defined" by HDS, as follows:

HDS Type	*Fortran Type*	*Description*
_INTEGER	INTEGER	Integer (signed)
_INT64	INTEGER∗8	64-bit Integer (signed)
_REAL	REAL	Single precision
_DOUBLE	DOUBLE PRECISION	Double precision
_LOGICAL	LOGICAL	Logical
_CHAR[∗n]	CHARACTER[∗n]	Character string
_WORD	INTEGER∗2	Word (signed)
_UWORD	INTEGER∗2	Unsigned word
_BYTE	BYTE	Byte (signed)
_UBYTE	BYTE	Unsigned byte

The first five of these primitive types are referred to as *standard data types*, since they correspond with standard Fortran 77 data types. The last four are *non-standard data types* and are typically required to accommodate raw data from instrumental hardware, which may generate numbers in these formats. The range of values which may be stored in each primitive numeric type is determined by the numerical capabilities of your machine and the particular implementation of Fortran you are using. Details of HDS, which include information about the properties of the primitive data types on the system you are using, may be found in the file:

```
hds_machine
```

in the directory `$STARLINK_DIR/help`.

The rules by which character values are handled by HDS is the same as for Fortran 77, *i.e.* character values are padded with blanks or truncated from the right depending on the relative length of the program value and the object.

3.3 Shape

Every HDS object has a *shape* or dimensionality. This is described by an integer (the number of dimensions) and an integer array (containing the size of each dimension). A *scalar* (for example

a single number) has, by convention, a dimensionality of zero; *i.e.* its number of dimensions is 0. A *vector* has a dimensionality of 1; *i.e.* its number of dimensions is 1, and the first element of the dimension array contains the size of the vector. An *array* refers to an object with 2 or more dimensions; a maximum of 7 dimensions are allowed. Objects may be referred to as *scalar primitives* or *vector structures* and so on.

3.4 State

The *state* of an HDS object specifies whether or not its value is defined. It is represented as a logical value where .TRUE. means defined and .FALSE. means undefined.

Objects start out undefined when they are created and become defined when you write a value to them. In general, an error will result if you attempt to obtain the value of an object while it is still undefined.

3.5 Group

In order to access an HDS object it is first necessary to "associate" a *locator* with it (like opening a file). This locator can then be used to address the object. When the program no longer needs to access the object the locator should be "annulled" (like closing the file again). A locator is roughly analogous to a Fortran logical unit number (but is actually a character variable, not an integer). The *group* attribute is used to form an association between objects so that all their locators can be annulled together. A group is written as a character string whose rules of formation are the same as for *name*.

3.6 Value

When a primitive object is first created it contains no value (rather like opening a file) and it must be given a value in a separate operation. A value can be a scalar, a vector or an array. The elements of a vector or array must all be of the same type.

3.7 Illustration

To fix these ideas, look at the example below. Here we have described a structure of type 'NDF' using the following notation to describe each object:

```
NAME[(dimensions)] <TYPE> [value]
```

Note that scalar objects have no dimensions and that each level down the hierarchy is indented.

```
DATASET <NDF>
   DATA_ARRAY(512,1024)    <_UBYTE>   0,0,0,1,2,3,255,3,...
   LABEL                   <_CHAR*80> 'This is the data label'
   AXIS(2) <AXIS>
      AXIS <AXIS>
         DATA_ARRAY(512)   <_REAL>    0.5,1.5,2.5,...
         LABEL             <_CHAR*30> 'Axis 1'
      AXIS <AXIS>
         DATA_ARRAY(1024)  <_REAL>    5,10,15.1,20.3,...
         LABEL             <_CHAR*10> 'Axis 2'
```

This example exhibits several of the properties mentioned above:

- Both structures and primitives are present in the structure.
- Scalar and non-scalar objects are present.
- The AXIS component is a vector structure (with two elements).

4 ROUTINES AND CONSTANTS

4.1 Routine Names

HDS objects are created, accessed, modified and deleted in programs by means of calls to HDS routines. The routine names have the following structure:

```
<pkg>_<func><qual>
```

where <pkg> is the "package name", <func> represents the function performed by the routine and <qual> is a qualifier which is used to identify different versions of the GET, MAP, MOD, NEW and PUT routines.

4.2 Symbolic Names and Include Files

Stand-alone HDS programs will typically have the following structure:

```
<declarations>

INCLUDE 'SAE_PAR'
INCLUDE 'DAT_PAR'
INCLUDE 'DAT_ERR'
INCLUDE 'CMP_ERR'

STATUS = SAI__OK

<executable statements>

END
```

Various symbolic names should be used for important constant values in HDS programs to make the programs clear, portable and to insulate them from possible future changes. These symbolic names are defined by several Fortran "include" files. This explains the existence of the INCLUDE statements in the example above. The following include files are available:

SAE_PAR: This file is not actually part of HDS, but it defines the global symbolic constant SAI__OK (the value of the status return indicating success) and will be required by nearly all routines which call HDS. It should normally be included as a matter of course.

DAT_PAR: Defines various symbolic constants for HDS. These should be used whenever the associated value is required (typically this is when program variables are defined):

DAT__MXDIM	Maximum number of object dimensions
DAT__NOLOC	Null (invalid) locator value
DAT__NOWLD	Null wild-card search context
DAT__SZGRP	Size of group name
DAT__SZLOC	Size of locator
DAT__SZMOD	Size of access mode string
DAT__SZNAM	Size of object name
DAT__SZTYP	Size of type string

DAT_ERR: This defines symbolic names for the error status values returned by the DAT_ and HDS_ routines.

CMP_ERR: This defines symbolic names for the additional error status values returned by the CMP_ routines.

If it is required to test for specific error conditions, the appropriate include file should be used and the symbolic names (listed in Appendix E) used in the test. Here is an example of how to use these symbols:

```
      INCLUDE 'SAE_PAR'
      INCLUDE 'DAT_PAR'
      INCLUDE 'DAT_ERR'
      ...

      CHARACTER * ( DAT__SZLOC ) LOC1, LOC2
      CHARACTER * ( DAT__SZNAM ) NAME
      INTEGER STATUS
      ...

* Find a structure component.
      CALL DAT_FIND( LOC1, NAME, LOC2, STATUS )

* Check the status value returned.
      IF ( STATUS .EQ. SAI__OK ) THEN
         <normal action>
      ELSE IF ( STATUS .EQ. DAT__OBJNF ) THEN
         <take appropriate action for object not found>
      ELSE
         <action on other errors>
      END IF
```

5 CREATING OBJECTS

Here is an example showing how to create the NDF structure in §3.7.

```
      INCLUDE 'SAE_PAR'
      INCLUDE 'DAT_PAR'
      CHARACTER * ( DAT__SZLOC ) NLOC, ALOC, CELL
      INTEGER DIMS( 2 ), STATUS
      DATA DIMS / 512, 1024 /

*  Create a container file with a top level scalar object of type NDF.
      CALL HDS_NEW( 'dataset', 'DATASET', 'NDF', 0, 0, NLOC, STATUS )

*  Create components in the top level object.
      CALL DAT_NEW( NLOC, 'DATA_ARRAY', '_UBYTE', 2, DIMS, STATUS )
      CALL DAT_NEWC( NLOC, 'LABEL', 80, 0, 0, STATUS )
      CALL DAT_NEW( NLOC, 'AXIS', 'AXIS', 1, 2, STATUS )

*  Create components in the AXIS structure...

*  Get a locator to the AXIS component.
      CALL DAT_FIND( NLOC, 'AXIS', ALOC, STATUS )

*  Get a locator to the array cell AXIS(1).
      CALL DAT_CELL( ALOC, 1, 1, CELL, STATUS )

*  Create internal components within AXIS(1) using the CELL locator.
      CALL DAT_NEW( CELL, 'DATA_ARRAY', '_REAL', 1, DIM( 1 ), STATUS )
      CALL DAT_NEWC( CELL, 'LABEL', 30, 0, 0, STATUS )

*  Annul the cell locator
      CALL DAT_ANNUL( CELL, STATUS )

*  Do the same for AXIS(2).
      CALL DAT_CELL( ALOC, 1, 2, CELL, STATUS )
      CALL DAT_NEW( CELL, 'DATA_ARRAY', '_REAL', 1, DIM( 2 ), STATUS )
      CALL DAT_NEWC( CELL, 'LABEL', 10, 0, 0, STATUS )
      CALL DAT_ANNUL( CELL, STATUS )

*  Access objects which have been created.
      ...

*  Tidy up
      CALL DAT_ANNUL( ALOC, STATUS )
      CALL DAT_ANNUL( NLOC, STATUS )

      END
```

The following points should be borne in mind:

- The structure we have created is in no way static – we can add new objects or delete existing ones at any level without disturbing what already exists.

- No primitive values have been stored yet (*i.e.* the structure does not yet contain any data) – we will do that next.
- All new container files are created using the disk format indicated by the "VERSION" tuning parameter.

Here are some notes on particular aspects of this example:

DAT__SZLOC: This is an INTEGER constant which is defined in the include file DAT_PAR and specifies the length in characters of all HDS locators. Similar constants, DAT__SZNAM and DAT__SZTYP, specify the maximum lengths of object names and types.

STATUS: HDS routines conform to Starlink error handling conventions (described fully in SUN/104) and use *inherited status checking*. This normally means (unless otherwise noted in the routine description) that if the STATUS variable is not set to a value indicating success on entry (STATUS=SAI__OK), then the subroutine will exit without action. This behaviour can be very useful in reducing the effort required to check properly for all possible error conditions.

HDS_NEW: A container file called "dataset" is created (HDS provides the default file extension of '.sdf'). A scalar structure called DATASET with a type of 'NDF' is created within this file, and a locator (NLOC) is associated with this structure. It is usually convenient, although not essential, to make the top-level object name match the container file name, as here.

DAT_NEW/DAT_NEWC: These routines create new objects within HDS itself – they are not equivalent to HDS_NEW because they don't have any reference to the container file, only to a higher level structure.

We use two variants of this routine simply because the character string length has to be specified when creating a character object and it is normally most convenient to provide this via an additional integer argument. However, DAT_NEW may be used to create new objects of any type, including character objects. In this case the character string length would be provided via the type specification, *e.g.* '_CHAR*15' (a character string length of one is assumed if '_CHAR' is specified alone).

DAT_FIND: After an object has been created it is necessary to associate a locator with it before values can be inserted into it; DAT_FIND performs this function.

DAT_CELL: There are several routines for accessing subsets of objects. DAT_CELL obtains a locator to a scalar object (structure or primitive) within a non-scalar object like a vector.

DAT_ANNUL: This is used to release a locator (just as CLOSE releases a logical unit number in Fortran). It also closes the container file when it is no longer being used.

6 WRITING AND READING OBJECTS

Having created our structure we now want to put some values into it. This can be done by using the DAT_PUT and DAT_PUTC routines. For example, to fill the main data array in the above example with values we might do the following:

```
      ...
      BYTE IMVALS( 512, 1024 )
      CHARACTER * ( DAT__SZLOC ) LOC

*  Put data from array IMVALS to the object DATA_ARRAY.
      CALL DAT_FIND( NLOC, 'DATA_ARRAY', LOC, STATUS )
      CALL DAT_PUT( LOC, '_UBYTE', 2, DIMS, IMVALS, STATUS )
      CALL DAT_ANNUL( LOC, STATUS )

*  Put data from character constant to the object LABEL.
      CALL DAT_FIND( NLOC, 'LABEL', LOC, STATUS )
      CALL DAT_PUTC( LOC, 0, 0, 'This is the data label', STATUS )
      CALL DAT_ANNUL( LOC, STATUS )
      ...
```

Because this sort of activity occurs quite often, "packaged" access routines have been developed and are available to the programmer (see Appendix A). A complementary set of routines also exists for getting data from objects back into program arrays or variables; these are the DAT_GET routines. Again, packaged versions exist and are often handy in reducing the number of subroutine calls required.

7 ACCESSING OBJECTS BY MAPPING

Another technique for accessing values stored in primitive HDS objects is termed "mapping".[2] An important advantage of this technique is that it removes a size restriction imposed by having to declare fixed size program arrays to hold data. This simplifies software, so that a single routine can handle objects of arbitrary size without recourse to accessing subsets.

HDS provides mapped access to primitive objects via the DAT_MAP routines. Essentially DAT_MAP will return a pointer to a region of the computer's memory in which the object's values are stored. This pointer can then be passed to another routine using the (non-standard, but widely available) Fortran "%VAL" facility,[3] together with the CNF_PVAL function[4] (described in SUN/209) which is defined in the CNF_PAR include file. An example will illustrate this:

```
      ...
      INCLUDE 'CNF_PAR'
      INTEGER PNTR, EL
```

[2]This terminology derives from the facility provided by some operating systems for *mapping* the contents of files into the computer's memory, so that they appear as if they are arrays of numbers directly accessible to a program. Although HDS exploits this technique when available, other techniques can also be used to simulate this behaviour, so HDS does not depend on the operating system providing this facility.

[3]This technique works because Fortran normally passes the address of an array to a subroutine, so the routine is fooled into thinking it's getting an array.

[4]We illustrate the use of the CNF_PVAL function in this document, although in most existing software which calls HDS this function is not used. It has been introduced because of the possibility that Fortran software which stores pointers in 32-bit INTEGERs may need to execute in circumstances where 64 bits are required to store a memory pointer. The purpose of the CNF_PVAL function is then to expand the pointer value out to its full size before use, if necessary. On currently-supported platforms, this step is not normally needed, so use of CNF_PVAL may considered optional. However, its inclusion in new software is recommended as a useful precaution.

```
*  Map the DATA_ARRAY component of the NDF structure as a vector of type
*  _REAL (even though the object is actually a 512 x 1024 array whose
*  elements are of type _UBYTE).
      CALL DAT_FIND( NLOC, 'DATA_ARRAY', LOC, STATUS )
      CALL DAT_MAPV( LOC, '_REAL', 'UPDATE', PNTR, EL, STATUS )

*  Pass the "array" to a subroutine.
      CALL SUB( EL, %VAL( CNF_PVAL( PNTR ) ), STATUS )

*  Unmap the object and annul the locator.
      CALL DAT_UNMAP( LOC, STATUS )
      CALL DAT_ANNUL( LOC, STATUS )

      END

*  Routine which takes the LOG of all values in a REAL array.
      SUBROUTINE SUB( N, A, STATUS )
      INCLUDE 'SAE_PAR'
      INTEGER N, STATUS
      REAL A( N )
      IF ( STATUS .NE. SAI__OK ) RETURN

      DO 1 I = 1, N
         A( I ) = LOG( A( I ) )
 1    CONTINUE

      END
```

This example illustrates two features of HDS which we haven't already mentioned:

Vectorisation: It is possible to force HDS to regard objects as vectors, irrespective of their true shape. This facility was useful in the above example as it made the subroutine SUB much more general in that it could be applied to any numeric primitive object.

Automatic type conversion: The program can specify the data type it wishes to work with and the program will work even if the data is stored as a different type. HDS will (if necessary) automatically convert the data to the type required by the program.[5] This useful feature can greatly simplify programming – simple programs can handle all data types. Automatic conversion works on reading, writing and mapping.

Note that once a primitive has been mapped, the associated locator cannot be used to access further data until the original object is unmapped.

8 MAPPING CHARACTER DATA

Although the previous example used a numeric type of '_REAL' to access the data, HDS actually allows any primitive type to be specified as an access type, including '_CHAR'. In the particular

[5]This will work even if the object was originally created on a different computer which formats its numbers differently.

case of mapping data as character strings, however, there are some additional considerations imposed by the way that compilers handle this data type. These are described here.

HDS gives you a choice about how to determine the length of the character strings it will map. You may either specify the length you want explicitly, *e.g:*

```
CALL DAT_MAPV( LOC '_CHAR*30', 'READ', PNTR, EL, STATUS )
```

(in which case HDS would map an array of character strings with each string containing 30 characters) or you may leave HDS to decide on the length required by omitting the length specification, thus:

```
CALL DAT_MAPV( LOC '_CHAR', 'READ', PNTR, EL, STATUS )
```

In this latter case, HDS will determine the number of characters actually required to format the object's values without loss of information. It uses decimal strings for numerical values and the values 'TRUE' and 'FALSE' to represent logical values as character strings. If the object is already of character type, then its actual length will be used directly. The routine DAT_MAPC also operates in this manner.

On most UNIX systems, the length of a character string is passed to a subroutine by the compiler, which adds an additional "invisible" length argument to the end of the subroutine call for each character argument passed. Unfortunately, a compiler cannot recognise the data type of a mapped array, so it will not be able to automatically add length information about mapped character strings. This information must therefore be passed explicitly.

As an example, suppose we mapped an array of character strings specifying that each string should be of length 80 characters, thus:

```
CALL DAT_MAPV( LOC, '_CHAR*80', 'READ', PNTR, EL, STATUS )
```

This mapped array could be passed to a subroutine for use as follows:

```
INCLUDE 'CNF_PAR'

...

CALL SHOW( EL, %VAL( CNF_PVAL( PNTR ) ), %VAL( CNF_CVAL( 80 ) ) )
```

The character string length (80) is passed as an additional argument (which does not appear in the formal argument list of the SHOW subroutine) using the "%VAL" facility. The array would then be declared for use within this routine as if it were a normal Fortran character array, as follows:

```
SUBROUTINE SHOW( EL, CARRAY )
INTEGER EL
CHARACTER * ( * ) CARRAY( EL )
...
```

Its character string length could then be determined, as usual, by using the Fortran intrinsic LEN function within the SHOW routine.

Note that if an access mode of '_CHAR' were specified when mapping the array (leaving HDS to determine the length of the mapped strings) then an additional call to DAT_CLEN would be required to determine this length, as follows:

```
...
INTEGER LENGTH

CALL DAT_MAPV( LOC, '_CHAR', 'READ', PNTR, EL, STATUS )
CALL DAT_CLEN( LOC, LENGTH, STATUS )
CALL SHOW( EL, %VAL( CNF_PVAL( PNTR ) ), %VAL( CNF_CVAL( LENGTH ) ) )
```

If more than one mapped character array is being passed, then the length of each must be passed separately by adding it to the end of the argument list. This must be done in the same order as the arrays themselves are passed.

Unfortunately, this technique will only work if there are no other character values being passed at earlier points in the argument list. Otherwise, there is no way of adding additional length information so that it appears in the correct position relative to the "invisible" information that the compiler will already have generated for the other argument(s). The only solution in such cases is to provide a dummy routine whose purpose is simply to permute the argument order so that mapped character arrays may be passed at the start of the formal argument list.

9 COPYING AND DELETING OBJECTS

HDS objects may be individually copied and deleted. The routines DAT_COPY and DAT_ERASE will recursively copy and erase all levels of the hierarchy below that specified in the subroutine call:

```
      ...
      CHARACTER * ( DAT__SZLOC ) OLOC

*  Copy the AXIS structure to component AXISCOPY of the structure located
*  by OLOC (which must have been previously defined).
      CALL DAT_FIND( NLOC, 'AXIS', ALOC, STATUS )
      CALL DAT_COPY( ALOC, OLOC, 'AXISCOPY', STATUS )
      CALL DAT_ANNUL( ALOC, STATUS )

* Erase the original AXIS structure.
      CALL DAT_ERASE( NLOC, 'AXIS', STATUS )
```

Note that we annulled the locator to the AXIS object before attempting to delete it. This whole operation can also be done using DAT_MOVE:

```
CALL DAT_MOVE( ALOC, OLOC, 'AXISCOPY', STATUS )
```

10 SUBSETS OF OBJECTS

The routine DAT_CELL accesses a single element of an array (an example was shown in §5). The routine DAT_SLICE accesses a subset of an arbitrarily dimensioned object. This subset can then be treated for most purposes as if it were an object in its own right. For example:

```
      ...
      CHARACTER * ( DAT__SZLOC ) SLICE
      INTEGER LOWER( 2 ), UPPER( 2 )
      DATA LOWER / 100, 100 /
      DATA UPPER / 200, 200 /

*  Get a locator to the subset DATA_ARRAY(100:200,100:200).
      CALL DAT_FIND( NLOC, 'DATA_ARRAY', LOC, STATUS )
      CALL DAT_SLICE( LOC, 2, LOWER, UPPER, SLICE, STATUS )

*  Map the subset as a vector.
      CALL DAT_MAPV( SLICE, '_REAL', 'UPDATE', PNTR, EL, STATUS )
      ...
```

In contrast to DAT_SLICE, DAT_ALTER makes a permanent change to a non-scalar object. The object can be made larger or smaller, but only the last dimension can be ALTERed. This function is entirely dynamic, *i.e.* can be done at any time provided that the object is not mapped for access. Note that DAT_ALTER works on both primitives and structures. It is important to realise that the number of dimensions cannot be changed by DAT_ALTER.

11 TEMPORARY OBJECTS

Temporary objects of any type and shape may be created by using the DAT_TEMP routine. This returns a locator to the newly created object and this may then be manipulated just as if it were an ordinary object (in fact a temporary container file is created with a unique name to hold all such objects, and this is deleted when the program terminates, or when HDS_STOP is called, if this happens earlier). This is often useful for providing workspace for algorithms which may have to deal with large arrays.

12 USING LOCATORS

12.1 Locator Validity

As has been illustrated earlier, HDS refers to data objects by means of values held in character variables called *locators*. Of course, these character values are not HDS data objects themselves; they simply identify the data objects, whose internal details are hidden within the HDS system.

Each locator has a unique value which will not be re-used, and this property makes it possible to tell at any time whether a character value is a valid HDS locator or not. A locator's validity depends on a number of things, such as its actual value (the value DAT__NOLOC[6] is never valid for instance) and the previous history and current state of the HDS system (a locator which refers to a data object which has been deleted will no longer be valid). Note that locator values should not be explicitly altered by applications, as this may also cause them to become invalid.

[6] As defined in the include file DAT_PAR.

Locator validity can be determined by using the routine DAT_VALID, which returns a logical value of .TRUE. via its VALID argument if the locator supplied is valid:

```
CALL DAT_VALID( LOC, VALID, STATUS )
```

This is the only HDS routine to which an invalid locator may be passed without provoking an error.

12.2 Annulling Locators

The number of locators available at any time is quite large, but each locator consumes various computer resources which may be limited, so it is important to ensure that locators are *annulled* once they have been finished with, *i.e.* when access to the associated data object is no longer required.

Annulling an HDS locator renders it invalid and resets its value to DAT__NOLOC. It differs from simply setting the locator's variable to this value, however, because it ensures that all resources associated with it are released and made available for re-use. A locator is annulled using the routine DAT_ANNUL, as follows:

```
CALL DAT_ANNUL( LOC, STATUS )
```

Note that annulling an invalid locator will produce an error, but this will be ignored if the STATUS argument is not set to SAI__OK when DAT_ANNUL is called (*i.e.* indicating that a previous error has occurred). This means that it is not usually necessary to check whether a locator is valid before annulling it, so long as the only possible reason for it being invalid is a previous error which has set a STATUS value.

12.3 Cloning Locators

Since an HDS locator only refers to a data object and does not itself contain any data values, it is possible to have several locators referring to the same object. A duplicate locator for an HDS object may be derived from an existing one by a process called *cloning*, which is performed by the routine DAT_CLONE, as follows:

```
CALL DAT_CLONE( LOC1, LOC2, STATUS )
```

This returns a second locator LOC2 which refers to the same data object as LOC1.

Cloning is not required frequently, but it can occasionally be useful in allowing an application to "hold on" to a data object when a locator is passed to a routine which may annul it; *i.e.* you simply pass the original locator and keep the cloned copy.

12.4 Primary and Secondary Locators

Since data objects are stored in container files, HDS has to decide when to open and close these files (it would be very inefficient if a file had to be opened every time an object within it was referenced). To allow control over this, HDS locators are divided into two classes termed *primary* and *secondary*. It is primary locators that are responsible for holding container files open.

To be more specific, an HDS container file will remain open, so that data objects within it are accessible, for as long as there is at least one valid primary locator associated with it (that is, with one of the data objects within the file). Not surprisingly, those routines intended for opening container files (HDS_OPEN, HDS_NEW and HDS_WILD) will return primary locators – so that the file subsequently remains open. However, all other routines return secondary locators (with the exception of DAT_PRMRY, which may be used to manipulate this attribute – see §12.6).

12.5 Container File Reference Counts

The number of primary locators associated with an HDS container file is called its *reference count* and may be determined using the DAT_REFCT routine as follows:

```
CALL DAT_REFCT( LOC, REFCT, STATUS )
```

Here, LOC is a locator associated with any object in the file and the reference count is returned via the integer REFCT argument. The file will remain open for as long as this value is greater than zero.

Normally, a file's reference count will fall to zero due to annulling the last primary locator associated with it (usually the locator obtained when the file was originally opened), and at this point the file will be closed. Before this happens, however, any mapped primitive objects within it will be unmapped. In addition, any secondary locators that remain associated with data objects in the same file will be annulled (*i.e.* they will become invalid).[7] No further reference to objects within the file may be made until it has been explicitly re-opened.

12.6 Promoting and Demoting Locators

A locator may, at any time, be "promoted" to become a primary locator (thus incrementing the container file's reference count) or "demoted" to become a secondary locator (and decrementing the reference count). This is done by using the DAT_PRMRY routine with its (first) SET argument set to .TRUE., thus:

```
PRMRY = .TRUE.              ! Promote the locator
CALL DAT_PRMRY( .TRUE., LOC, PRMRY, STATUS )
```

With its first argument set to .FALSE., the same routine may also be used to enquire whether a locator is primary or not.

The main reason for promoting locators is to allow HDS objects to be passed between routines while ensuring that the associated container file remains open, so that the object remains accessible. For example, consider the following simple routine which returns a locator for a named object inside a container file:

```
      SUBROUTINE FINDIT( FILE, LOC, STATUS )

      <declarations, etc.>
```

[7] You are advised not to depend on this mechanism for annulling secondary locators because you will not normally have complete control over a file's reference count (for instance, it may be opened independently for some other purpose in the same piece of software).

```
*  Open the container file and find the required object.
      CALL HDS_OPEN( FILE, 'READ', TMPLOC, STATUS )
      CALL DAT_FIND( TMPLOC, 'MY_OBJECT', LOC, STATUS )

*  Promote the new locator and annul the original.
      CALL DAT_PRMRY( .TRUE., LOC, .TRUE., STATUS )
      CALL DAT_ANNUL( LOCTMP, STATUS )
      END
```

Note how the temporary locator returned by HDS_OPEN is annulled after first promoting the secondary locator derived from it, so that the container file remains open. If this is the first time the file has been opened, its reference count will be 1 when this routine exits, so it will be closed when the caller later annuls the returned locator LOC.

13 ENQUIRIES

One of the most important properties of HDS is that its data files are self describing. This means that each object carries with it information describing all its attributes (not just its value), and these attributes can be obtained by means of enquiry routines. An example will illustrate:

```
      ...
      PARAMETER ( MAXCMP = 10 )
      CHARACTER * ( DAT__SZNAM ) NAME( MAXCMP )
      CHARACTER * ( DAT__SZTYP ) TYPE( MAXCMP )
      INTEGER NCOMP, I
      LOGICAL PRIM( MAXCMP )

*  Enquire the names and types of up to MAXCMP components...

*  First store the total number of components.
      CALL DAT_NCOMP( NLOC, NCOMP, STATUS)

*  Now index through the structure's components, obtaining locators and the
*  required information.
      DO 1 I = 1, MIN( NCOMP, MAXCMP )

*  Get a locator to the I'th component.
         CALL DAT_INDEX( NLOC, I, LOC, STATUS )

* Obtain its name and type.
         CALL DAT_NAME( LOC, NAME( I ), STATUS )
         CALL DAT_TYPE( LOC, TYPE( I ), STATUS )

* Is it primitive?
         CALL DAT_PRIM( LOC, PRIM( I ), STATUS )
         CALL DAT_ANNUL( LOC, STATUS )
 1    CONTINUE
      ...
```

Here DAT_INDEX is used to get locators to objects about which (in principle) we know nothing. This is just like listing the files in a directory, except that the order in which the components are stored in an HDS structure is arbitrary so they won't necessarily be accessed in alphabetical order.

14 PACKAGED ROUTINES

HDS provides families of routines which provide a more convenient method of access to objects than the basic routines. For instance DAT_PUTnx and DAT_GETnx package DAT_PUT and DAT_GET by dimensionality and type. Thus DAT_PUT0I will write a single INTEGER value to a scalar primitive. DAT_GET1R will read the value of a vector primitive and store it in a REAL program array. There are no DAT_GET2x routines; all dimensionalities higher than one are handled by DAT_GETNx and DAT_PUTNx.

Another family of routines are the CMP_ routines. These access components of the "current level". This usually involves:

- FINDing the required object and getting a locator to it.
- Performing the required operation, *e.g.* PUTting some value into it.
- ANNULling the locator.

The CMP_ routines package this sort of operation, replacing three or so subroutine calls with one. The naming scheme is based on the associated DAT_ routines. An example is shown below.

```
      ...
      CHARACTER * 80 DLAB
      INTEGER DIMS( 2 )
      REAL IMVALS( 512, 1024 )
      DATA DIMS / 512, 1024 /

*  Get REAL values from the DATA_ARRAY component.
      CALL CMP_GETNR( NLOC, 'DATA_ARRAY', 2, DIMS, IMVALS, DIMS, STATUS )

*  Get a character string from the LABEL component and store it in DLAB.
      CALL CMP_GET0C( NLOC, 'LABEL', DLAB, STATUS )
      ...
```

15 TUNING

15.1 Setting HDS Tuning Parameters

HDS has a number of internal integer *tuning parameters* whose values control various aspects of its behaviour and some of which may have an important effect on its performance. Each of these parameters has a default value, but may be over-ridden by either of two mechanisms.

Environment Variables.

By defining appropriate environment variables it is possible to set new default values for HDS tuning parameters. The translation of these environment variables is picked up when HDS starts up (typically when the first HDS routine is called) and an attempt is then made to interpret the resulting value as an integer. If successful, the default value of the tuning parameter is set to the new value. If not, it remains unchanged. HDS applies sensible constraints to any new values supplied.

For example, the UNIX C shell environment variable definition:

```
setenv HDS_INALQ 10
```

could be used to increase the default size of all newly created HDS files to 10 blocks.[8]

The name of the environment variable is constructed by prefixing the string 'HDS_' to the tuning parameter name. All such environment variables must be specified using upper case.

It should be recognised that this ability to set tuning parameter values via environment variables can be dangerous. It is provided mainly to encourage experimentation and to overcome "one-off" tuning problems, but it carries a risk of disrupting normal program behaviour. In particular, you should not expect that all HDS programs will necessarily continue to work with all possible settings of their tuning parameters, and software developers are urged not to write programs which depend on non-default settings of HDS tuning parameters, as this may give rise to conflicts with other software. If a tuning parameter setting really is critical, then it should be set by the software itself (see below), so as to prohibit outside interference.

Calling HDS_TUNE.

Tuning parameter values may also be set directly from within an item of software by means of calls to the routine HDS_TUNE. This allows programs to over-ride the default settings (or those established *via* environment variables). To modify the 'MAP' tuning parameter, for example, the following call might be used:

```
CALL HDS_TUNE( 'MAP', 0, STATUS )
```

This would have the effect of disabling file mapping in favour of reading and writing as the preferred method of accessing data in container files. The related routine HDS_GTUNE may be used to determine the current setting of a tuning parameter (see §15.3 for an example of its use).

15.2 Tuning Parameters Available

HDS currently uses the following tuning parameters to control its behaviour.

Parameters which control the top level HDS library

VERSION - Data Format for New Files:

This determines the data format that is used when a new container file is created. It may

[8] An HDS block is 512 bytes.

take the value 3, 4 or 5, and currently defaults to 4 (although this will change to 5 in a future release). Version 4 is a Starlink-specific data format that is a development of the original HDS data format created by the Starlink project in the 1980's. Version 5 uses the HDF5 data format on disk to mimic the facilities of version 4. Such files cannot be accessed by versions of the HDS library prior to version 6, but can be examined using various publicly available HDF5 tools - see section "DISK FORMATS AND HDF5". Version 3 refers to the Starlink-specific data format as it was prior to the introduction of 64-bit mode.

VERSION=5:
Use the HDS data format 5 library, which is based on HDF5.

VERSION=4:
Use the HDS data format 4 library with its "`64BIT`" tuning parameter set to 1.

VERSION=3:
Use the same HDS library as data format 4, but with its "`64BIT`" tuning parameter set to 0.

V4LOCKERROR - controls error reporting:
If non-zero, an error is reported if a thread lock function (datLock, datUnlock or datLocked) is used on a locator for an object stored using disk format version 4. Otherwise, the function returns without action. The default is to return without action. This facility is intended for debugging purposes.

Parameters used with both HDS version 4 and version 5 files

MAP - Use file mapping if available?
This value controls the method by which HDS performs I/O operations on the values of primitive objects and may take the following values:

MAP=1:
Use "file mapping" (if supported) as the preferred method of accessing primitive data.

MAP=0:
Use read/write operations (if supported) as the preferred data access method.

MAP= −1:
Use whichever method is normally faster for sequential access to all elements of a large array of data.

MAP= −2:
Use whichever method is normally faster for sparse random access to a large array of data.

MAP= −3:
Use whichever method normally makes the smaller demand on system memory resources (normally this means a request to minimise use of address space or swap file space, but the precise interpretation is operating system dependent). This is normally the appropriate option if you intend to use HDS arrays as temporary workspace.

HDS converts all other values to one. The value may be changed at any time.

A subsequent call to HDS_GTUNE, specifying the 'MAP' tuning parameter, will return 0 or 1 to indicate which option was actually chosen. This may depend on the capabilities of the host operating system and the particular implementation of HDS in use. The default value for this tuning parameter is also system dependent (see §F.3).

Typically, file mapping has the following plus and minus points:

\+ It allows large arrays accessed via the HDS mapping routines to be sparsely accessed in an efficient way. In this case, only those regions of the array actually accessed will need to be read/written, as opposed to reading the entire array just to access a small fraction of it. This might be useful, for instance, if a 1-dimensional profile through a large image were being generated.

\+ It allows HDS container files to act as "backing store" for the virtual memory associated with objects accessed via the mapping routines. The operating system can then use HDS files, rather than its own backing (swap) file, to implement virtual memory management. This means that you do not need to have a large system backing file available in order to access large datasets.

\+ For the same reason, temporary objects created with DAT_TEMP and mapped to provide temporary workspace make no additional demand on the system backing file.

? On some operating systems file mapping may be less efficient in terms of elapsed time than direct read/write operations. Conversely, on some operating systems it may be more efficient.

\− Despite the memory efficiency of file mapping, there may be a significant efficiency penalty when large arrays are mapped to provide workspace. This is because the scratch data will often be written back to the container file when the array is unmapped (despite the fact that the file is about to be deleted). This can take a considerable time and cannot be prevented as the operating system has control over this process.

Unfortunately, on some operating systems, this process appears to occur even when normal system calls are used to allocate memory because file mapping is used implicitly. In this case, HDS's file mapping is at no particular disadvantage.

\− Not all operating systems support file mapping and it generally requires system-specific programming techniques, making it more trouble to implement on a new operating system.

Using read/write access has the following advantages and disadvantages:

\+**?** On some operating systems it may be more efficient than file mapping in terms of elapsed time in cases where an array of data will be accessed in its entirety (the normal situation). This is generally not true of UNIX systems, however,

\− It is an inefficient method of accessing a small subset of a large array because it requires the entire array to be read/written. The solution to this problem is to explicitly access the required subset using (*e.g.*) DAT_SLICE, although this complicates the software somewhat.

– It makes demands on the operating system's backing file which the file mapping technique avoids (see above). As a result, there is little point in creating scratch arrays with DAT_TEMP for use as workspace unless file mapping is available (because the system backing file will be used anyway).

? If an object is accessed several times simultaneously using HDS mapping routines, then modifications made via one mapping may not be consistently reflected in the other mapping (modifications will only be updated in the container file when the object is unmapped, so the two mappings may get out of step in the mean time). Conversely, if file mapping is in use and a primitive object is mapped in its entirety without type conversion, then this behaviour does not occur (all mappings remain consistent). It may occur, however, if a slice is being accessed or if type conversion is needed.

It is debatable which behaviour is preferable. The best policy is to avoid the problem entirely by not utilising multiple access to the same object while modifications are being made.

SHELL - Preferred shell:

This parameter determines which UNIX shell should be used to interpret container file names which contain "special" characters representing pattern-matching, environment variable substitution, *etc.* Each shell typically has its own particular way of interpreting these characters, so users of HDS may wish to select the same shell as they normally use for entering commands. The following values are allowed:

SHELL=2:
Use the "tcsh" shell (if available). If this is not available, then use the same shell as when SHELL=1.

SHELL=1:
Use the "csh" shell (C shell on traditional UNIX systems). If this is not available, then use the same shell as when SHELL=0.

SHELL=0 (the default):
Use the "sh" shell. This normally means the Bourne Shell on traditional UNIX systems, but on systems which support it, the similar POSIX "sh" shell may be used instead.

SHELL= −1:
Don't use any shell for interpreting single file names (all special characters are to be interpreted literally). When performing "wild-card" searches for multiple files (with HDS_WILD), use the same shell as when SHELL=0.

HDS converts all other values to zero.

Parameters used only with HDS version 4 files

INALQ - Initial File Allocation Quantity:

This value determines how many blocks[9] are to be allocated when a new container file is created. The default value of 2 is the minimum value allowed; the first block contains header information and the second contains the top-level object. Note that the host

[9] An HDS block is 512 bytes.

operating system may impose further restrictions on allowable file sizes, so the actual size of a file may not match the value specified exactly.

The value of this parameter reverts to its default value (or the value specified by the HDS_INALQ environment variable) after each file is created, so if it is being set from within a program, it must be set every time that it is required.

If a file is to be extended frequently (through the creation of new objects within it), then this parameter may provide a worthwhile efficiency gain by allowing a file of a suitable size to be created initially. On most UNIX systems, however, the benefits are minimal.

MAXWPL - Maximum Size of the "Working Page List":

This value specifies how many blocks are to be allocated to the memory cache which HDS uses to hold information about the structure of HDS files and objects and to buffer its I/O operations when obtaining this information. The default value is 32 blocks; this value cannot be decreased. Modifications to this value will only have an effect if made before HDS becomes active (*i.e.* before any call is made to another HDS routine).

There will not normally be any need to increase this value unless excessively complex data structures are being accessed with very large numbers of locators simultaneously active.

NBLOCKS - Size of the internal "Transfer Buffer":

When HDS has to move large quantities of data from one location to another, it often has to store an intermediate result. In such cases, rather than allocate a large buffer to hold all the intermediate data, it uses a smaller buffer and performs the transfer in pieces. This parameter specifies the maximum size in blocks which this transfer buffer may have and is constrained to be no less than the default, which is 32 blocks.

The value should not be too small, or excessive time will be spent in loops which repeatedly refill the buffer. Conversely, too large a value will make excessive demands on memory. In practice there is a wide range of acceptable values, so this tuning parameter will almost never need to be altered.

NCOMP - Optimum number of structure components:

This value may be used to specify the expected number of components which will be stored in an HDS structure. HDS does not limit the number of structure components, but when a structure is first created, space is set aside for creation of components in future. If more than the expected number of components are subsequently created, then HDS must eventually re-organise part of the container file to obtain the space needed. Conversely, if fewer components are created, then some space in the file will remain unused. The value is constrained to be at least one, the default being 6 components.

The value of this parameter is used during the creation of the first component in every new structure. It reverts to its default value (or the value specified by the HDS_NCOMP environment variable) afterwards, so if it is being set from within a program, it must be set every time it is needed.

SYSLCK - System wide lock flag:

This parameter is present for historical reasons and has no effect on UNIX systems.

WAIT - Wait for locked files?

This parameter is present for historical reasons and currently has no effect on UNIX systems, where HDS file locking is not implemented.

64BIT - Use 64-bit (HDS version 4) files?

This value can be used to select whether new files are created in the 64-bit (HDS version 4) format. If 64BIT=0 then files are created in the previous (HDS version 3) format.

This parameter is normally overriden by the VERSION parameter of the top level library.

15.3 Tuning in Practice

Normally, a single application which wished to tune HDS itself (rather than accepting the default settings, or those specified by environment variables) would do so via calls to HDS_TUNE at the start, and would thus establish a default "tuning profile" to apply throughout the rest of the program. Similarly, a software environment can initially tune HDS to obtain the required default behaviour for the applications it will later invoke.

Sometimes, however, it may be necessary to modify a tuning parameter to improve performance locally while not affecting behaviour of other parts of a program (or other applications in a software environment). The routine HDS_GTUNE may therefore be used to determine the current setting of an HDS tuning parameter, so that it may later be returned to its original value. For instance, if the 'MAP' parameter were to be set locally to allow sparse access to a large array of data, the following technique might be used:

```
      ...
      INTEGER OLDMAP

*  Obtain the original setting of the MAP parameter.
      CALL HDS_GTUNE( 'MAP', OLDMAP, STATUS )
      IF ( STATUS .EQ. SAI__OK ) THEN

*  Set a new value.
         CALL HDS_TUNE( 'MAP', -2, STATUS )

         <map the array>

*  Return to the old tuning setting.
         CALL ERR_BEGIN( STATUS )
         CALL HDS_TUNE( 'MAP', OLDMAP, STATUS )
         CALL ERR_END( STATUS )
      END IF
```

Notice how great care has been taken over handling error conditions. In a large software system it could prove disastrous if a tuning parameter remained set to an incorrect value (perhaps causing gross inefficiencies elsewhere) simply because HDS_TUNE did not execute after an unexpected error had caused STATUS to be set to an error value.

16 COMPILING AND LINKING

Standalone applications which use HDS may be linked by specifying 'hds_link' on the compiler command line. Thus, to compile and link a stand-alone application called "prog", the following might be used:

```
f77 prog.f -I$STARLINK_DIR/include -L$STARLINK_DIR/lib ‘hds_link‘ -o prog
```

Note the use of the "−L" flag to specify the location of the Starlink libraries and the backward quote characters which perform command substitution. Include files are located using the "−I" flag.

Users of the ADAM programming environment (SG/4) on UNIX systems need take no special steps in order to link with HDS because the normal commands for building ADAM tasks will do this automatically. Thus, for instance, an A-task which calls HDS routines might be built simply as follows:

```
alink prog.f
```

If you want to compile subroutine source code without linking, you should use the "-I" flag as for the standalone example:

```
f77 -c -I$STARLINK_DIR/include subroutine.f
```

17 ACKNOWLEDGEMENTS

The routines historically distributed as HDS and which are still used to access HDS version 4 files, derive from two original sources. The first is the initial implementation written by Dave Pearce and Anton Walter at RAL for the VAX/VMS operating system (initially using the BLISS language), which was subsequently converted into C by Mike Lawden (RAL) and William Lupton (RGO and AAO). The second source is a collection of higher-level routines mostly written at University College London in Fortran 77 by Sid Wright and Jack Giddings. William Lupton subsequently performed some of the initial work aimed at making HDS more portable.

HDS then underwent considerable internal change, including substantial re-coding to produce a fully portable system based on the POSIX operating system interface - work done by Rodney Warren-Smith at RAL.

More recently, Tim Jenness (Cornell University) has re-implemented the HDS API on top of the HDF5 library to create the current (version 6) library.

A ALPHABETICAL LIST OF HDS ROUTINES

CMP_GET0x*(loc,name;value,status)* – Read scalar component
CMP_GET1x*(loc,name,elx;value,el,status)* – Read vector component
CMP_GETNx*(loc,name,ndim,dimx;value,dim,status)* – Read array component
CMP_GETVx*(loc,name,elx;value,el,status)* – Read vectorised component
CMP_LEN*(loc,name;len,status)* – Enquire component precision
CMP_MAPN*(loc,name,type,mode,ndim;pntr,dim,status)* – Map array component
CMP_MAPV*(loc,name,type,mode;pntr,el,status)* – Map vectorised component
CMP_MOD*(loc,name,type,ndim,dim;status)* – Obtain component
CMP_MODC*(loc,name,len,ndim,dim;status)* – Obtain string component
CMP_PRIM*(loc,name;reply,status)* – Enquire component primitive
CMP_PUT0x*(loc,name,value;status)* – Write scalar component
CMP_PUT1x*(loc,name,el,value;status)* – Write vector component
CMP_PUTNx*(loc,name,ndim,dimx,value,dim;status)* – Write array component
CMP_PUTVx*(loc,name,el,value;status)* – Write vectorised component
CMP_SHAPE*(loc,name,ndimx;dim,ndim,status)* – Enquire component shape
CMP_SIZE*(loc,name;size,status)* – Enquire component size
CMP_STRUC*(loc,name;reply,status)* – Enquire component structure
CMP_TYPE*(loc,name;type,status)* – Enquire component type
CMP_UNMAP*(loc,name;status)* – Unmap component

DAT_ALTER*(loc,ndim,dim;status)* – Alter object size
DAT_ANNUL*(loc;status)* – Annul locator
DAT_BASIC*(loc,mode;pntr,len,status)* – Map primitive as basic units
DAT_CCOPY*(loc1,loc2,name;loc3,status)* – Copy one structure level
DAT_CCTYP*(size;type)* – Create type string
DAT_CELL*(loc1,ndim,sub;loc2,status)* – Locate cell
DAT_CLEN*(loc;clen,status)* – Enquire character string length
DAT_CLONE*(loc1;loc2,status)* – Clone locator
DAT_COERC*(loc1,ndim;loc2,status)* – Coerce object shape
DAT_COPY*(loc1,loc2,name;status)* – Copy object
DAT_DREP*(loc;format,order,status)* – Obtain primitive data representation information
DAT_ERASE*(loc,name;status)* – Erase component
DAT_ERMSG*(status;length,msg)* – Translate error status
DAT_FIND*(loc1,name;loc2,status)* – Find named component
DAT_GET*(loc,type,ndim,dim;value,status)* – Read primitive
DAT_GETx*(loc,ndim,dim;value,status)* – Read primitive
DAT_GET0x*(loc;value,status)* – Read scalar primitive
DAT_GET1x*(loc,elx;value,el,status)* – Read vector primitive
DAT_GETNx*(loc,ndim,dimx;value,dim,status)* – Read array primitive
DAT_GETVx*(loc,elx;value,el,status)* – Read vectorised primitive
DAT_INDEX*(loc1,index;loc2,status)* – Index into component list
DAT_LEN*(loc;len,status)* – Enquire primitive precision
DAT_MAP*(loc,type,mode,ndim,dim;pntr,status)* – Map primitive
DAT_MAPx*(loc,mode,ndim,dim;pntr,status)* – Map primitive
DAT_MAPN*(loc,type,mode,ndim;pntr,dim,status)* – Map array primitive
DAT_MAPV*(loc,type,mode;pntr,el,status)* – Map vectorised primitive
DAT_MOULD*(loc,ndim,dim;status)* – Alter object shape
DAT_MOVE*(loc1,loc2,name;status)* – Move object
DAT_MSG*(token,loc)* – Assign object name to message token
DAT_NAME*(loc;name,status)* – Enquire object name

DAT_NCOMP*(loc;ncomp,status)* – Enquire number of components
DAT_NEW*(loc,name,type,ndim,dim;status)* – Create component
DAT_NEW0x*(loc,name;status)* – Create scalar component
DAT_NEW0C*(loc,name,len;status)* – Create scalar string component
DAT_NEW1x*(loc,name,el;status)* – Create vector component
DAT_NEW1C*(loc,name,len,el;status)* – Create vector string component
DAT_NEWC*(loc,name,len,ndim,dim;status)* – Create string component
DAT_PAREN*(loc1;loc2,status)* – Find parent
DAT_PREC*(loc;nbyte,status)* – Enquire storage precision
DAT_PRIM*(loc;reply,status)* – Enquire object primitive
DAT_PRMRY*(set;loc,prmry,status)* – Set or enquire primary/secondary locator
DAT_PUT*(loc,type,ndim,dim,value;status)* – Write primitive
DAT_PUTx*(loc,ndim,dim,value;status)* – Write primitive
DAT_PUT0x*(loc,value;status)* – Write scalar primitive
DAT_PUT1x*(loc,el,value;status)* – Write vector primitive
DAT_PUTNx*(loc,ndim,dimx,value,dim;status)* – Write array primitive
DAT_PUTVx*(loc,el,value;status)* – Write vectorised primitive
DAT_REF*(loc;ref,lref,status)* – Obtain reference name for object
DAT_REFCT*(loc;refct,status)* – Enquire the reference count for a container file
DAT_RENAM*(loc,name;status)* – Rename object
DAT_RESET*(loc;status)* – Reset object state
DAT_RETYP*(loc,type;status)* – Change object type
DAT_SHAPE*(loc,ndimx;dim,ndim,status)* – Enquire object shape
DAT_SIZE*(loc;size,status)* – Enquire object size
DAT_SLICE*(loc1,ndim,diml,dimu;loc2,status)* – Locate slice
DAT_STATE*(loc;reply,status)* – Enquire object state
DAT_STRUC*(loc;reply,status)* – Enquire object structure
DAT_TEMP*(type,ndim,dim;loc,status)* – Create temporary object
DAT_THERE*(loc,name;reply,status)* – Enquire component existence
DAT_TYPE*(loc;type,status)* – Enquire object type
DAT_UNMAP*(loc;status)* – Unmap object
DAT_VALID*(loc;reply,status)* – Enquire locator valid
DAT_VEC*(loc1;loc2,status)* – Vectorise object
DAT_WHERE*(loc;block,offset,status)* – Where is primitive data in file?

HDS_COPY*(loc,file,name;status)* – Copy object to container file
HDS_ERASE*(loc;status)* – Erase container file
HDS_EWILD*(iwld,status)* – End a wild-card search for HDS container files
HDS_FLUSH*(group;status)* – Flush locator group
HDS_FREE*(loc;status)* – Free container file
HDS_GROUP*(loc;group,status)* – Enquire locator group
HDS_GTUNE*(param;value,status)* – Enquire value of tuning parameter
HDS_LINK*(loc,group;status)* – Link locator group
HDS_LOCK*(loc;status)* – Lock container file
HDS_NEW*(file,name,type,ndim,dim;loc,status)* – Create container file
HDS_OPEN*(file,mode;loc,status)* – Open container file
HDS_SHOW*(topic;status)* – Show HDS statistics
HDS_STATE*(state,status)* – Enquire HDS state
HDS_STOP*(status)* – Close down HDS
HDS_TRACE*(loc;nlev,path,file,status)* – Trace object path
HDS_TUNE*(param,value;status)* – Set HDS parameter
HDS_WILD*(fspec,mode;iwld,loc,status)* – Perform a wild-card search for HDS container files

B ROUTINE DESCRIPTIONS

This appendix gives specifications for all the stand-alone HDS routines. Some general information on the most common parameters is given below:

- *dim, dimx, ndim, ndimx:* Parameters *dim* and *ndim* specify the shape of an object; *dimx* and *ndimx* specify the largest permitted values of *dim* and *ndim*. The vector *dim* holds the sizes of the object dimensions. Thus *dim(1)* holds the size of the first dimension, *dim(2)* holds the size of the second dimension, and so on. The integer *ndim* holds the number of dimensions in an object. HDS supports a maximum of seven object dimensions, thus *dim* should have a maximum of seven elements and *ndim* should not be more than 7. The values of *dim* and *ndim* must match the actual shape of the object being processed. A value of zero for *ndim* denotes a scalar object; any value specified for *dim* will be ignored in this case. A vector containing a single value is different from a scalar.

- *el, elx:* When an object is a vector (or is treated as one), *el* holds the number of elements in the vector. When a vector is being read (as in the GET routines), the actual number of elements may be unknown. In this case *elx* holds the maximum permissible number of elements. This should be equal to the size of the *value* array which is to receive the vector.

- *loc, loc1, loc2, loc3:* A locator is a CHARACTER∗(DAT__SZLOC) variable, substring or array element used to "locate" objects within the data system. The contents are used by HDS to access internal information and must not be altered explicitly by a program. When more than one routine parameter is a locator, *loc1*, *loc2* and *loc3* are used to identify them. The locator for a structure array refers to the array as a whole. Each element of a structure array is itself a structure If you wish to address an element of a structure array you must obtain a new locator for it (using DAT_CELL). It is important to distinguish between routines which operate on structures and those which operate on arrays.

- *name:* This is a character value specifying the name of an object. A name is written as a character string containing any printing characters. White space is ignored and alphabetic characters are capitalised. There are no special rules governing the first character (*i.e.* it can be numeric).

- *pntr:* Routines which map an object value return a pointer to the first element of that object in the parameter *pntr*. This can be converted to a normal array reference by use of the non-standard Fortran "%VAL" facility together with the CNF_PVAL function as shown in §7 and §8.

- *status:* This receives the HDS status value. If, on entry, a routine finds that *status* is not equal to SAI__OK it assumes an error has occurred previously and returns immediately without action. This allows tests of *status* to be deferred until after several routine calls have been made. If a routine detects an error itself, it will set *status* to one of the error codes specified in Appendix E and will make an error report in the usual manner (see SUN/104). Where exceptions to this behaviour exist, they are documented in the individual routine descriptions.

- *type:* This specifies the data type the program wishes to use when manipulating a value. This may be different from the data type used to store the value. We will call the type used

by the program the *access type* and the type used to store the value the *storage type*. If these two types differ, automatic conversion is performed. This is particularly relevant to the GET, PUT and MAP routines. The types specified in the names or the *type* parameter of these routines refer to the access type, thus CMP_GET0I will read a scalar component and present it to the program as an integer, no matter how that value is stored in the container file. Thus, *type* means *access type*; it also specifies the *storage type* in routines which write a value.

- *value:* This holds the value of a primitive and could be a scalar, vector or an array. If a value is being read, the size and shape of the vector or array *value* should match the shape of the object. In some routines *value* holds a "vectorised" object value. This means that the value will be addressed as a linear sequence of elements instead of as its normal structure. The effect of this depends on how a structure is mapped. The purpose of vectorising an array is to enable a simple operation (*e.g.* adding a constant) to be carried out on every element without bothering about how each element should be addressed

CMP_GET0x
Read scalar component

Description:

Read a scalar primitive component of a structure.

Invocation:

```
CALL CMP_GET0x( LOC, NAME, VALUE, STATUS )
```

Arguments:

LOC = CHARACTER * (DAT__SZLOC) (Given)
Structure locator.

NAME = CHARACTER * (DAT__SZNAM) (Given)
Component name.

VALUE = ? (Returned)
Component value.

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

Notes:

There is a routine for each standard data type; to call them, replace x by C (Character), D (Double), I (Integer), L (Logical), R (Real).

CMP_GET1x
Read vector component

Description:

Read a vector primitive component of a structure.

Invocation:

```
CALL CMP_GET1x( LOC, NAME, ELX, VALUE, EL, STATUS )
```

Arguments:

LOC = CHARACTER ∗ (DAT__SZLOC) (Given)
Structure locator.

NAME = CHARACTER ∗ (DAT__SZNAM) (Given)
Component name.

ELX = INTEGER (Given)
Maximum size of value.

VALUE = ?(ELX) (Returned)
Component value.

EL = INTEGER (Returned)
Number of elements.

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

Notes:

There is a routine for each standard data type; to call them, replace x by C (Character), D (Double), I (Integer), L (Logical), R (Real).

CMP_GETNx
Read array component

Description:

Read an array primitive component of a structure.

Invocation:

```
CALL CMP_GETNx( LOC, NAME, NDIM, DIMX, VALUE, DIM, STATUS )
```

Arguments:

LOC = CHARACTER * (DAT__SZLOC) (Given)
Structure locator.

NAME = CHARACTER * (DAT__SZNAM) (Given)
Component name.

NDIM = INTEGER (Given)
Number of dimensions.

DIMX = INTEGER(NDIM) (Given)
Dimensions of value.

VALUE = ?(<dimx(1)>, <dimx(2)>, ...) (Returned)
Component value.

DIM = INTEGER(NDIM) (Returned)
Component dimensions.

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

Notes:

There is a routine for each standard data type; to call them, replace x by C (Character), D (Double), I (Integer), L (Logical), R (Real).

CMP_GETVx
Read vectorised component

Description:

Read a primitive component of a structure as if vectorised.

Invocation:

```
CALL CMP_GETVx( LOC, NAME, ELX, VALUE, EL, STATUS )
```

Arguments:

LOC = CHARACTER * (DAT__SZLOC) (Given)
Structure locator.

NAME = CHARACTER * (DAT__SZNAM) (Given)
Component name.

ELX = INTEGER (Given)
Maximum size of value.

VALUE = ?(ELX) (Returned)
Component value.

EL = INTEGER (Returned)
Number of elements.

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

Notes:

There is a routine for each standard data type; to call them, replace x by C (Character), D (Double), I (Integer), L (Logical), R (Real).

CMP_LEN
Enquire component precision

Description:

Enquire the number of bytes per element of a primitive component.

Invocation:

```
CALL CMP_LEN( LOC, NAME, LEN, STATUS )
```

Arguments:

LOC = CHARACTER * (DAT__SZLOC) (Given)
Structure locator.

NAME = CHARACTER * (DAT__SZNAM) (Given)
Component name.

LEN = INTEGER (Returned)
Number of bytes per element.

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

CMP_MAPN
Map array component

Description:

Map a primitive component of a structure as an array.

Invocation:

```
CALL CMP_MAPN( LOC, NAME, TYPE, MODE, NDIM, PNTR, DIM, STATUS )
```

Arguments:

LOC = CHARACTER * (DAT__SZLOC) (Given)
Structure locator.

NAME = CHARACTER * (DAT__SZNAM) (Given)
Component name.

TYPE = CHARACTER * (DAT__SZTYP) (Given)
Data type.

MODE = CHARACTER * (DAT__SZMOD) (Given)
Access mode ('READ', 'UPDATE' or 'WRITE').

NDIM = INTEGER (Given)
Number of dimensions.

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

DIM = INTEGER(NDIM) (Returned)
Component dimensions.

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

CMP_MAPV
Map vectorised component

Description:

Map a primitive component of a structure as if vectorised.

Invocation:

```
CALL CMP_MAPV( LOC, NAME, TYPE, MODE, PNTR, EL, STATUS )
```

Arguments:

LOC = CHARACTER * (DAT__SZLOC) (Given)
Structure locator.

NAME = CHARACTER * (DAT__SZNAM) (Given)
Component name.

TYPE = CHARACTER * (DAT__SZTYP) (Given)
Data type.

MODE = CHARACTER * (DAT__SZMOD) (Given)
Access mode ('READ', 'UPDATE' or 'WRITE').

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

EL = INTEGER (Returned)
Number of elements.

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

CMP_MOD
Ensure component exists

Description:

Ensure that a component with the specified name, type and shape exists. If a component with the required name does not exist, one is created. If a component with that name does exist but has a different type or shape, it is deleted and a new one created with the required attributes.

Invocation:

```
CALL CMP_MOD( LOC, NAME, TYPE, NDIM, DIM, STATUS )
```

Arguments:

LOC = CHARACTER * (DAT__SZLOC) (Given)
Structure locator.

NAME = CHARACTER * (DAT__SZNAM) (Given)
Component name.

TYPE = CHARACTER * (DAT__SZTYP) (Given)
Data type.

NDIM = INTEGER (Given)
Number of dimensions.

DIM = INTEGER(NDIM) (Given)
Object dimensions.

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

CMP_MODC
Ensure that string component exists

Description:

Ensure that a component with the specified name, type, string precision and shape exists. If a component with the required name does not exist, one is created. If a component with that name does exist but has a different type, string precision or shape, it is deleted and a new one created with the required attributes.

Invocation:

```
CALL CMP_MODC( LOC, NAME, LEN, NDIM, DIM, STATUS )
```

Arguments:

LOC = CHARACTER * (DAT__SZLOC) (Given)
Structure locator.

NAME = CHARACTER * (DAT__SZNAM) (Given)
Component name.

LEN = INTEGER (Given)
Number of characters per value.

NDIM = INTEGER (Given)
Number of dimensions.

DIM = INTEGER(NDIM) (Given)
Object dimensions.

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

CMP_PRIM
Enquire component primitive

Description:

Enquire if a structure component is a primitive object.

Invocation:

```
CALL CMP_PRIM( LOC, NAME, REPLY, STATUS )
```

Arguments:

LOC = CHARACTER * (DAT__SZLOC) (Given)
Structure locator.

NAME = CHARACTER * (DAT__SZNAM) (Given)
Component name.

REPLY = LOGICAL (Returned)
.TRUE. if primitive, otherwise .FALSE..

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

CMP_PUT0x
Write scalar component

Description:

Write a scalar primitive component.

Invocation:

```
CALL CMP_PUT0x( LOC, NAME, VALUE, STATUS )
```

Arguments:

LOC = CHARACTER * (DAT__SZLOC) (Given)
Structure locator.

NAME = CHARACTER * (DAT__SZNAM) (Given)
Component name.

VALUE = ? (Given)
Component value.

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

Notes:

There is a routine for each standard data type; to call them, replace x by C (Character), D (Double), I (Integer), L (Logical), R (Real).

CMP_PUT1x
Write vector component

Description:

Write a vector primitive component.

Invocation:

```
CALL CMP_PUT1x( LOC, NAME, EL, VALUE, STATUS )
```

Arguments:

LOC = CHARACTER * (DAT__SZLOC) (Given)
Structure locator.

NAME = CHARACTER * (DAT__SZNAM) (Given)
Component name.

EL = INTEGER (Given)
Number of elements.

VALUE = ?(EL) (Given)
Component value.

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

Notes:

There is a routine for each standard data type; to call them, replace x by C (Character), D (Double), I (Integer), L (Logical), R (Real).

CMP_PUTNx
Write array component

Description:

Write an array primitive component.

Invocation:

```
CALL CMP_PUTNx( LOC, NAME, NDIM, DIMX, VALUE, DIM, STATUS )
```

Arguments:

LOC = CHARACTER * (DAT__SZLOC) (Given)
Structure locator.

NAME = CHARACTER * (DAT__SZNAM) (Given)
Component name.

NDIM = INTEGER (Given)
Number of dimensions.

DIMX = INTEGER(NDIM) (Given)
Dimensions of value.

VALUE = ?(<dimx(1)>, <dimx(2)>, ...) (Given)
Component value.

DIM = INTEGER(NDIM) (Given)
Component dimensions.

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

Notes:

There is a routine for each standard data type; to call them, replace x by C (Character), D (Double), I (Integer), L (Logical), R (Real).

CMP_PUTVx
Write vectorised component

Description:

Write a primitive component as if it were vectorised.

Invocation:

```
CALL CMP_PUTVx( LOC, NAME, EL, VALUE, STATUS )
```

Arguments:

LOC = CHARACTER * (DAT__SZLOC) (Given)
Structure locator.

NAME = CHARACTER * (DAT__SZNAM) (Given)
Component name.

EL = INTEGER (Given)
Number of elements.

VALUE = ?(EL) (Given)
Component value.

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

Notes:

There is a routine for each standard data type; to call them, replace x by C (Character), D (Double), I (Integer), L (Logical), R (Real).

CMP_SHAPE
Enquire component shape

Description:
Enquire the shape of a component.

Invocation:
```
CALL CMP_SHAPE( LOC, NAME, NDIMX, DIM, NDIM, STATUS )
```

Arguments:

LOC = CHARACTER * (DAT__SZLOC) (Given)
Structure locator.

NAME = CHARACTER * (DAT__SZNAM) (Given)
Component name.

NDIMX = INTEGER (Given)
Size of DIM array.

DIM = INTEGER(NDIMX) (Returned)
Component dimensions.

NDIM = INTEGER (Returned)
Number of dimensions.

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

CMP_SIZE
Enquire component size

Description:

Enquire the size of a component. For an array this will be the product of the dimensions; for a scalar, a value of 1 is returned.

Invocation:

```
CALL CMP_SIZE( LOC, NAME, SIZE, STATUS )
```

Arguments:

LOC = CHARACTER ∗ (DAT__SZLOC) (Given)
Structure locator.

NAME = CHARACTER ∗ (DAT__SZNAM) (Given)
Component name.

SIZE = INTEGER (Returned)
Component size.

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

CMP_STRUC
Enquire component structure

Description:

Enquire if a structure component is a structure.

Invocation:

```
CALL CMP_STRUC( LOC, NAME, REPLY, STATUS )
```

Arguments:

LOC = CHARACTER * (DAT__SZLOC) (Given)
Structure locator.

NAME = CHARACTER * (DAT__SZNAM) (Given)
Component name.

REPLY = LOGICAL (Returned)
.TRUE. if a structure, otherwise .FALSE..

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

CMP_TYPE
Enquire component type

Description:

Enquire the type of a structure component.

Invocation:

```
CALL CMP_TYPE( LOC, NAME, TYPE, STATUS )
```

Arguments:

LOC = CHARACTER * (DAT__SZLOC) (Given)
Structure locator.

NAME = CHARACTER * (DAT__SZNAM) (Given)
Component name.

TYPE = CHARACTER * (DAT__SZTYP) (Returned)
Data type.

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

CMP_UNMAP
Unmap component

Description:

Unmap a structure component mapped by CMP_MAPN or CMP_MAPV.

Invocation:

```
CALL CMP_UNMAP( LOC, NAME, STATUS )
```

Arguments:

LOC = CHARACTER * (DAT__SZLOC) (Given)
Structure locator.

NAME = CHARACTER * (DAT__SZNAM) (Given)
Component name.

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

DAT_ALTER
Alter object size

Description:

Alter the size of an array by increasing or reducing the last (or only) dimension. If a structure array is to be reduced in size, the operation will fail if any truncated elements contain components.

Invocation:

```
CALL DAT_ALTER( LOC, NDIM, DIM, STATUS )
```

Arguments:

LOC = CHARACTER * (DAT__SZLOC) (Given)
Object locator.

NDIM = INTEGER (Given)
Number of dimensions.

DIM = INTEGER(NDIM) (Given)
Object dimensions.

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

DAT_ANNUL
Annul locator

Description:

Cancel the association between a locator and an object. Any primitive value currently mapped to the locator is automatically unmapped. If the last primary locator associated with a container file is annulled, then the container file will be closed.

Invocation:

```
CALL DAT_ANNUL( LOC, STATUS )
```

Arguments:

LOC = CHARACTER * (DAT__SZLOC) (Given and Returned)
Object locator. A value of DAT__NOLOC 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. In particular, it will fail if the locator supplied is not valid, but this will only be reported if status is set to SAI__OK on entry.

DAT_BASIC
Map primitive as basic units

Description:

Map a primitive as a sequence of basic machine units (bytes) for reading, writing or updating.

Invocation:

```
CALL DAT_BASIC( LOC, MODE, PNTR, LEN, STATUS )
```

Arguments:

LOC = CHARACTER * (DAT__SZLOC) (Given)
Primitive locator.

MODE = CHARACTER * (DAT__SZMOD) (Given)
Access mode ('READ', 'UPDATE' or 'WRITE').

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

LEN = INTEGER (Returned)
Total number of bytes mapped.

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

Notes:

If the primitive data have been written on a different machine which uses a different data representation, then they will be mapped as written (i.e. they will not be converted to or from the data representation used by the host machine).

DAT_CCOPY
Copy one structure level

Description:

Copy an object into a structure and give the new component the specified name. If the source object is a structure, a new structure of the same type and shape is created but the content of the original structure is not copied.

Invocation:

```
CALL DAT_CCOPY( LOC1, LOC2, NAME, LOC3, STATUS )
```

Arguments:

LOC1 = CHARACTER * (DAT__SZLOC) (Given)
Object locator.

LOC2 = CHARACTER * (DAT__SZLOC) (Given)
Structure locator.

NAME = CHARACTER * (DAT__SZNAM) (Given)
Component name.

LOC3 = CHARACTER * (DAT__SZLOC) (Returned)
Component locator.

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

DAT_CCTYP
Create type string

Description:

Create a type string for a specified size of character string.

Invocation:

```
CALL DAT_CCTYP( SIZE, TYPE )
```

Arguments:

SIZE = INTEGER (Given)
Character string size.

TYPE = CHARACTER * (DAT__SZTYP) (Returned)
Type string.

DAT_CELL
Locate cell

Description:

Return a locator to a "cell" (element) of an array object.

Invocation:

```
CALL DAT_CELL( LOC1, NDIM, SUB, LOC2, STATUS )
```

Arguments:

LOC1 = CHARACTER * (DAT__SZLOC) (Given)
Object locator.

NDIM = INTEGER (Given)
Number of dimensions.

SUB = INTEGER(NDIM) (Given)
Subscript values.

LOC2 = CHARACTER * (DAT__SZLOC) (Returned)
Cell locator.

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

Notes:

Typically, this is used to locate an element of a structure array for subsequent access to its components, although this does not preclude its use in accessing a single pixel in a 2-D image for example.

DAT_CLEN
Obtain character string length

Description:

The routine returns the number of characters required to represent the values of a primitive object. If the object is character-type, then its length is returned directly. Otherwise, the value returned is the number of characters required to format the object's values (as a decimal string if appropriate) without loss of information.

Invocation:

```
CALL DAT_CLEN( LOC, CLEN, STATUS )
```

Arguments:

LOC = CHARACTER * (DAT__SZLOC) (Given)
Primitive object locator.

CLEN = INTEGER (Returned)
Character string length.

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

Notes:

- The value returned by this routine is equal to the default number of characters allocated to each element whenever a primitive object is mapped using an access type of "_CHAR" (i.e. without specifying the length to be used explicitly).
- If this routine is called with STATUS set, then a value of 1 will be returned for the CLEN argument, although no further processing will occur. The same value will also be returned if the routine should fail for any reason.

DAT_CLONE
Clone locator

Description:

Clone (duplicate) a locator.

Invocation:

```
CALL DAT_CLONE( LOC1, LOC2, STATUS )
```

Arguments:

LOC1 = CHARACTER * (DAT__SZLOC) (Given)
Object locator.

LOC2 = CHARACTER * (DAT__SZLOC) (Returned)
Object locator.

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

Notes:

All information is inherited except that concerned with any mapped primitive data and the primary/secondary locator characteristic (a secondary locator is always produced – see DAT_PRMRY). A call to this routine is NOT equivalent to the Fortran statement "LOC2 = LOC1".

DAT_COERC
Coerce object shape

Description:

Temporarily coerce an object into changing its shape.

Invocation:

```
CALL DAT_COERC( LOC1, NDIM, LOC2, STATUS )
```

Arguments:

LOC1 = CHARACTER * (DAT__SZLOC) (Given)
Object locator.

NDIM = INTEGER (Given)
Number of dimensions.

LOC2 = CHARACTER * (DAT__SZLOC) (Returned)
Coerced object locator.

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

Notes:

If the number of dimensions in the object is to be increased, each additional dimension size is set to 1, e.g. if loc1 is associated with a 2-D object of shape (512,256) say, setting ndim to 3 transforms the dimensions to (512,256,1). Likewise, if the number of dimensions is to be reduced, the appropriate trailing dimension sizes are discarded; the routine will fail if any of these do not have the value 1. As with DAT_VEC, only the appearance of the object is changed – the original shape remains intact.

DAT_COPY
Copy object

Description:

Recursively copy an object into a component. This means that the complete object (including its components and its components's components, etc.) is copied, not just the top level.

Invocation:

```
CALL DAT_COPY( LOC1, LOC2, NAME, STATUS )
```

Arguments:

LOC1 = CHARACTER * (DAT__SZLOC) (Given)
Object locator.

LOC2 = CHARACTER * (DAT__SZLOC) (Given)
Structure locator.

NAME = CHARACTER * (DAT__SZNAM) (Given)
Component name.

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

DAT_DREP
Obtain primitive data representation information

Description:

The routine returns information describing how the data stored in a primitive object are actually represented. An object's data representation will match that used by the computer system on which it was created, and this forms a permanent attribute of the object. If necessary, HDS will automatically perform conversion to the representation used by the host computer when the data are accessed (except when using DAT_BASIC, which provides direct access to the data without conversion).

Invocation:

```
CALL DAT_DREP( LOC, FORMAT, ORDER, STATUS )
```

Arguments:

LOC = CHARACTER * (DAT__SZLOC) (Given)
Primitive object locator.

FORMAT = CHARACTER * (*) (Returned)
Description of the format used to encode each data element (see the "Data Format" section).

ORDER = CHARACTER * (*) (Returned)
Description of the (byte) storage order used for each data element (see the "Storage Order" section).

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

Notes:

Not all combinations of data format and storage order are supported.

Data Format :

HDS currently supports the following encodings of primitive data elements. Each description is preceded by the character string returned by DAT_DREP to describe it:

- 'BIT0': Used to encode logical values, in which the least significant bit (bit zero) holds the logical value such that 1 implies .TRUE. and 0 implies .FALSE.. All other bits are disregarded (except in "bad" data values when they are all significant).
- 'NZ': Used to encode logical values, in which all bits set to zero implies .FALSE. and any bit set to 1 (i.e. a non-zero data value) implies .TRUE..
- 'BINARY': Used for unsigned integers; this is a straight binary encoding.
- '2COMP': Used for signed integers in which a "2's complement" binary encoding of the sign information is employed.
- 'VAXF': Used for single precision floating point values; this is the VAX/VMS "F-floating" number representation.
- 'IEEE_S': Used for single precision floating point values; this is the standard IEEE single precision floating point format.
- 'VAXD': Used for double precision floating point values; this is the VAX/VMS "D-floating" number representation.
- 'IEEE_D': Used for double precision floating point values; this is the standard IEEE double precision floating point format.

- 'ASCII': Used for character strings; each character employs the standard ASCII encoding.

Storage Order :

HDS currently supports the following storage orders for the bytes of primitive data elements. Each description is preceded by the character string returned by DAT_DREP to describe it:

- 'MSB': Most significant byte stored first.
- 'LSB': Least significant byte stored first.

In the case of floating point formats, the byte in question is the most/least significant byte of the fraction.

DAT_ERASE
Erase component

Description:

Recursively delete a component. This means that all its lower level components are deleted as well.

Invocation:

```
CALL DAT_ERASE( LOC, NAME, STATUS )
```

Arguments:

LOC = CHARACTER * (DAT__SZLOC) (Given)
Structure locator.

NAME = CHARACTER * (DAT__SZNAM) (Given)
Component name.

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

DAT_ERMSG
Translate a status value into an error message

Description:

This routine translates an error status value into an associated error message. It first attempts to translate the value supplied as a DAT__ error code. If this fails, it then attempts to translate it as a system status code for the host operating system. If this also fails, then the returned string is a message indicating that the status value could not be translated.

Invocation:

```
CALL DAT_ERMSG( STATUS, LENGTH, MSG )
```

Arguments:

STATUS = INTEGER (Given)
The error status value to be translated.

LENGTH = INTEGER (Returned)
Number of significant characters in the returned error message (i.e. excluding trailing blanks). This value will not exceed the length of the character variable supplied for the MSG argument.

MSG = CHARACTER * (*) (Returned)
The error message.

Notes:

- If the variable supplied for the MSG argument is not long enough to accommodate the error message, then the message will be truncated and the returned value of LENGTH will reflect the truncated length.
- No returned error message will contain more significant characters than the value of the EMS__SZMSG symbolic constant. This constant is defined in the include file EMS_PAR.

DAT_FIND
Find named component

Description:

Obtain a locator for a named component.

Invocation:

```
CALL DAT_FIND( LOC1, NAME, LOC2, STATUS )
```

Arguments:

LOC1 = CHARACTER * (DAT__SZLOC) (Given)
Structure locator.

NAME = CHARACTER * (DAT__SZNAM) (Given)
Component name.

LOC2 = CHARACTER * (DAT__SZLOC) (Returned)
Component locator.

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

Notes:

If the structure is an array, loc1 must be explicitly associated with an individual cell. If the component is a structure array, loc2 will be associated with the complete array, not the first cell. Access to its components can only be made through another locator explicitly associated with an individual cell (see DAT_CELL).

DAT_GET
Read primitive

Description:

Read a primitive (access type specified by a parameter).

Invocation:

```
CALL DAT_GET( LOC, TYPE, NDIM, DIM, VALUE, STATUS )
```

Arguments:

LOC = CHARACTER * (DAT__SZLOC) (Given)
Primitive locator.

TYPE = CHARACTER * (DAT__SZTYP) (Given)
Data type.

NDIM = INTEGER (Given)
Number of dimensions.

DIM = INTEGER(NDIM) (Given)
Object dimensions.

VALUE = ?(<dim(1)>, <dim(2)>, ...) (Returned)
Object value.

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

Notes:

When reading character strings the normal Fortran 77 rules for character assignment are followed, ie. if the string length of the primitive object is less than that of the value array, each string is padded to the right with blanks; if greater, the strings are truncated from the right.

DAT_GETx
Read primitive

Description:

Read a primitive (access type specified by routine name).

Invocation:

```
CALL DAT_GETx( LOC, NDIM, DIM, VALUE, STATUS )
```

Arguments:

LOC = CHARACTER * (DAT__SZLOC) (Given)
Primitive locator.

NDIM = INTEGER (Given)
Number of dimensions.

DIM = INTEGER(NDIM) (Given)
Object dimensions.

VALUE = ?(<dim(1)>, <dim(2)>, ...) (Returned)
Object value.

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

Notes:

There is a routine for each standard data type; to call them, replace x by C (Character), D (Double), I (Integer), L (Logical), R (Real). See DAT_GET notes on character conversion.

DAT_GET0x
Read scalar primitive

Description:

Read a scalar primitive.

Invocation:

```
CALL DAT_GET0x( LOC, VALUE, STATUS )
```

Arguments:

LOC = CHARACTER * (DAT__SZLOC) (Given)
Primitive locator.

VALUE = ? (Returned)
Object value.

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

Notes:

There is a routine for each standard data type; to call them, replace x by C (Character), D (Double), I (Integer), L (Logical), R (Real). See DAT_GET notes on character conversion.

DAT_GET1x
Read vector primitive

Description:

Read a vector primitive.

Invocation:

```
CALL DAT_GET1x( LOC, ELX, VALUE, EL, STATUS )
```

Arguments:

LOC = CHARACTER * (DAT__SZLOC) (Given)
Primitive locator.

ELX = INTEGER (Given)
Maximum size of value.

VALUE = ?(ELX) (Returned)
Object value.

EL = INTEGER (Returned)
Number of elements.

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

Notes:

There is a routine for each standard data type; to call them, replace x by C (Character), D (Double), I (Integer), L (Logical), R (Real). See DAT_GET notes on character conversion.

DAT_GETNx
Read array primitive

Description:

Read an array primitive.

Invocation:

```
CALL DAT_GETNx( LOC, NDIM, DIMX, VALUE, DIM, STATUS )
```

Arguments:

LOC = CHARACTER * (DAT__SZLOC) (Given)
Primitive locator.

NDIM = INTEGER (Given)
Number of dimensions.

DIMX = INTEGER(NDIM) (Given)
Dimensions of value.

VALUE = ?(<dimx(1)>, <dimx(2)>, ...) (Returned)
Object value.

DIM = INTEGER(NDIM) (Returned)
Object dimensions.

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

Notes:

There is a routine for each standard data type; to call them, replace x by C (Character), D (Double), I (Integer), L (Logical), R (Real). See DAT_GET notes on character conversion.

DAT_GETVx
Read vectorised primitive

Description:

Read a primitive as if it were vectorised (ie. regardless of its actual shape).

Invocation:

```
CALL DAT_GETVx( LOC, ELX, VALUE, EL, STATUS )
```

Arguments:

LOC = CHARACTER * (DAT__SZLOC) (Given)
Primitive locator.

ELX = INTEGER (Given)
Maximum size of value.

VALUE = ?(ELX) (Returned)
Object value.

EL = INTEGER (Returned)
Number of elements.

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

Notes:

There is a routine for each standard data type; to call them, replace x by C (Character), D (Double), I (Integer), L (Logical), R (Real). See DAT_GET notes on character conversion.

DAT_INDEX
Index into component list

Description:

Index into a structure's component list and return a locator to the object at the given position.

Invocation:

```
CALL DAT_INDEX( LOC1, INDEX, LOC2, STATUS )
```

Arguments:

LOC1 = CHARACTER * (DAT__SZLOC) (Given)
Structure locator.

INDEX = INTEGER (Given)
List position.

LOC2 = CHARACTER * (DAT__SZLOC) (Returned)
Component locator.

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

DAT_LEN
Enquire primitive length

Description:

Enquire the length of a primitive. In the case of a character object, this is the number of characters per element. For other primitive types it is the number of bytes per element.

Invocation:

```
CALL DAT_LEN( LOC, LEN, STATUS )
```

Arguments:

LOC = CHARACTER * (DAT__SZLOC) (Given)
Primitive locator.

LEN = INTEGER (Returned)
Number of bytes per element.

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

DAT_MAP
Map primitive

Description:

Map a primitive (access type specified by a parameter).

Invocation:

```
CALL DAT_MAP( LOC, TYPE, MODE, NDIM, DIM, PNTR, STATUS )
```

Arguments:

LOC = CHARACTER * (DAT__SZLOC) (Given)
Primitive locator.

TYPE = CHARACTER * (DAT__SZTYP) (Given)
Data type.

MODE = CHARACTER * (DAT__SZMOD) (Given)
Access mode ('READ', 'UPDATE' or 'WRITE').

NDIM = INTEGER (Given)
Number of dimensions.

DIM = INTEGER(NDIM) (Given)
Object dimensions.

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

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

DAT_MAPx
Map primitive

Description:

Map a primitive (access type specified by routine name).

Invocation:

```
CALL DAT_MAPx( LOC, MODE, NDIM, DIM, PNTR, STATUS )
```

Arguments:

LOC = CHARACTER * (DAT__SZLOC) (Given)
Primitive locator.

MODE = CHARACTER * (DAT__SZMOD) (Given)
Access mode ('READ', 'UPDATE' or 'WRITE').

NDIM = INTEGER (Given)
Number of dimensions.

DIM = INTEGER(NDIM) (Given)
Object dimensions.

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

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

Notes:

There is a routine for each standard data type; to call them, replace x by C (Character), D (Double), I (Integer), L (Logical), R (Real).

DAT_MAPN
Map array primitive

Description:
Map a primitive as an array.

Invocation:

```
CALL DAT_MAPN( LOC, TYPE, MODE, NDIM, PNTR, DIM, STATUS )
```

Arguments:

LOC = CHARACTER * (DAT__SZLOC) (Given)
Primitive locator.

TYPE = CHARACTER * (DAT__SZTYP) (Given)
Data type.

MODE = CHARACTER * (DAT__SZMOD) (Given)
Access mode ('READ' 'UPDATE' or 'WRITE').

NDIM = INTEGER (Given)
Number of dimensions.

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

DIM = INTEGER(NDIM) (Returned)
Object dimensions.

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

DAT_MAPV
Map vectorised primitive

Description:

Map a primitive as if it were vectorised.

Invocation:

```
CALL DAT_MAPV( LOC, TYPE, MODE, PNTR, EL, STATUS )
```

Arguments:

LOC = CHARACTER * (DAT__SZLOC) (Given)
Primitive locator.

TYPE = CHARACTER * (DAT__SZTYP) (Given)
Data type.

MODE = CHARACTER * (DAT__SZMOD) (Given)
Access mode ('READ', 'UPDATE' or 'WRITE').

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

EL = INTEGER (Returned)
Number of elements.

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

DAT_MOULD
Alter object shape

Description:

Alter an object's shape permanently.

Invocation:

```
CALL DAT_MOULD( LOC, NDIM, DIM, STATUS )
```

Arguments:

LOC = CHARACTER * (DAT__SZLOC) (Given)
Object locator.

NDIM = INTEGER (Given)
Number of dimensions.

DIM = INTEGER(NDIM) (Given)
Object dimensions.

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

Notes:

The shape is altered permanently (unlike DAT_COERC). The number of dimensions cannot be increased.

DAT_MOVE
Move object

Description:

Move an object to a new location (ie. copy and erase the original).

Invocation:

```
CALL DAT_MOVE( LOC1, LOC2, NAME, STATUS )
```

Arguments:

LOC1 = CHARACTER * (DAT__SZLOC) (Given and Returned)
Object locator. A value of DAT__NOLOC is returned.

LOC2 = CHARACTER * (DAT__SZLOC) (Given)
Structure locator.

NAME = CHARACTER * (DAT__SZNAM) (Given)
Component name.

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

Notes:

If the object is an array, loc1 must point to the complete array, not a slice or cell. loc1 is annulled if the operation is successful (if it is the last primary locator associated with a container file, then the container file will be closed – see DAT_PRMRY). The operation will fail if a component of the same name already exists in the structure. The object to be moved need not be in the same container file as the structure.

DAT_MSG
Assign the name of an HDS object to a message token

Description:

The routine assigns the full name (including the file name) of an HDS object to a message token for use with the ERR_ and MSG_ routines (SUN/104) or with the EMS_ routines (SSN/4). Appropriate syntax is used to represent file names which do not have the standard '.sdf' file extension.

Invocation:

```
CALL DAT_MSG( TOKEN, LOC )
```

Arguments:

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

LOC = CHARACTER * (DAT__SZLOC) (Given)
Locator to the HDS object.

Notes:

- This routine has no STATUS argument and does not perform normal error checking. If it should fail, then no value will be assigned to the message token and this will be apparent in the final message.

DAT_NAME
Enquire object name

Description:

Enquire the name of an object.

Invocation:

```
CALL DAT_NAME( LOC, NAME, STATUS )
```

Arguments:

LOC = CHARACTER * (DAT__SZLOC) (Given)
Object locator.

NAME = CHARACTER * (DAT__SZNAM) (Returned)
Object name.

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

DAT_NCOMP
Enquire number of components

Description:

Return the number of components in a structure.

Invocation:

```
CALL DAT_NCOMP( LOC, NCOMP, STATUS )
```

Arguments:

LOC = CHARACTER * (DAT__SZLOC) (Given)
Structure locator.

NCOMP = INTEGER (Returned)
Number of components.

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

Notes:

If the structure is an array, loc must be explicitly associated with an individual cell.

DAT_NEW
Create component

Description:

Create a new component in a structure.

Invocation:

```
CALL DAT_NEW( LOC, NAME, TYPE, NDIM, DIM, STATUS )
```

Arguments:

LOC = CHARACTER * (DAT__SZLOC) (Given)
Structure locator.

NAME = CHARACTER * (DAT__SZNAM) (Given)
Component name.

TYPE = CHARACTER * (DAT__SZTYP) (Given)
Data type.

NDIM = INTEGER (Given)
Number of dimensions.

DIM = INTEGER(NDIM) (Given)
Component dimensions.

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

Notes:

If type matches one of the primitive type names a primitive of the appropriate type is created, otherwise the object is assumed to be a structure. The new object can subsequently be located by DAT_FIND. The operation will fail if a component of the same name already exists.

DAT_NEW0x
Create scalar component

Description:

Create a new scalar primitive component in a structure.

Invocation:

```
CALL DAT_NEW0x( LOC, NAME, STATUS )
```

Arguments:

LOC = CHARACTER * (DAT__SZLOC) (Given)
Structure locator.

NAME = CHARACTER * (DAT__SZNAM) (Given)
Component name.

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

Notes:

There is a routine for each standard data type; to call them, replace x by D (Double), I (Integer), L (Logical), R (Real).

DAT_NEW0C
Create scalar string component

Description:

Create a new scalar string component in a structure.

Invocation:

```
CALL DAT_NEW0C( LOC, NAME, LEN, STATUS )
```

Arguments:

LOC = CHARACTER * (DAT__SZLOC) (Given)
Structure locator.

NAME = CHARACTER * (DAT__SZNAM) (Given)
Component name.

LEN = INTEGER (Given)
Number of characters per value.

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

DAT_NEW1x
Create vector component

Description:

Create a new vector primitive component in a structure.

Invocation:

```
CALL DAT_NEW1x( LOC, NAME, EL, STATUS )
```

Arguments:

LOC = CHARACTER * (DAT__SZLOC) (Given)
Structure locator.

NAME = CHARACTER * (DAT__SZNAM) (Given)
Component name.

EL = INTEGER (Given)
Number of elements.

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

Notes:

There is a routine for each standard data type; to call them, replace x by D (Double), I (Integer), L (Logical), R (Real).

DAT_NEW1C
Create vector string component

Description:

Create a new vector string component in a structure.

Invocation:

```
CALL DAT_NEW1C( LOC, NAME, LEN, EL, STATUS )
```

Arguments:

LOC = CHARACTER * (DAT__SZLOC) (Given)
Structure locator.

NAME = CHARACTER * (DAT__SZNAM) (Given)
Component name.

LEN = INTEGER (Given)
Number of characters per value.

EL = INTEGER (Given)
Number of elements.

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

DAT_NEWC
Create string component

Description:

Create a new component whose elements are of type "_CHAR∗n".

Invocation:

```
CALL DAT_NEWC( LOC, NAME, LEN, NDIM, DIM, STATUS )
```

Arguments:

LOC = CHARACTER ∗ (DAT__SZLOC) (Given)
Structure locator.

NAME = CHARACTER ∗ (DAT__SZNAM) (Given)
Component name.

LEN = INTEGER (Given)
Number of characters per value.

NDIM = INTEGER (Given)
Number of dimensions.

DIM = INTEGER(NDIM) (Given)
Object dimensions.

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

DAT_PAREN
Locate parent structure

Description:

The routine returns a locator for the parent structure of an HDS object; i.e. the structure which contains the object.

Invocation:

```
CALL DAT_PAREN( LOC1, LOC2, STATUS )
```

Arguments:

LOC1 = CHARACTER * (DAT__SZLOC) (Given)
Object locator.

LOC2 = CHARACTER * (DAT__SZLOC) (Returned)
Parent structure locator.

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

Notes:

- On successful exit, the parent structure locator will identify a scalar structure (number of dimensions zero). If appropriate, this may be a cell of a structure array.
- An error will result, and the STATUS value DAT__OBJIN will be returned if the object supplied does not have a parent; i.e. if it is the top-level object in a container file. The DAT__OBJIN error code is defined in the include file DAT_ERR.
- If this routine is called with STATUS set, then a value of DAT__NOLOC will be returned for the LOC2 argument, although no further processing will occur. The same value will also be returned if the routine should fail for any reason. The DAT__NOLOC constant is defined in the include file DAT_PAR.

Example :

The parent of the object A.B.C.D is A.B.C, the parent of X.DATA.ARRAY(1:256) is X.DATA, and the parent of Z.STRUC(17).FLAG is Z.STRUC(17).

DAT_PREC
Enquire storage precision

Description:

Enquire the number of basic machine units (bytes) used to store a single element of a primitive.

Invocation:

```
CALL DAT_PREC( LOC, NBYTE, STATUS )
```

Arguments:

LOC = CHARACTER * (DAT__SZLOC) (Given)
Primitive locator.

NBYTE = INTEGER (Returned)
Number of machine units.

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

DAT_PRIM
Enquire object primitive

Description:

Enquire if an object is a primitive.

Invocation:

```
CALL DAT_PRIM( LOC, REPLY, STATUS )
```

Arguments:

LOC = CHARACTER * (DAT__SZLOC) (Given)

Object locator.

REPLY = LOGICAL (Returned)

.TRUE. if primitive, otherwise .FALSE..

STATUS = INTEGER (Given and Returned)

The global status.

DAT_PRMRY
Set or enquire primary/secondary locator status

Description:

The routine may be used to promote a locator to become a "primary" locator, to demote it to become a "secondary" locator, or to enquire about the primary/secondary status of a locator. It allows control over the duration for which an HDS container file remains open; each file remains open only so long as there is at least one primary locator associated with it.

Invocation:

```
CALL DAT_PRMRY( SET, LOC, PRMRY, STATUS )
```

Arguments:

SET = LOGICAL (Given)

If a .TRUE. value is given for this argument, then the routine will perform a "set" operation to set the primary/secondary status of a locator. Otherwise it will perform an "enquire" operation to return the value of this status without changing it.

LOC = CHARACTER * (DAT__SZLOC) (Given and Returned)

The locator whose primary/secondary status is to be set or enquired.

PRMRY = LOGICAL (Given and Returned)

If SET is .TRUE., then this is an input argument and specifies the new value to be set (.TRUE. for a primary locator, .FALSE. for a secondary locator). If SET is .FALSE., then this is an output argument and will return a value indicating whether or not a primary locator was supplied.

STATUS = INTEGER (Given and Returned)

The global status.

Notes:

- The value of the LOC argument will not normally be changed. However, if it is the last primary locator associated with a container file, and is being demoted from a primary to a secondary locator, then the container file will be left without an associated primary locator. In this case, the locator supplied will be annulled (along with any other secondary locators associated with the same file), a value of DAT__NOLOC will be returned, and the file will be closed.
- The DAT__NOLOC constant is defined in the include file DAT_PAR.

DAT_PUT
Write primitive

Description:

Write a primitive (type specified by a parameter).

Invocation:

```
CALL DAT_PUT( LOC, TYPE, NDIM, DIM, VALUE, STATUS )
```

Arguments:

LOC = CHARACTER * (DAT__SZLOC) (Given)
Primitive locator.

TYPE = CHARACTER * (DAT__SZTYP) (Given)
Data type.

NDIM = INTEGER (Given)
Number of dimensions.

DIM = INTEGER(NDIM) (Given)
Object dimensions.

VALUE = ?(<dim(1)>, <dim(2)>, ...) (Given)
Object value.

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

Notes:

When writing character strings the normal Fortran 77 rules for character assignment are followed, ie. if the string length of the value array is less than that of the primitive, each string is padded to the right with blanks; if greater, they are truncated from the right.

DAT_PUTx
Write primitive

Description:

Write a primitive (type specified by routine name).

Invocation:

```
CALL DAT_PUTx( LOC, NDIM, DIM, VALUE, STATUS )
```

Arguments:

LOC = CHARACTER * (DAT__SZLOC) (Given)
Primitive locator.

NDIM = INTEGER (Given)
Number of dimensions.

DIM = INTEGER(NDIM) (Given)
Object dimensions.

VALUE = ?(<dim(1)>, <dim(2)>, ...) (Given)
Object value.

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

Notes:

There is a routine for each standard data type; to call them, replace x by C (Character), D (Double), I (Integer), L (Logical), R (Real). See DAT_PUT notes on character conversion.

DAT_PUT0x
Write scalar primitive

Description:

Write a scalar primitive.

Invocation:

```
CALL DAT_PUT0x( LOC, VALUE, STATUS )
```

Arguments:

LOC = CHARACTER * (DAT__SZLOC) (Given)
Primitive locator.

VALUE = ? (Given)
Object value.

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

Notes:

There is a routine for each standard data type; to call them, replace x by C (Character), D (Double), I (Integer), L (Logical), R (Real). See DAT_PUT notes on character conversion.

DAT_PUT1x
Write vector primitive

Description:

Write a vector primitive.

Invocation:

```
CALL DAT_PUT1x( LOC, EL, VALUE, STATUS )
```

Arguments:

LOC = CHARACTER * (DAT__SZLOC) (Given)
Primitive locator.

EL = INTEGER (Given)
Number of elements.

VALUE = ?(EL) (Given)
Object value.

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

Notes:

There is a routine for each standard data type; to call them, replace x by C (Character), D (Double), I (Integer), L (Logical), R (Real). See DAT_PUT notes on character conversion.

DAT_PUTNx
Write array primitive

Description:

Write an array primitive.

Invocation:

```
CALL DAT_PUTNx( LOC, NDIM, DIMX, VALUE, DIM, STATUS )
```

Arguments:

LOC = CHARACTER * (DAT__SZLOC) (Given)
Primitive locator.

NDIM = INTEGER (Given)
Number of dimensions.

DIMX = INTEGER(NDIM) (Given)
Dimensions of value.

VALUE = ?(<dimx(1)>, <dimx(2)>, ...) (Given)
Object value.

DIM = INTEGER(NDIM) (Given)
Object dimensions.

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

Notes:

There is a routine for each standard data type; to call them, replace x by C (Character), D (Double), I (Integer), L (Logical), R (Real). See DAT_PUT notes on character conversion.

DAT_PUTVx
Write vectorised primitive

Description:

Write a primitive as if it were vectorised (ie. regardless of its actual shape).

Invocation:

```
CALL DAT_PUTVx( LOC, EL, VALUE, STATUS )
```

Arguments:

LOC = CHARACTER * (DAT__SZLOC) (Given)
Primitive locator.

EL = INTEGER (Given)
Number of elements.

VALUE = ?(EL) (Given)
Object value.

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

Notes:

There is a routine for each standard data type; to call them, replace x by C (Character), D (Double), I (Integer), L (Logical), R (Real). See DAT_PUT notes on character conversion.

DAT_REF
Obtain a reference for an HDS object

Description:

The routine returns a "reference name" for an HDS object whose locator is supplied. This name identifies the object uniquely by including both the name of the container file and the "path name" which locates the object within this file. If a locator to a cell or a slice is supplied, then subscript information will also be included. Appropriate syntax is used to represent file names which do not have the standard '.sdf' file extension.

Invocation:

```
CALL DAT_REF( LOC, REF, LREF, STATUS )
```

Arguments:

LOC = CHARACTER * (DAT__SZLOC) (Given)
Locator to the HDS object.

REF = CHARACTER * (*) (Returned)
The object's reference name.

LREF = INTEGER (Returned)
Number of significant characters in the reference name.

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

DAT_REFCT
Enquire the reference count for a container file

Description:

The routine returns a count of the number of "primary" locators associated with an HDS container file (its reference count). The file will remain open for as long as this number is greater than zero.

Invocation:

```
CALL DAT_REFCT( LOC, REFCT, STATUS )
```

Arguments:

LOC = CHARACTER * (DAT__SZLOC) (Given)
Locator associated with any object in the container file.

REFCT = INTEGER (Returned)
Number of primary locators currently associated with the file.

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

Notes:

This routine may be used to determine whether annulling a primary locator will cause a container file to be closed (also see the routine DAT_PRMRY).

DAT_RENAM
Rename object

Description:

Rename an object.

Invocation:

```
CALL DAT_RENAM( LOC, NAME, STATUS )
```

Arguments:

LOC = CHARACTER * (DAT__SZLOC) (Given)

Object locator.

NAME = CHARACTER * (DAT__SZNAM) (Given)

New object name.

STATUS = INTEGER (Given and Returned)

The global status.

DAT_RESET
Reset object state

Description:

Reset the state of a primitive, ie. "un-define" its value. All subsequent read operations will fail until the object is written to (re-defined).

Invocation:

```
CALL DAT_RESET( LOC, STATUS )
```

Arguments:

LOC = CHARACTER * (DAT__SZLOC) (Given)

Primitive locator.

STATUS = INTEGER (Given and Returned)

The global status.

DAT_RETYP
Change object type

Description:

Change the type of an object. A blank type may be specified.

Invocation:

```
CALL DAT_RETYP( LOC, TYPE, STATUS )
```

Arguments:

LOC = CHARACTER * (DAT__SZLOC) (Given)
Object locator.

TYPE = CHARACTER * (DAT__SZTYP) (Given)
Data type.

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

DAT_SHAPE
Enquire object shape

Description:
Enquire the shape of an object.

Invocation:
```
CALL DAT_SHAPE( LOC, NDIMX, DIM, NDIM, STATUS )
```

Arguments:

LOC = CHARACTER * (DAT__SZLOC) (Given)
Object locator.

NDIMX = INTEGER (Given)
Size of dim.

DIM = INTEGER(NDIMX) (Returned)
Object dimensions.

NDIM = INTEGER (Returned)
Number of dimensions.

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

DAT_SIZE
Enquire object size

Description:

Enquire the size of an object. For an array this will be the product of the dimensions; for a scalar, a value of 1 is returned.

Invocation:

```
CALL DAT_SIZE( LOC, SIZE, STATUS )
```

Arguments:

LOC = CHARACTER * (DAT__SZLOC) (Given)
Object locator.

SIZE = INTEGER (Returned)
Object size.

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

DAT_SLICE
Locate slice

Description:

Return a locator to a "slice" of a vector or an array.

Invocation:

```
CALL DAT_SLICE( LOC1, NDIM, DIML, DIMU, LOC2, STATUS )
```

Arguments:

LOC1 = CHARACTER * (DAT__SZLOC) (Given)
Array locator.

NDIM = INTEGER (Given)
Number of dimensions.

DIML = INTEGER(NDIM) (Given)
Lower dimension bounds.

DIMU = INTEGER(NDIM) (Given)
Upper dimension bounds.

LOC2 = CHARACTER * (DAT__SZLOC) (Returned)
Slice locator.

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

Notes:

If any of the upper bounds are zero or negative, the size of the corresponding dimension in the object is used. Currently, only 1, 2 or 3-D objects can be sliced.

DAT_STATE
Enquire object state

Description:

Enquire the state of a primitive, ie. whether its value is defined or not.

Invocation:

```
CALL DAT_STATE( LOC, REPLY, STATUS )
```

Arguments:

LOC = CHARACTER * (DAT__SZLOC) (Given)
Primitive locator.

REPLY = LOGICAL (Returned)
.TRUE. if defined, otherwise .FALSE..

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

DAT_STRUC
Enquire object structure

Description:

Enquire if an object is a structure.

Invocation:

```
CALL DAT_STRUC( LOC, REPLY, STATUS )
```

Arguments:

LOC = CHARACTER * (DAT__SZLOC) (Given)

Object locator.

REPLY = LOGICAL (Returned)

.TRUE. if structure, otherwise .FALSE..

STATUS = INTEGER (Given and Returned)

The global status.

DAT_TEMP
Create temporary object

Description:

Create an object that exists only for the lifetime of the program run. This may be used to hold temporary objects – including those mapped to obtain scratch space.

Invocation:

```
CALL DAT_TEMP( TYPE, NDIM, DIM, LOC, STATUS )
```

Arguments:

TYPE = CHARACTER * (DAT__SZTYP) (Given)
Data type.

NDIM = INTEGER (Given)
Number of dimensions.

DIM = INTEGER(NDIM) (Given)
Object dimensions.

LOC = CHARACTER * (DAT__SZLOC) (Returned)
Object locator.

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

Notes:

If type matches one of the primitive type names, a primitive of appropriate type is created; otherwise the object is assumed to be a structure. If the object is a structure array, loc will be associated with the complete array, not the first cell. Thus, new components can only be created through another locator which is explicitly associated with an individual cell (see DAT_CELL).

DAT_THERE
Enquire component existence

Description:
Enquire if a component of a structure exists.

Invocation:

```
CALL DAT_THERE( LOC, NAME, REPLY, STATUS )
```

Arguments:

LOC = CHARACTER * (DAT__SZLOC) (Given)
Structure locator.

NAME = CHARACTER * (DAT__SZNAM) (Given)
Component name.

REPLY = LOGICAL (Returned)
.TRUE. if exists, otherwise .FALSE..

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

DAT_TYPE
Enquire object type

Description:

Enquire the type of an object.

Invocation:

```
CALL DAT_TYPE( LOC, TYPE, STATUS )
```

Arguments:

LOC = CHARACTER * (DAT__SZLOC) (Given)
Object locator.

TYPE = CHARACTER * (DAT__SZTYP) (Returned)
Data type.

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

DAT_UNMAP
Unmap object

Description:

Unmap an object mapped by another DAT_ routine.

Invocation:

```
CALL DAT_UNMAP( LOC, STATUS )
```

Arguments:

LOC = CHARACTER * (DAT__SZLOC) (Given)

Primitive locator.

STATUS = INTEGER (Given and Returned)

The global status.

DAT_VALID
Enquire locator valid

Description:

Enquire if a locator is valid, ie. currently associated with an object.

Invocation:

```
CALL DAT_VALID( LOC, REPLY, STATUS )
```

Arguments:

LOC = CHARACTER * (DAT__SZLOC) (Given)
Locator.

REPLY = LOGICAL (Returned)
.TRUE. if valid, otherwise .FALSE..

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

DAT_VEC
Vectorise object

Description:

Address an array as if it were a vector.

Invocation:

```
CALL DAT_VEC( LOC1, LOC2, STATUS )
```

Arguments:

LOC1 = CHARACTER * (DAT__SZLOC) (Given)
Array locator.

LOC2 = CHARACTER * (DAT__SZLOC) (Returned)
Vector locator.

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

Notes:

In general, it is not possible to vectorise an array slice such as created by DAT_SLICE. If LOC1 is a locator for a slice, an error will be reported if the elements of the slice are dis-contiguous. If the elements of the slice are contiguous, then no error will be reported.

DAT_WHERE
Find position of primitive in HDS file

Description:

The routine returns information describing the position in an HDS container file at which the data associated with a primitive object are stored.

Invocation:

```
CALL DAT_WHERE( LOC, BLOCK, OFFSET, STATUS )
```

Arguments:

LOC = CHARACTER * (DAT__SZLOC) (Given)

Primitive object locator.

BLOCK = INTEGER (Returned)

Number of the file block in which the object's data start. HDS file blocks are 512 bytes long and are numbered from the beginning of the file, starting at block 1.

OFFSET = INTEGER (Returned)

Byte offset (zero based) of the start of data within the file block.

STATUS = INTEGER (Given and Returned)

The global status.

Notes:

- The use of this routine is not recommended. It is provided solely for use by experienced programmers who are familiar with the internal structure of HDS container files and who wish to access the file contents directly. Note, however, that changes to the format of an HDS file may occur in future.
- The start of the data associated with a cell or a slice of a primitive object may be located with this routine, but the data associated with a slice will not, in general, be stored at contiguous locations within the file.
- Care must be taken that no changes are made to adjacent bytes within the file which are not part of the requested object's data.
- Note that the data associated with primitive objects may not necessarily be stored contiguously in future versions of HDS.

HDS_COPY
Copy an object to a new container file

Description:

The routine makes a copy of an HDS object, placing the copy in a new container file (which is created), as the top-level object. The copying operation is recursive; i.e. all sub-components of a structure will also be copied.

Invocation:

```
CALL HDS_COPY( LOC, FILE, NAME, STATUS )
```

Arguments:

LOC = CHARACTER * (DAT__SZLOC) (Given)
Locator for the object to be copied.

FILE = CHARACTER * (*) (Given)
Name of the new container file to be created.

NAME = CHARACTER * (*) (Given)
Name which the new top-level object is to have.

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

Notes:

- This routine attempts to eliminate unused space during the copying operation and is therefore a useful method of compressing a container file from which components have been deleted.
- The routine may be used to copy both primitive and structured objects, but cannot be used to make a copy of a cell or a slice.

HDS_ERASE
Erase container file

Description:

Mark a container file for deletion and annul the locator associated with the top-level object. The container file will not be physically deleted if other primary locators are still associated with the file – this is only done when the reference count drops to zero.

Invocation:

```
CALL HDS_ERASE( LOC, STATUS )
```

Arguments:

LOC = CHARACTER * (DAT__SZLOC) (Given)
Locator to the container file's top-level object.

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

HDS_EWILD
End a wild-card search for HDS container files

Description:

The routine ends a wild-card search for HDS container files begun by HDS_WILD, and annuls the wild-card search context used. It should be called after a wild-card search is complete in order to release the resources used.

Invocation:

```
CALL HDS_EWILD( IWLD, STATUS )
```

Arguments:

IWLD = INTEGER (Given and Returned)

Identifier for the wild-card search context to be annulled, as returned by HDS_WILD. A value of DAT__NOWLD is returned (as defined in the include file DAT_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.

HDS_FLUSH
Flush locator group

Description:

Annul all locators currently assigned to a specified locator group.

Invocation:

```
CALL HDS_FLUSH( GROUP, STATUS )
```

Arguments:

GROUP = CHARACTER * (DAT__SZGRP) (Given)

Group name.

STATUS = INTEGER (Given and Returned)

The global status.

HDS_FREE
Free container file

Description:

Release all explicit or implicit locks on a container file, thereby granting write-access to other processes.

Invocation:

```
CALL HDS_FREE( LOC, STATUS )
```

Arguments:

LOC = CHARACTER * (DAT__SZLOC) (Given)
Locator to the container file's top-level object.

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

HDS_GROUP
Enquire locator group

Description:

Return the name of the group to which a locator belongs.

Invocation:

```
CALL HDS_GROUP( LOC, GROUP, STATUS )
```

Arguments:

LOC = CHARACTER * (DAT__SZLOC) (Given)
Object locator.

GROUP = CHARACTER * (DAT__SZGRP) (Returned)
Group name.

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

HDS_GTUNE
Obtain tuning parameter value

Description:

The routine returns the current value of an HDS tuning parameter (normally this will be its default value, or the value last specified using the HDS_TUNE routine).

Invocation:

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

Arguments:

PARAM = CHARACTER * (*) (Given)

Name of the tuning parameter whose value is required (case insensitive).

VALUE = INTEGER (Returned)

Current value of the parameter.

STATUS = INTEGER (Given and Returned)

The global status.

Notes:

Tuning parameter names may be abbreviated to 4 characters.

HDS_LINK
Link locator group

Description:
Link a locator to a group.

Invocation:

```
CALL HDS_LINK( LOC, GROUP, STATUS )
```

Arguments:

LOC = CHARACTER * (DAT__SZLOC) (Given)
Object locator.

GROUP = CHARACTER * (DAT__SZGRP) (Given)
Group name.

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

HDS_LOCK
Lock container file

Description:

Lock a container file in update mode; this protects the file from being accessed by other writers. The file remains explicitly locked until HDS_FREE is called or until the file is physically closed.

Invocation:

```
CALL HDS_LOCK( LOC, STATUS )
```

Arguments:

LOC = CHARACTER * (DAT__SZLOC) (Given)
Locator to the container file's top-level object.

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

HDS_NEW
Create container file

Description:

Create a new container file and return a primary locator to the top-level object.

Invocation:

```
CALL HDS_NEW( FILE, NAME, TYPE, NDIM, DIM, LOC, STATUS )
```

Arguments:

FILE = CHARACTER * (*) (Given)
Container file name.

NAME = CHARACTER * (DAT__SZNAM) (Given)
Object name.

TYPE = CHARACTER * (DAT__SZTYP) (Given)
Data type.

NDIM = INTEGER (Given)
Number of dimensions.

DIM = INTEGER(NDIM) (Given)
Object dimensions (ignored if NDIM is zero).

LOC = CHARACTER * (DAT__SZLOC) (Returned)
Object locator.

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

Notes:

If a file extension is not specified, '.sdf' is used. If type matches one of the HDS primitive type names a primitive of that type is created, otherwise the object is assumed to be a structure. If the top-level object is a structure array, loc will be associated with the complete array, not the first cell. Thus, new components can only be created through another locator which is explicitly associated with an individual cell (see DAT_CELL).

HDS_OPEN
Open container file

Description:

Open an existing container file for reading, writing or updating and return a primary locator to the top-level object. If the file is currently open in the specified mode, the routine increments the reference count rather than re-opening the file.

Invocation:

```
CALL HDS_OPEN( FILE, MODE, LOC, STATUS )
```

Arguments:

FILE = CHARACTER * (*) (Given)
Container file name.

MODE = CHARACTER * (DAT__SZMOD) (Given)
Access mode ('READ', 'UPDATE' or 'WRITE').

LOC = CHARACTER * (DAT__SZLOC) (Returned)
Object locator.

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

Notes:

If a file extension is not specified, '.sdf' is used. If the top-level object is a structure array, loc will be associated with the complete array, not the first cell. Thus, access to any of the structure's components can only be made through another locator which is explicitly associated with an individual cell (see DAT_CELL).

HDS_SHOW
Show HDS statistics

Description:

Display statistics about the specified topic on the standard output.

Invocation:

```
CALL HDS_SHOW( TOPIC, STATUS )
```

Arguments:

TOPIC = CHARACTER * (*) (Given)

Topic name.

STATUS = INTEGER (Given and Returned)

The global status.

Notes:

This facility is provided as an implementation aid only. The following topics are currently supported:

- 'DATA': display details of the representation of primitive data in use by the host machine.
- 'FILES': list all open container files and their associated attributes in the format:

  <filename>, disp=<disp>, mode=<mode>, refcnt=<refcnt>

  where <disp> denotes the file disposition ([K] for keep or [D] for delete); <mode> indicates the access mode ([R] for read-only, [U] for update or write) and <refcnt> specifies the number of primary locators associated with the file.

- 'LOCATORS': display the fully resolved path names of all objects currently associated with active locators.

HDS_STATE
Enquire the current state of HDS

Description:

This routine returns a logical value indicating whether HDS is currently active or inactive.

Invocation:

```
CALL HDS_STATE( STATE, STATUS )
```

Arguments:

STATE = LOGICAL (Returned)

The current state of HDS: .TRUE. for active, .FALSE. for inactive.

STATUS = INTEGER (Given and Returned)

The global status.

HDS_STOP
Close down HDS

Description:

This routine closes down HDS, annulling all active locators, closing all container files and releasing all associated resources. It returns without action if HDS is not active.

Invocation:

```
CALL HDS_STOP( STATUS )
```

Arguments:

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.

HDS_TRACE
Trace object path

Description:

Trace the path of an object and return the fully resolved name as a text string.

Invocation:

```
CALL HDS_TRACE( LOC, NLEV, PATH, FILE, STATUS )
```

Arguments:

LOC = CHARACTER * (DAT__SZLOC) (Given)
Object locator.

NLEV = INTEGER (Returned)
Number of path levels.

PATH = CHARACTER * (*) (Returned)
Object path name within container file.

FILE = CHARACTER * (*) (Returned)
Container file name.

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

Notes:

The path name is constructed as a sequence of "node" specifications thus: node.node.....node.object such as: NGC1365.SKYPOS.RA.MINUTES where MINUTES is the name of the object associated with the specified locator and NGC1365 is the top-level object in the structure. If any of the nodes are non-scalar the appropriate subscript expression is included thus: AAO.OBS(6).IMAGE_DATA If the bottom-level object is a slice or cell of an array, the appropriate subscript expression is appended thus: M87.MAP(100:412,200:312) or CENA(3,2)

HDS_TUNE
Set HDS tuning parameter

Description:
Alter an HDS control setting.

Invocation:

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

Arguments:

PARAM = CHARACTER * (*) (Given)
Name of the tuning parameter.

VALUE = INTEGER (Given)
New parameter value.

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

HDS_WILD
Perform a wild-card search for HDS container files

Description:

The routine searches for HDS container files whose names match a given wild-card file specification, and which are accessible using a specified mode of access. It is normally called repeatedly, returning a locator for the top-level object in a new container file on each occasion, and a null locator value (DAT__NOLOC) when no more container files remain to be located.

In normal use, the IWLD argument should be set to the value DAT__NOWLD before the first call to HDS_WILD. The value returned through this argument subsequently identifies the search context, which is retained between calls. In this way, several wild-card searches may be performed concurrently if required.

A call to HDS_EWILD should be made to annul the search context identifier when the search is complete. This will release any resources used.

Invocation:

```
CALL HDS_WILD( FSPEC, MODE, IWLD, LOC, STATUS )
```

Arguments:

FSPEC = CHARACTER * (*) (Given)

The wild-card file specification identifying the container files required (a default file type extension of '.sdf' is assumed, if not specified). The syntax of this specification depends on the host operating system (see §F.2).

MODE = CHARACTER * (DAT__SZMOD) (Given)

The mode of access required to the container files: 'READ', 'UPDATE' or 'WRITE' (case insensitive).

IWLD = INTEGER (Given and Returned)

If a value of DAT__NOWLD is supplied on input, then a new wild-card search context will be started, a new value for IWLD will be returned, and the first HDS container file matching the file specification given in FSPEC will be located. If an IWLD value saved from a previous invocation of HDS_WILD is supplied, then the previous search context will be used and the next container file appropriate to that context will be located. In this case, the value of FSPEC is not used.

LOC = CHARACTER * (DAT__SZLOC) (Returned)

A primary locator to the top-level object in the next container file to satisfy the file specification given in FSPEC. A value of DAT__NOLOC will be returned (without error) if no further container files remain to be located.

STATUS = INTEGER (Given and Returned)

The global status.

Notes:

- This routine does not return locators for files which are not valid HDS container files or which are not accessible using the specified access mode.
- The routine takes a "snapshot" of the file system when it is first called (with IWLD set to DAT__NOWLD) and subsequently returns locators for each of the HDS container files found, one at a time. This strategy avoids possible run-away conditions if (say) output files created by an application were later to be selected by a wild-card search used to identify further input files.

- An error may result if any file matched by an initial call to HDS_WILD (with IWLD set to DAT__NOWLD) cannot be accessed when a subsequent call requires that a locator be returned for it. This might happen, for instance, if the file has been deleted in the intervening time. If the resulting error condition is annulled, the offending file may be skipped and subsequent calls to HDS_WILD will continue to locate any remaining files.
- An error will result and a STATUS value of DAT__FILNF will be returned if no HDS container files can be found which match the wild-card specification supplied on an initial call to HDS_WILD.
- A value of DAT__NOLOC will be returned for the LOC argument if this routine is called with STATUS set, or if it should fail for any reason. In addition, the value of IWLD will be returned unchanged if the routine is called with STATUS set or if failure should occur during an initial call (i.e. when IWLD is set to DAT__NOWLD on entry).
- The DAT__NOLOC and DAT__NOWLD constants are defined in the include file DAT_PAR. The DAT__FILNF error code is defined in the include file DAT_ERR.

C OBSOLETE ROUTINES

The routines described below have been rendered obsolete by developments in HDS and should not be used in new software. They are included here simply as an aid to understanding existing software which uses them, and to allow them to be replaced with alternative techniques as the opportunity arises. These routines will eventually be removed from the HDS documentation and may, in some cases, eventually be eliminated from HDS altogether.

The reason for obsolescence is indicated in each case.

DAT_CONV
Enquire data conversion

Description:

Compare the data type of a primitive with the type of data being used by the program and determine whether conversion is possible.

Invocation:

```
CALL DAT_CONV( LOC, TYPE, REPLY, STATUS )
```

Arguments:

LOC = CHARACTER * (DAT__SZLOC) (Given)
Primitive locator.

TYPE = CHARACTER * (DAT__SZTYP) (Given)
Data type used by the program.

REPLY = LOGICAL (Returned)
.TRUE. if conversion possible, otherwise .FALSE..

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

Notes:

This routine is obsolete as HDS now permits all conversions between primitive types.

DAT_ERDSC
Report object error

Description:

Report an error in the form "object_name: message" where message is the text associated with error number status.

Invocation:

```
CALL DAT_ERDSC( LOC, STATUS )
```

Arguments:

LOC = CHARACTER * (DAT__SZLOC) (Given)
Object locator.

STATUS = INTEGER (Given)
The global status.

Notes:

This routine is obsolete as HDS now reports its own errors. Any additional error reports from applications should be made using the ERR_ routines (SUN/104). Error reports from system code should use the EMS_ routines (SSN/4).

DAT_ERDSN
Report component error

Description:

Report an error in the form "structure_name.component_name: message" where message is the text associated with error number status.

Invocation:

```
CALL DAT_ERDSN( LOC, NAME, STATUS )
```

Arguments:

LOC = CHARACTER * (DAT__SZLOC) (Given)
Structure locator.

NAME = CHARACTER * (DAT__SZNAM) (Given)
Component name.

STATUS = INTEGER (Given)
The global status.

Notes:

This routine is obsolete as HDS now reports its own errors. Any additional error reports from applications should be made using the ERR_ routines (SUN/104). Error reports from system code should use the EMS_ routines (SSN/4).

DAT_ERTXT
Report error

Description:

Report an error in the form "text: message" where text is a specified character string and message is the text associated with error number status.

Invocation:

```
CALL DAT_ERTXT( TEXT, STATUS )
```

Arguments:

TEXT = CHARACTER * (*) (Given)
Character string.

STATUS = INTEGER (Given)
The global status.

Notes:

This routine is obsolete as HDS now reports its own errors. Any additional error reports from applications should be made using the ERR_ routines (SUN/104). Error reports from system code should use the EMS_ routines (SSN/4).

DAT_RCERA
Recursive erase

Description:

Recursively delete a component of a structure.

Invocation:

```
CALL DAT_RCERA( LOC, NAME, STATUS )
```

Arguments:

LOC = CHARACTER * (DAT__SZLOC) (Given)
Object locator.

NAME = CHARACTER * (DAT__SZNAM) (Given)
Component name.

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

Notes:

This routine is obsolete; its action is now performed by DAT_ERASE.

DAT_RCOPY
Recursive copy

Description:

Recursively copy an object into a component of a structure.

Invocation:

```
CALL DAT_RCOPY( LOC1, LOC2, NAME, STATUS )
```

Arguments:

LOC1 = CHARACTER * (DAT__SZLOC) (Given)
Object locator.

LOC2 = CHARACTER * (DAT__SZLOC) (Given)
Structure locator.

NAME = CHARACTER * (DAT__SZNAM) (Given)
Component name.

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

Notes:

This routine is obsolete; its action is now performed by DAT_COPY.

DAT_TUNE
Set HDS parameter

Description:
Specify a value for an HDS control parameter.

Invocation:
```
CALL DAT_TUNE( PARAM, VALUE, STATUS )
```

Arguments:

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

VALUE = INTEGER (Given)
Parameter value.

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

Notes:
This routine is obsolete – HDS_TUNE should be used instead.

HDS_CLOSE
Close container file

Description:

Annul the locator associated with the top-level object in a container file, decrement the container file reference count by one and close the file if the reference count reaches zero. If the file is closed, all other locators associated with it (both primary and secondary) will also be annulled.

Invocation:

```
CALL HDS_CLOSE( LOC, STATUS )
```

Arguments:

LOC = CHARACTER * (DAT__SZLOC) (Given)
Locator to the container file's top-level object.

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

Notes:

This routine is obsolete – its behaviour is flawed because it does not recognise that some top-level locators are not associated with a container file reference count (those created with DAT_CLONE, for instance). HDS_CLOSE will always decrement the reference count and, hence, may close a container file while it is still in use via other locators. The dependence on top-level locators to hold a container file open has been superceded by the more general primary/secondary locator attribute (see DAT_PRMRY) and the role previously played by HDS_CLOSE is now incorporated in DAT_ANNUL.

HDS_RUN
Run an HDS application subroutine

Description:

This routine starts up HDS, runs an application subroutine which uses HDS, and then closes HDS down again.

Invocation:

```
CALL HDS_RUN( APP, STATUS )
```

Arguments:

APP = SUBROUTINE (Given)
The subroutine to be executed.

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

Notes:

This routine is obsolete as HDS is now self-starting. The application subroutine can therefore be called directly.

HDS_START
Start up HDS

Description:

This routine should be called to start up HDS prior to making calls to other HDS routines. It ensures that HDS is active, returning without action if it is already active.

Invocation:

```
CALL HDS_START( STATUS )
```

Arguments:

STATUS = INTEGER (Given and Returned)

The global status.

Notes:

This routine is now obsolete, as HDS starts itself up automatically when required.

D CALLING HDS FROM C

The C interface for the HDS library is defined within file `$STARLINK_DIR/include/star/hds.h`. There is a one-to-one correspondance between each C function and each Fortran routine described in this document - the Fortran documentation should be consulted for information about the purpose of each function and the nature of its arguments.

Within C, each HDS locator is referred to by a pointer to an " `HDSLoc`", and all such variables *must* be initialised to NULL when they are declared. Functions that require a locator as input should be supplied with an "`HDSLoc *`". Functions that return a locator as output should be supplied with an "`HDSLoc **`" (i.e. the address of an "`HDSLoc *`"):

```
#include "star/hds.h"
...
HDSLoc *loc = NULL;
HDSLoc *cloc = NULL;
int status;
...
hdsOpen( "fred.sdf", "READ", &loc, &status );
datFind( loc, "DATA", &cloc, &status );
...
datAnnul( &cloc, status );
datAnnul( &loc, status );
```

Each C function returns the integer status value as the function value in addition to returning it in the usual manner via the inherited status variable. In most cases, the returned function value will be ignored, as in the above example.

The C API includes the following three extra functions to manage the locking of HDS Locators by specific threads:

datLock
Lock an object for exclusive use by the current thread

Description:

This function locks an HDS object for use by the current thread. An object can be locked for read-only access or read-write access. Multiple threads can lock an object simultaneously for read-only access, but only one thread can lock an object for read-write access at any one time. Use of any HDS function that may modify the object will fail with an error unless the thread has locked the object for read-write access. Use of an HDS function that cannot modify the object will fail with an error unless the thread has locked the object (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 the object), this function will report an error if any other thread currently has a lock (read-only or read-write) on the object.

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

If the object is a structure, each component object will have its own lock, which is independent of the lock on the parent object. A component object and its parent can be locked by different threads. However, as a convenience function this function allows all component objects to be locked in addition to the supplied object (see " recurs").

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

Invocation:

```
datLock( HDSLoc *locator, int recurs, int readonly, int *status );
```

Arguments:

locator = HDSLoc * (Given)
Locator to the object that is to be locked.

recurs = int (Given)
If the supplied object is locked successfully, and " recurs" is non-zero, then an attempt is made to lock any component objects contained within the supplied object. An error is reported if any components cannot be locked due to them being locked already by a different thread. This operation is recursive - any children of the child components are also locked, etc.

readonly = int (Given)
If non-zero, the object (and child objects if " recurs" is non-zero) is locked for read-only access. Otherwise it is locked for read-write access.

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

Notes:

- An error will be reported if the supplied object is currently locked by another thread. If " recurs" is non-zero, an error is also reported if any component objects contained within the supplied object are locked by other threads.

- The majority of HDS functions will report an error if the object supplied to the function has not been locked for use by the calling thread. The exceptions are the functions that manage these locks - datLock, datUnlock and datLocked.
- Attempting to lock an object 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.

datLocked
See if an object is locked

Description:

This function returns a value that indicates if the object specified by the supplied locator has been locked for use by one or more threads. A thread can lock an object either for read-only access or for read-write access. The lock management functions (datLock and datUnlock) will ensure that any thread that requests and is given a read-write lock will have exclusive access to the object - no other locks of either type will be issued to other threads until the first thread releases the read-write lock using datUnlock. 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 = datLocked( const HDSLoc *locator, int recursive, int *status );
```

Arguments:

locator = const HDSLoc * (Given)
A locator for the object to be checked.

recursive = int (Given)
If non-zero, then all descendants of the supplied object are also checked in the same way.

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

Notes:

- The locking performed by datLock, datUnlock and datLocked is based on POSIX threads, and has no connection with the locking referred to in hdsLock and hdsFree.
- Zero is returned as the function value if an error has already occurred, or if an error occurs in this function.

Returned function value :

A value indicating the status of the supplied Object:

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

0: the supplied object is unlocked. If " recursive" is non-zero, then all descendant objects are also unlocked, and this is then the condition that must be met for the current thread to be able to lock the supplied object for read-write access using function datLock. This condition can be achieved by releasing any existing locks using datUnlock.

1: the supplied object 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 object in any HDS function that might modify the object (except for the locking and unlocking functions - see below). If " recursive" is non-zero, then all descendant objects are also locked for reading and writing. This condition can be achieved by calling datLock.

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

is non-zero, then either the object itself or one of its descendant objects is locked for reading and writing.

3: the supplied object is locked read-only by the current thread (and maybe other threads as well). If " recursive" is non-zero, then all descendant objects are also locked read-only by the current thread. This is the condition that must be met for the current thread to be able to use the supplied object in any HDS function that cannot modify the object. An error will be reported if the current thread attempts to use the object in any HDS function that could modify the object. This condition can be achieved by calling datLock.

4: the supplied object 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 object in any other HDS function. If " recursive" is non-zero, then all descendant objects are also locked read-only by one or more other threads.

5: Some complex mix of locked and unlocked descendants not covered by any of the above values.

datUnlock
Unlock an object so that it can be locked by a different thread

Description:

This function removes a lock on the supplied HDS object. An error is reported if the object is not locked by the current thread. If " recurs" is non-zero, an error will be reported if any child component within the supplied object is not locked by the current thread. See datLock.

The object must be locked again, using datLock, before it can be used by any other HDS function. All objects are initially locked by the current thread when they are created.

Invocation:

```
datUnlock( HDSLoc *locator, int recurs, int *status );
```

Arguments:

locator = HDSLoc * (Given)
Locator to the object that is to be unlocked.

recurs = int (Given)
If the supplied object is unlocked successfully, and " recurs" is non-zero, then an attempt is made to unlock any component objects contained within the supplied object.

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

Notes:

- The majority of HDS functions will report an error if the object supplied to the function has not been locked for use by the calling thread. The exceptions are the functions datLock and datLocked.

E ERROR CODES

E.1 Error Code Descriptions

An error status value (not equal to SAI__OK) is returned by any HDS routine which detects an error condition. If a program is to perform specific tests on these status values, then the HDS-supplied symbolic names described below should be used rather than explicit numerical values. These symbolic status names are prefixed either with 'DAT' (for error codes returned by the DAT_ and HDS_ routines) or 'CMP' (for error codes returned by the CMP_ routines) and a double underscore. The appropriate symbol definitions are contained in the include files DAT_ERR (for the DAT__ error codes) and CMP_ERR (for the CMP__ error codes). The symbols defined in these two files are shown in the following tables and described in more detail below.

Symbolic Name	*Meaning*
DAT__ACCON	Access conflict
DAT__BOUND	Outside bounds of object
DAT__COMEX	Component already exists
DAT__CONER	Conversion error
DAT__DELIN	Deletion invalid
DAT__DIMIN	Dimensions invalid
DAT__FATAL	Fatal internal error
DAT__FILCK	File locking error
DAT__FILCL	File close error
DAT__FILCR	File create error
DAT__FILIN	File invalid
DAT__FILMP	File mapping error
DAT__FILND	File not deleted
DAT__FILNF	File not found
DAT__FILNX	File not extended
DAT__FILPR	File protected
DAT__FILRD	File read error
DAT__FILWR	File write error
DAT__GRPIN	Group invalid
DAT__INCHK	Integrity check
DAT__ISMAP	Data currently mapped
DAT__LOCIN	Locator invalid
DAT__MODIN	Mode invalid
DAT__NAMIN	Name invalid
DAT__NOMEM	Insufficient memory available
DAT__OBJIN	Object invalid
DAT__OBJNF	Object not found
DAT__PRMAP	Primitive data mapped
DAT__SUBIN	Subscripts invalid
DAT__TRUNC	Text truncated
DAT__TYPIN	Type invalid
DAT__UNSET	Primitive data undefined
DAT__VERMM	Version mismatch
DAT__WLDIN	Wild-card search context invalid

Symbolic Name	*Meaning*
CMP__DIMIN	Dimensions invalid
CMP__FATAL	Fatal internal error
CMP__ISMAP	Data currently mapped
CMP__NOMAP	Not mapped
CMP__TYPIN	Type invalid

ACCON: A write operation has been rejected because the container file was originally opened for read-only access.

BOUND: An array specification is outside the permitted dimension bounds.

COMEX: An attempt to create a new component of a structure has failed because a component of the same name already exists.

CONER: One of the following primitive data conversion errors occurred during a GET, PUT, MAP or UNMAP operation:

- Floating point overflow.
- Integer overflow.
- Truncation of non-blank characters in a character string.

DELIN: The object to be deleted was a non-empty structure.

DIMIN: One of the following conditions occurred:

- When creating an object, the number of dimensions specified exceeds the system-imposed limit (currently 7) or is negative, or one or more of the dimension sizes is zero or negative.
- When slicing an object, the number of dimensions specified exceeds the system-imposed limit (currently 3) or is negative, or an incorrect number of lower and upper bounds has been specified.
- An attempt to read, write, or map primitive data has failed because the dimensions specified do not match the actual shape of the object.

FATAL: An unrecoverable error has been detected by HDS. This may indicate an internal programming error within HDS itself, but may also result from the use of a corrupt container file. The associated error report will contain further information about the nature of the error.

FILCK: An attempt to lock a container file has failed. This is most likely to be because another process currently has the file locked with a conflicting access mode.

FILCL: An error occurred while closing an HDS file.

FILCR: An error occurred while creating an HDS file.

FILMP: An error occurred while mapping data in an HDS file.

FILND: An error occurred while deleting an HDS file.

FILNX: An error occurred while extending the size of an HDS file.

FILIN: One of the following conditions occurred:

- A file to be opened does not conform to the structure of an HDS container file.
- The name given for a file to be created matches the name of a file which is already in use by HDS.
- The name given for a file to be created matches the name of an existing file which cannot be removed because it is not a regular file (it may be a directory, for instance).

FILNF: The specified file cannot be found, or the syntax of the file name has been rejected as invalid by the host operating system.

FILPR: The file is protected from being accessed in the specified mode (or from being deleted).

FILRD: An error occurred while reading data from an HDS file.

FILWR: An error occurred while writing data to an HDS file.

GRPIN: The supplied character string does not conform to the syntax of a group specification.

INCHK: Typically, an attempt to access an object has failed because the object has been deleted or moved through another locator, either in the same program or in another process. This error may also be produced through the use of a corrupt container file.

ISMAP: An object is currently mapped.

LOCIN: One of the following conditions occurred:

- A non-character variable has been specified as a locator (or a character variable of the wrong length).
- The specified locator is not currently valid, *i.e.* not associated with an object.
- The specified locator was generated by an HDS routine which did not run to successful completion.

MODIN: The supplied character string is not a valid access mode.

NAMIN: The supplied character string does not conform to the syntax of a name specification.

NOMEM: HDS was unable to dynamically allocate sufficient memory to perform the requested operation.

OBJIN: An input locator points to an object which is not suitable for the type of operation requested. Specifically, when the locator is associated with one of the following:

- A structure object where a primitive is required (and vice versa).
- An n-D object where a scalar (or cell) is required.
- A slice or cell of an n-D object where the complete array is required.
- A top-level object where a component object is required (and vice versa).

OBJNF: A request to locate a component of a structure, by name or position, has failed because the object does not exist.

PRMAP: A GET, PUT or MAP operation has been rejected because primitive data are currently mapped to the specified locator.

SUBIN: An attempt to locate a slice or cell of an n-D object has failed because the subscript information references an element which is outside the array bounds of the object (or a lower bound exceeds an upper bound).

TRUNC: A character string has been corrupted through the truncation of significant (non-blank) characters. Typically this indicates that a program has not supplied a character string of sufficient length as an argument to an HDS routine.

TYPIN: The supplied character string does not conform to the syntax of a type specification.

UNSET: An attempt to read primitive object data has failed because the values are currently undefined.

VERMM: The HDS file format version number of a container file exceeds the version number recognised by the HDS software. Typically, this signifies that the file has been created or modified by a new version of HDS and cannot be handled by a program linked with an older version.

WLDIN: The wild-card search context identifier supplied has not been generated by a previous successful call to an HDS routine and is not equal to the null value (DAT__NOWLD).

E.2 Obsolete Error Codes

The following error codes are no longer used by HDS, although their values remain defined:

DAT__ACTIV

DAT__CONIN

DAT__ERACT

DAT__EREXH

DAT__EXCPA

DAT__ISOPN

DAT__RELIN

DAT__STKOF

DAT__UNKPA

DAT__WEIRD

F MACHINE-DEPENDENT FEATURES

Although the implementations of HDS on different computer systems have been made as similar as possible, there are necessarily some differences in behaviour (especially compared to earlier VAX/VMS implementations) due to the underlying operating system. This appendix describes the behaviour for the operating systems on which HDS is currently supported.

F.1 File Naming

This section describes the naming rules for files when opening individual HDS container files. The rules used when searching for container files using the "wild-carding" facilities provided by HDS_WILD differ slightly (see §F.2).

File names may contain characters of either case on UNIX systems and HDS therefore does not perform any case conversion on file names. All leading and trailing white space on file names is ignored.

If a file name does not have an extension (*i.e.* does not have a period in the last field of its UNIX pathname), then HDS provides a default extension by appending '.sdf' to the name. This occurs both when searching for existing files and when creating new ones. To make it possible to create file names without a period if required, HDS will also remove exactly one period (if present) from the end of a file name before using it.

If a file name contains any "special" characters[10] and the value of the SHELL tuning parameter (§15.2) indicates that a shell is to be used for interpreting such file names, then the name will be passed to a process running the required shell for expansion before use. Thus any shell expression resulting in a valid file name may be used, such as:

```
$MY_DIR/datafile
/reduce/data
‘cat myfilelist‘
```

(the precise syntax and capabilities depending on which shell is selected). The actual algorithm used for expanding file names is described in §F.2. If expansion results in more than one file name, then only the first one is used.

There are no file version numbers on UNIX systems. If an existing file name is given as the name of a new output file, then the original file will be over-written. It is an error to specify a file which has already been opened by HDS (*e.g.* for input) as the name of a new output file.

F.2 Wild-Card File Searching

When searching for HDS container files using the "wild-card" facilities provided by HDS_WILD and HDS_EWILD, the following rules apply.

Since traditional UNIX systems do not generally provide callable pattern-matching facilities for finding files, nor do they support the concept of default file type extensions, HDS implements these by passing wild-card file names to a shell process and expanding them using the following algorithm:

[10] That is, anything except alphanumeric characters, slash '/', period '.', underscore '_' and minus '−'.

(1) The string supplied (which may also be a list of files or file specifications separated by white space) is first expanded with the shell's file name expansion facility turned off. This performs operations such as environment variable substitution and execution of embedded shell commands, but leaves pattern-matching characters in place.

(2) The resulting string is then scanned as a list of file specifications separated by white space.

(3) If any element of this list does not contain a file type extension, as indicated by the absence of a period in the final field (fields being separated by slash characters '/'), then an extension of '.sdf' is appended to it. If an element initially ends in a period, then this is removed and no extension is added. This stage performs file type defaulting on all elements of the list consistent with the rules applied to individual file names (§F.1).

(4) The resulting list of file specifications is then expanded with the shell's file name expansion facility turned on, resulting in the final list of files.

HDS will further filter the list of files found to exclude any which appear not to exist, cannot be accessed, or are not valid HDS container files.

F.3 File Mapping

The `sun4_Solaris` and `alpha_OSF1` implementations of HDS currently provide a choice between the use of file mapping to perform I/O operations on primitive objects and the use of direct read/write operations. This is selectable via the 'MAP' tuning parameter (see §15.2). The default is to use file mapping on both systems.

Only read/write operations will be supported on other systems, regardless of the setting of the 'MAP' tuning parameter.

F.4 Scratch Files

The environment variable HDS_SCRATCH is used on UNIX systems to define the directory in which HDS will create scratch container files to hold temporary objects created with DAT_TEMP. When creating a scratch file, HDS will use the file name:

$HDS_SCRATCH/<filename>

where $HDS_SCRATCH is the translation of this environment variable. If HDS_SCRATCH is undefined, then the file name will be used alone, so that scratch files will then be created in the current directory.

F.5 File Locking

HDS file locking is not currently implemented on UNIX systems. This means that any number of processes may access a container file simultaneously. This should not be allowed to happen, however (apart from shared read access), because modifications to the file will not be propagated consistently between the file and the separate processes. The result will be a corrupted container file and/or hung or crashed processes.

Note that HDS supports access to container files over the Network File System (NFS), but does not provide file locking.

G CHANGES AND NEW FEATURES

G.1 Changes in V4.1

The following describes the most significant changes which occurred in HDS between versions V4.0 and V4.1 (not the current version).

(1) The C source code of HDS now complies with the ANSI C standard, and UNIX implementations of HDS use a new "makefile". Both of these changes are designed to make it relatively straightforward to implement HDS on new platforms.

(2) HDS now contains routines which automatically determine the primitive data representation used by common types of computer hardware (*e.g.* the floating point number format and byte storage order). This also makes it easier to implement HDS on new platforms, since changes to the source code are not normally required.

(3) A DECstation implementation of HDS has been added. An implementation on Silicon Graphics hardware is also in use, although this is not currently supported by Starlink.

(4) The SUN Sparcstation implementation of HDS now uses file mapping as its default mode of file access. This can give appreciable performance gains, largely, it appears, as a result of reduced memory usage which may allow better buffering of file contents.

(5) The directory in which HDS resides now contains a file called `hds_datestamp` which holds information about how the HDS system was built. This includes details of the primitive data representation used by the host machine which may sometimes be of use to programmers.

(6) A bug has been fixed which could cause the length of new _CHAR objects to exceed the length actually requested. Typically, an extra digit was being appended to the length as a result of an un-terminated internal C string.

(7) A bug has been fixed which caused HDS not to recognise primitive data values written in big-endian IEEE double precision format (*i.e.* typically written on a SUN) which have the "bad" data value due to a previous format conversion error. The result of this bug was that a data conversion error could result if the data were read on a machine (typically a VAX) with a more restricted double precision number range. The correct behaviour is for the bad SUN value to be converted silently to the corresponding bad VAX value. This now occurs.

(8) A bug has been fixed which could corrupt a mapped slice of a primitive object because only a single byte of data was being transferred for the final element. This could happen both when reading and writing the data, although not in all circumstances.

(9) A problem has been fixed which typically resulted in "bus errors" on SUN systems when attempting to access primitive data as double precision values where format conversion was also required. This resulted from an inability to handle double precision values passed from Fortran to C, where the value may be badly aligned in memory (*i.e.* on a 4-byte rather than an 8-byte boundary). This problem was not always repeatable, in the sense that it depended on where the Fortran compiler placed the relevant variable in memory.

(10) A workaround has been installed for a problem sometimes encountered when writing to a file served by a VMS machine using the Network File System (NFS) from a SUN. Depending on the file size and the amount of data written, a failure to extend the file can sometimes occur. This seems to happen only when the new file size requested by the SUN lies between the current end of file and the physical file size allocated by VMS as a result of clustering of disk blocks. This problem seems to have arisen since the previous version of HDS due to changes to the VMS UCX software. The workaround involves repeating the file extend call if it fails on the first attempt.

(11) A bug has been fixed in the queue handling facility used internally by HDS. This could cause regions of memory to be over-written. It is not clear what adverse consequences this may have had.

(12) An error in the documentation concerning the order in which character arguments should appear when passing mapped character values using the %VAL facility has been corrected (see §8).

(13) The description of the DAT_BASIC routine has been improved to make it clear that this routine accesses bytes of primitive data *as written*, and does not perform conversion to/from the data representation of the host machine.

(14) Several minor typos in error messages have been corrected.

(15) Use of the routine HDS_START is no longer necessary in order to start up HDS before using it. The system is now self-starting, typically when the first routine which accesses a locator is called. HDS_START has been documented as obsolete, but its use remains optional.

(16) Use of the routine HDS_STOP at the end of a program is now optional, since its action will be performed automatically by an exit handler. Examples showing its use have been modified appropriately. Note that HDS_STOP is not obsolete, as it remains the only method of closing down HDS in the middle of a program. In practice, the need to do this is likely to be limited.

(17) Since HDS_START and HDS_STOP are now both optional, the routine HDS_RUN no longer serves a useful purpose and has been documented as obsolete.

(18) The method by which HDS determines how long a container file should be held open has been rationalised. This no longer depends on the existence of a "top-level" locator. Instead, the concept of "primary" and "secondary" locators has been introduced (see 12.4) and a container file remains open for as long as at least one primary locator is associated with it. Any locator may be designated as primary by means of the new routine DAT_PRMRY, thus removing the special status of top-level locators in this context. Routines which previously incremented the container file "reference count" now issue primary locators (all other locators are secondary by default), so that existing behaviour is retained.

    This change has been introduced to remove the requirement that all software using HDS maintain its own table of top-level locators in order to prevent container files being closed. It also opens the way for future improvements to the programming interface of HDS, which should allow objects to be identified by their pathname as well as by locator.

(19) All routines which annul locators will now also close the associated container file if there are no longer any primary locators associated with it. Any outstanding secondary locators associated with a closed container file are now correctly annulled (previously they were simply left "dangling"). The main implication of this is that DAT_ANNUL will now close a container file which is no longer in use, removing the need to do this explicitly.

(20) As a result of the above changes, the HDS_CLOSE routine is now redundant, and has been documented as obsolete. Its continued use is not recommended. Its behaviour has always been flawed, since it decrements the reference count for a container file regardless of whether the top-level locator it annuls originally caused this count to be incremented when it was created. Since several methods exist for generating top-level locators without incrementing this count, it is possible for a container file to be prematurely closed if HDS_CLOSE is used.

HDS_CLOSE may still be useful as an emergency measure to close a file in the presence of a programming error which has left it open.

(21) A new routine DAT_REFCT has been introduced to return the current reference count for a container file. This makes it possible to predict when a file will actually be closed.

(22) The temporary limit on the number of simultaneously open container files imposed in V4.0 of HDS has been removed. HDS now imposes no restrictions on the number of open files beyond those set by the host operating system.

(23) On UNIX systems, HDS now reports an error if the name of a container file which is already in use is given as the name of a new container file to be created. This avoids the fairly common problem on UNIX, where the user of an application supplies the same name for both the input file and the output file, and ends up with no file at all because the output over-writes the input before it has been read.

(24) When creating new container files on UNIX systems, HDS will now check before over-writing an existing file to ensure that it is a regular file and not a directory or FIFO, *etc.* An error results if it is not a regular file.

(25) HDS now consistently ignores all leading and trailing white space on file names passed to it.

(26) On UNIX systems, HDS now uses the full (absolute) pathnames of all files when referring to them in messages and when returning file names *via* routine arguments.

(27) On UNIX systems, HDS will now pass file names which contain "special" characters (anything except alphanumerics, '.', '/', '-' and '_') to a shell for interpretation. This means that normal UNIX shell syntax may be used to construct file names, thus allowing environment variable expansion, *etc.* In fact, any shell command can, in principle, be used to construct a file name. Pattern-matching characters are also accepted – if more than one file matches, then the first match is used.

(28) Two new routines HDS_WILD and HDS_EWILD have been introduced to permit "wild-card" searching for HDS container files specified using pattern-matching characters.

(29) A new tuning parameter 'SHELL' has been introduced (see 15.2) to allow a choice of which UNIX shell is used to interpret file names (this includes wild-carding of file names using

HDS_WILD and HDS_EWILD). By default, the "sh" shell is used but, if available, the "csh" and "tcsh" shells may also be selected. It is also possible to turn shell expansion of file names off.

(30) It is now possible to modify the default tuning profile of HDS by means of environment variables, which are read and interpreted by HDS at startup (see §15). For instance, the definition:

```
setenv HDS_MAP 0
```

could be used to disable file mapping in favour of I/O in any application which uses HDS.

(31) A call to HDS_GTUNE will now only return the value 1 for the 'MAP' tuning parameter (corresponding to file mapping being used as the file access mode) if this mode has previously been requested and if it is actually implemented on the host machine. This makes it possible for the caller to determine whether file mapping is implemented or not.

(32) Three new special values (-1, -2 and -3) may now be given for the 'MAP' tuning parameter in order to select the file access mode best suited for a given type of access. These options select (in order) the file access method which gives fastest sequential access, fastest random (sparse) access, or minimum usage of memory (see 15.2). The value returned by HDS_GTUNE may be used to determine which file access mode was actually selected, as this will depend on the host operating system.

(33) Some of the values returned by the routine DAT_CLEN describing the number of digits required to format floating point numbers have been changed to reflect the recommendations of the IEEE floating point standard.

(34) The HDS_SHOW routine now has a new 'DATA' option which displays details of the primitive data representation in use by the host machine.

(35) Internal changes have been made which allow HDS "container records" (which contain information about the components which reside within an HDS structure) to be reduced in size when components are erased. Some hysteresis is allowed in this process. This typically gives a slight performance improvement and saves a little file space.

(36) The NBLOCKS tuning parameter is now actually used at several places within HDS. Previously its value had no effect.

(37) A new script called `hds_dev` has been provided on UNIX systems to create and remove symbolic links to the HDS public include files within the current directory. This allows the same INCLUDE statements to be used in Fortran code on both UNIX and VMS systems.

(38) The new error code DAT__WLDIN and the symbolic constant DAT__NOWLD have been introduced.

(39) This document (SUN/92) has been updated and produced using a larger font.

G.2 Changes in V4.2

The following describes the most significant changes which occurred in HDS between versions V4.1 and V4.2 (not the current version).

(1) Support for the VAX/VMS operating system has been discontinued at this version of HDS and all references to VMS usage have been dropped from this document.

A VAX/VMS implementation of HDS V4.1 remains in operation and continues to be supported by Starlink on a "care and maintenance" basis, but as a separate entity. This means that the following and future additions to HDS will not be applied to the VAX/VMS version. Users of HDS on VAX/VMS should refer to the documentation that accompanies it for information, and not to this document.

(2) The OSF1 (Digital UNIX) implementation of HDS now supports file mapping. This should improve the performance of those applications that can take advantage of it.

(3) HDS now allows container files that are marked for deletion to be re-opened for use, so long as they have not yet been closed (and therefore deleted). This facility was found to be necessary to support foreign data file access in the NDF library (SSN/20).

(4) Due to the introduction of various new POSIX facilities on OSF1 (Digital UNIX) and associated compiler flags, some untested code was being compiled and used on this platform. The result was that file specifications containing "wild-carding" characters could not be used and the HDS_WILD routine did not function correctly in recent un-tested distributions of HDS for OSF1. These problems have now been fixed.

(5) A bug has been fixed which meant that the HDS_WILD routine would not function correctly if it was the first HDS routine to be called in an HDS application. A call to HDS_START would prevent this problem occurring, but is now no longer necessary.

(6) A bug has been fixed which could result in objects accessed for modification using file mapping to be written back to their container files using a `write` call (instead of unmapping the file) if the 'MAP' tuning parameter was changed between calls to DAT_MAP (or equivalent) and DAT_ANNUL (or equivalent). This problem did not occur if DAT_UNMAP was called to unmap the data explicitly. Surprisingly, this erroneous behaviour almost always gave the correct result, but could very occasionally result in a core dump.

(7) A bug has been fixed that could corrupt data in the immediate neighbourhood of a primitive object slice whose value was updated. This could only happen in rare circumstances and was normally limited to slices no more than 32 bytes in length.

(8) Certain error messages that are often the result of an attempt to access a corrupt HDS container file have been modified to explicitly give the affected file name.

(9) The HDS makefile has been extensively revised. Amongst other things, this is to permit automatic distribution *via* the *Starlink Software Store* on the World Wide Web.

(10) This document (SUN/92) has been revised and is now available in both Latex and hypertext (HTML) forms.

(11) The HDS implementations on SunOS and Ultrix systems have been superceded by Solaris and OSF1 implementations respectively (the latter now being called Digital UNIX). The code to support these (and other) earlier implementations of HDS has been retained, and may be of use to those wishing to port HDS back to these platforms. However, it is no longer supported on these systems by Starlink and has not recently been built or tested on them. An Open VMS implementation of HDS running on Dec Alpha workstations is also known to exist, but is not supported by Starlink.

Further information about old or unsupported implementations of HDS may be obtained by contacting Starlink Software Support (stardev@jiscmail.ac.uk)

G.3 Changes in V4.3

The following describes the most significant changes which occurred in HDS between versions V4.2 and V4.3 (the current version).

(1) A change has been made to the way that data pointers are allocated by HDS, so that the new CNF_PVAL function (described in SUN/209) may be used when passing pointers using the "%VAL" facility (see §7). Use of this function remains optional on currently-supported platforms, but it may be required if Fortran software which holds pointers in INTEGER variables is required to run in circumstances where memory pointers require a longer variable for storage.

    Use of CNF_PVAL is recommended in all new Fortran software which uses HDS pointers.

(2) The maximum number of objects which may be simultaneously mapped via CMP_ routines has been increased to 1024.

(3) This document (SUN/92) has been updated to reflect the above changes.

G.4 Changes in V6.0

(1) The entire HDS library has been re-implemented to support HDS files that use the HDF5 data format. The library can read and write both old (V4) and new (V5) formats. By default, new files are created using the old V4 format, but this can be changed using the new "VERSION" tuning parameter. Note, the default output format will change to V5 in a future release.

(2) New functions (datock, datUnlock and datLocked) have been added to the C interface to allow an HDS object to be locked for exclusive use by a single thread. An error is reported if a thread attempts to make a change to an object that it has not previously locked for its own exclusive use. By defaults objects are locked by the thread that creates them.