SUN/224.2

Starlink Project
Starlink User Note 224.2

A J Chipperfield

10 February 2000

HDSPAR - HDS Parameter Routines 1.1 Programmers manual

Abstract

HDSPAR is a library of subroutines which interface between the Starlink Hierarchical Data System (HDS) and the ADAM parameter system. HDS objects are handled by reference to a program parameter name rather than their object name.

Contents

1 Introduction

HDSPAR is a library of subroutines which interface between the Starlink Hierarchical Data System (HDS) and the ADAM parameter system.

The subroutines do things like opening, closing, creating and deleting HDS objects (see SUN/92) but instead of passing the object name to the subroutine, the name of a program parameter is passed. The parameter system then associates the parameter with the required object, using the normal parameter system process, involving the Interface Module (see SUN/115) and possibly prompting the user.

Apart from DAT_CANCL, all the routines obey the Starlink error-handling convention described in SUN/104 – if the STATUS argument is not SAI__OK on entry, they return without action; if they detect an error, they report an error message and return an appropriate STATUS value. DAT_CANCL will attempt to operate regardless of the given status value.

2 Associating with an Object

If you just want to perform some operations on an existing HDS object using basic HDS subroutines, you need an HDS locator (see SUN/92) for the object and usually you will want to specify the name of the object as a parameter of the program.

The DAT_ASSOC routine will associate an object with a program parameter and return a locator for it. The association can be cancelled and the locator annulled by calling DAT_CANCL.

```
INCLUDE 'DAT_PAR'

INTEGER STATUS
CHARACTER*(DAT__SZLOC) LOC

CALL DAT_ASSOC( 'OBJECT', 'READ', LOC, STATUS )

... HDS operations ...

CALL DAT_CANCL( 'OBJECT', STATUS )
```

If DAT_ASSOC fails to associate the parameter with an object it will report the problem and prompt for a new value. It will do this up to five times before giving up and returning status PAR__NULL.

DAT_EXIST is similar to DAT_ASSOC but it will return immediately if the association fails. It can therefore be used to test for the existence of an object. Status PAR__ERROR (not DAT_EXIST, for historical reasons) will be returned if the object does not exist, but the parameter value will not be cancelled so a subsequent HDSPAR subroutine call (such as DAT_CREAT) specifying the same parameter would not prompt for a new object name. See the example in Section 10 for an illustration of this.

The ACCESS Field (see SUN/115) specified in the Interface Module for the program must be compatible with the access mode specified in the DAT_ASSOC or DAT_EXIST call, otherwise status SUBPAR_ICACM will be returned.

Both routines may be called more than once for the same parameter but DAT_CANCL must be called to cancel an existing association if a different object is required (see 'Parameter States', Section 3).

3 Parameter States

A parameter has a number of states. Initially, a parameter has no value, and is said to be in the *ground* state. When the parameter has been given a value, the parameter moves to the *active* state. A parameter acquires a value in the first instance by looking for one supplied on the command line. Failing that, when the program requests a parameter value, the parameter system will attempt to get a value from one of a number of sources specified by the VPATH Field in the Interface Module (see SUN/115). This may cause the user to be prompted and the prompt may contain a *suggested value* which can be accepted by just hitting the RETURN key.

If a program requests a value for a parameter already in the active state, the existing value is returned. To obtain a new value the program must first *cancel* the parameter, moving it to the *cancelled* state. DAT_CANCL does this as well as annulling the associated locator.

When the application gets a value for a *cancelled* parameter the VPATH Field is ignored and the user prompted. When a value is obtained, the parameter returns to the *active* state.

The parameter may also go into the *null* state. This occurs if the parameter is given the null value, for example by the user entering ! in response to a prompt. When the parameter is in the *null* state, any attempt to get its value will return status PAR__NULL – it must be cancelled before another value can be obtained.

4 Creating Objects

DAT_CREAT may be used to create an object but note that it does not return a locator – a subsequent call to DAT_ASSOC or DAT_EXIST is required before the object can be used. Only one object can be created, its parents, if any, must already exist. If the parameter specifies a top-level object, any existing container file will be overwritten but it is an error (DAT__COMEX) to try to overwrite an existing lower-level component.

The ACCESS Field (see SUN/115) specified in the Interface Module for the program must be 'WRITE' or 'UPDATE', otherwise status SUBPAR_ICACM will be returned.

5 Deleting Objects

An object associated with a parameter can be deleted with the DAT_DELET routine – it calls HDS_ERASE to delete a top-level object, and DAT_ERASE to delete lower-level objects recursively. The association is then cancelled.

If an object has not already been associated with the specified parameter, an object name will be obtained from the parameter system.

The ACCESS Field (see SUN/115) specified in the Interface Module for the program must be 'WRITE' or 'UPDATE', otherwise status SUBPAR_ICACM will be returned.

6 Freeing Files

In multi-tasking systems, it is sometimes required that an HDS file is physically updated on disk and freed so that another program can use it. Subroutine DAT_UPDAT may be used to do this. It calls subroutine HDS_FREE for the container file of an object associated with a parameter. If there is no associated object, DAT_UPDAT will just return – no error is reported.

7 Setting Dynamic Defaults

The ADAM parameter system has the concept of 'dynamic defaults' for program parameters. These are values suggested by the program itself and the Interface Module may be set up to cause the dynamic default to be used as the value for a parameter using the VPATH Field (see SUN/115), or given as a suggested value in a prompt, using the PPATH Field (see SUN/115).

Subroutine DAT_DEF can be used to set an HDS object as the dynamic default for a parameter but its usefulness is limited because it has to be given a locator to the object. The locator need only be valid when DAT_DEF is called, the object name is found then and stored as a character string. The string is used when the dynamic default is required.

8 HDSPAR and PAR

There is a very close relationship between HDSPAR, intended for bulk data parameters, and PAR, intended for parameters which are simple, primitive-type values. If a parameter is given a primitive value, The ADAM parameter system creates an HDS object in the program's private parameter file (usually ~/adam/*program_name*). The object has the name of the parameter and contains the given value. This object is then associated with the parameter in the same way as DAT_ASSOC associates any other object.

This relationship allows primitive values and HDS object names to be used interchangeably for parameter values to be accessed by either HDSPAR or PAR subroutines (subject, of course, to suitable type and dimensionality).

Try typing:

```
% hdstrace 3.3
```

HDSTRACE (see SUN/102) would normally expect an HDS object (parameter OBJECT) as input and uses DAT_ASSOC to open it. In this case it opens the parameter file component and displays:

```
HDSTRACE.OBJECT  <_REAL>

   OBJECT         3.3

End of Trace.
```

This feature also allows the program to handle parameters without knowing in advance the type of parameter which will be given.

9 Compiling and Linking

HDSPAR does not itself have any Fortran INCLUDE files but use of the HDSPAR subroutines will almost certainly require the use of the general Starlink Applications Environment INCLUDE file, `SAE_PAR`, and INCLUDE files from other libraries, in particular PAR and HDS,

To set up links to these files, type:

```
% star_dev
% par_dev
% dat_dev
```

Then include the files in the program with statements like:

```
*  Define SAI__OK etc.
      INCLUDE 'SAE_PAR'
*  Define PAR__NULL, PAR__ERROR etc.
      INCLUDE 'PAR_ERR'
*  Define DAT__SZLOC etc.
      INCLUDE 'DAT_PAR'
```

INCLUDE files from other libraries are handled similarly.

Programs using HDSPAR subroutines will be ADAM programs, linked using `alink` or `ilink`. For example:

```
% alink program.f
```

Will compile the Fortran program in file *`program`*`.f` and link it with the ADAM infrastructure libraries.

10 Example

The following contrived program shows how the HDSPAR routines can be used.

```
      SUBROUTINE THDSPAR( STATUS)
*   Exercise the HDSPAR routines
*
*   The program creates a structure (parameter 'STRUCTURE') and an
*   INTEGER array component (parameter 'COMPONENT1') if they do not
*   already exist. (These would normally be a top-level structure and
*   a component of it.)
*   A REAL array is then written to the component and the component
*   set as the dynamic default for the parameter 'INPUT'.
*   A REAL array is then read from the data object associated with the
*   'INPUT' parameter - if the dynamic default is chosen, this will be
*   the INTEGER component just written.
*   The input array is then displayed. (Note that conversion will have
*   occurred in writing a REAL array to an INTEGER component.)
*   The INPUT object is then set as the dynamic default for COMPONENT2
*   and then deleted.
*   COMPONENT2 is then created if it does not exist (it should not
*   exist if the dynamic default is used) and a second attempt made to
*   create it, expecting error DAT__COMEX.

      INCLUDE 'DAT_PAR'
      INCLUDE 'PAR_ERR'

      INTEGER STATUS
      INTEGER NDIMS
      INTEGER DIMS(2), ACTDIMS(2)
      INTEGER I, J
      REAL ARR(2,3)
      CHARACTER*(DAT__SZLOC) LOC1, LOC2

      DATA ARR/1.1,2.2,3.3,4.4,5.5,6.6/

      NDIMS = 2
      DIMS(1) = 2
      DIMS(2) = 3

*   Create a structure if it does not already exist.
      CALL DAT_EXIST( 'STRUCTURE', 'WRITE', LOC1, STATUS )
      IF ( STATUS .EQ. PAR__ERROR ) THEN
         CALL ERR_REP( ' ', 'Structure did not exist', STATUS )
         CALL ERR_FLUSH( STATUS )
         CALL DAT_CREAT( 'STRUCTURE', 'STRUC', 0, DIMS, STATUS )
      ENDIF

*   Cancel the structure parameter.
      CALL DAT_CANCL( 'STRUCTURE', STATUS )

*   Create a component if it does not already exist.
      CALL DAT_EXIST( 'COMPONENT1', 'WRITE', LOC1, STATUS )
      IF ( STATUS .EQ. PAR__ERROR ) THEN
         CALL ERR_ANNUL( STATUS )
         CALL MSG_OUT( ' ', 'Component did not exist.', STATUS )
         CALL DAT_CREAT( 'COMPONENT1', '_INTEGER', 2, DIMS, STATUS )
      ENDIF
```

```
*   Get a locator for the specified component and write to it.
      CALL DAT_ASSOC( 'COMPONENT1', 'WRITE', LOC2, STATUS )
      CALL DAT_PUTNR( LOC2, NDIMS, DIMS, ARR, DIMS, STATUS )

*   Update the disk - we can't see the effect of this.
      CALL DAT_UPDAT( 'COMPONENT1', STATUS )

*   Set the specified component as the dynamic default for 'INPUT'.
      CALL DAT_DEF( 'INPUT', LOC2, STATUS )

*   The above locator may now be annulled.
      CALL DAT_CANCL( 'COMPONENT1', STATUS )

*   Get a locator for the 'INPUT' component and read from it.
      CALL DAT_ASSOC( 'INPUT', 'READ', LOC1, STATUS )
      CALL DAT_GETNR( LOC1, NDIMS, DIMS, ARR, ACTDIMS, STATUS )

*   Display the input data.
      CALL MSG_OUT( ' ', 'Input array is:', STATUS )
      DO 20 J = 1,ACTDIMS(2)
         DO 10 I = 1, ACTDIMS(1)
            CALL MSG_SETR( 'ROW', ARR(I,J) )
            CALL MSG_SETC( 'ROW', ' ' )
   10    CONTINUE
         CALL MSG_OUT( ' ', '^ROW', STATUS )
   20 CONTINUE

*   Set the 'INPUT' object as dynamic default for 'COMPONENT2',
      CALL DAT_DEF( 'COMPONENT2', LOC1, STATUS )

*   Delete the 'INPUT' component.
      CALL DAT_DELET( 'INPUT', STATUS )

*   Check that 'COMPONENT2' does not exist
*   and create it.
      CALL DAT_EXIST( 'COMPONENT2', 'WRITE', LOC1, STATUS )
      IF ( STATUS .EQ. PAR__ERROR ) THEN
         CALL ERR_REP( ' ', 'Component did not exist.', STATUS )
         CALL ERR_FLUSH( STATUS )
         CALL DAT_CREAT( 'COMPONENT2', '_INTEGER', 2, DIMS, STATUS )
      ENDIF

*   Attempt to create it again - expect an error.
      CALL MSG_OUT( ' ', 'Expect error DAT__COMEX.', STATUS )
      CALL DAT_CREAT( 'COMPONENT2', '_INTEGER', 2, DIMS, STATUS )

      END
```

The following interface file could be used. This will cause prompts for parameters `STRUCTURE` and `COMPONENT1`, and take the dynamic defaults for `INPUT` and `COMPONENT2` (unless values are given on the command line).

```
interface THDSPAR
 parameter STRUCTURE
   position 1
   type univ
   access write
   vpath prompt
   ppath default
   default created
 endparameter
 parameter COMPONENT1
   position 2
   type univ
   access update
   vpath prompt
   ppath default
   default created.comp
 endparameter
 parameter INPUT
   position 3
   type univ
   access read
   vpath dynamic
 endparameter
 parameter COMPONENT2
   position 4
   type univ
   vpath dynamic
 endparameter
endinterface
```

The resultant session, accepting suggested values would look like this:

```
% thdspar
STRUCTURE /@created/ >
COMPONENT1 /@created.comp/ >
Input array is:
1 2
3 4
5 6
!! SUBPAR: Error finding component 'COMP' in
!     "/tmp_mnt/mount_nfs/user1/dec/ajc/test/created.sdf"CREATED.COMP
!  Component did not exist
Expect error DAT__COMEX
!! DAT_NEW: Error creating a new HDS component.
!  Application exit status DAT__COMEX, Component already exists
```

A Subroutine Specifications

DAT_ASSOC
Return a locator associated with a parameter

Description:

An HDS locator for the data object associated with the specified parameter is returned. The parameter system will attempt to associate an object if one is not already associated. In the event of a failure, an error message will be displayed and another attempt made (usually by prompting the user). Up to five attempts will be made, after which status PAR__NULL will be returned.

The object will be opened with the appropriate ACCESS mode. If ACCESS is incompatible with the access mode specified for the parameter in the program's Interface File, status SUBPAR__ICACM will be returned.

Invocation:

```
CALL DAT_ASSOC ( PARAM, ACCESS, LOC, STATUS )
```

Arguments:

PARAM=CHARACTER*(*) (given)
Name of program parameter

ACCESS=CHARACTER*(*) (given)
Access mode, 'READ', 'WRITE' or 'UPDATE' (case insignificant)

LOC=CHARACTER*(*) (returned)
Locator to the associated data object

STATUS=INTEGER (given and returned)
Global status

DAT_CANCL
Cancel association between a parameter and a data object

Description:

An existing association between the named parameter and a data system object is cancelled, and the container file closed. This routine will attempt to operate regardless of the given STATUS value.

The parameter enters the CANCELLED state.

Invocation:

```
CALL DAT_CANCL ( PARAM, STATUS )
```

Arguments:

PARAM=CHARACTER∗(∗) (given)
Name of program parameter

STATUS=INTEGER (given and returned)
Global status

DAT_CREAT
Create a data structure component

Description:

An HDS data object is created, as specified by the character string associated with the parameter, and the given type and dimensionality. If the object is a component of a structure, the structure must already exist.

Invocation:

```
CALL DAT_CREAT ( PARAM, TYPE, NDIMS, DIMS, STATUS )
```

Arguments:

PARAM=CHARACTER∗(∗) (given)
Name of program parameter

TYPE=CHARACTER∗(∗) (given)
Type of HDS component. This may be a primitive type or a structure

NDIMS=INTEGER (given)
Number of dimensions of the component

DIMS(∗)=INTEGER (given)
Dimensions of the component

STATUS=INTEGER (given and returned)
Global status

DAT_DEF
Suggest values for parameter

Description:

Set a data-system object as the dynamic default for a parameter. The given locator must be valid when DAT_DEF is called but may be annulled before the dynamic default is used.

Invocation:

```
CALL DAT_DEF ( PARAM, LOC, STATUS)
```

Arguments:

PARAM=CHARACTER∗(∗) (given)
Name of program parameter.

LOC=CHARACTER∗(∗) (given)
Locator to a data object.

STATUS=INTEGER (given and returned)
Global status

DAT_DELET
Delete an object associated with a parameter

Description:

Get an object name and delete the object.

Invocation:

```
CALL DAT_DELET ( PARAM, STATUS )
```

Arguments:

PARAM=CHARACTER∗(∗) (given)

Name of program parameter

STATUS=INTEGER (given and returned)

Global status

DAT_EXIST
Return a locator associated with a parameter

Description:

An HDS locator for the data object associated with the specified parameter is returned. The parameter system will attempt to associate an object if one is not already associated. This operation is identical with DAT_ASSOC except under error conditions. When there is an error, DAT_EXIST reports the error and returns the status value immediately, whereas DAT_ASSOC repeatedly attempts to get a valid locator. If the named object does not exist, status PAR__ERROR is returned.

The object will be opened with the appropriate ACCESS mode. If ACCESS is incompatible with the access mode specified for the parameter in the program's Interface File, status SUBPAR__ICACM will be returned.

Invocation:

```
CALL DAT_EXIST ( PARAM, ACCESS, LOC, STATUS )
```

Arguments:

PARAM=CHARACTER*(*) (given)
program parameter name

ACCESS=CHARACTER*(*) (given)
Access mode, 'READ', 'WRITE' or 'UPDATE' (case insignificant)

LOC=CHARACTER*(*) (returned)
Locator to the associated data object

STATUS=INTEGER (given and returned)
Global status

DAT_UPDAT
Force HDS update

Description:

If there is an HDS object associated with the parameter, force its container file to be freed so that its memory cache coincides with the data on disk and the file is available for other programs to use.

Invocation:

```
CALL DAT_UPDAT ( PARAM, STATUS )
```

Arguments:

PARAM=CHARACTER∗(∗) (given)

Name of program parameter.

STATUS=INTEGER (given and returned)

Global status