

SUN/2.8

Starlink Project  
Starlink User Note 2.8

D.S. Berry  
M.B. Taylor

11th October 2012

Copyright © 2000 Council for the Central Laboratory of the Research Councils

---

**NDG**  
**Routines for Accessing Groups of NDFs**  
**Version 7.0**  
**Programmer's Manual**

---

## **Abstract**

This document describes the routines provided within the NDG subroutine library for accessing groups of NDF data objects.

## Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Interaction Between NDG and GRP</b>	<b>1</b>
<b>3</b>	<b>General overview of the NDG_ system</b>	<b>1</b>
<b>4</b>	<b>An example NDG application</b>	<b>2</b>
<b>A</b>	<b>List of Routines</b>	<b>4</b>
<b>B</b>	<b>Routines for accessing groups of NDFs</b>	<b>5</b>
	NDG_ABPTH . . . . .	6
	NDG_ASEXP . . . . .	7
	NDG ASSO1 . . . . .	9
	NDG_ASSOC . . . . .	10
	NDG_COPY . . . . .	12
	NDG_CPSUP . . . . .	13
	NDG_CREA1 . . . . .	14
	NDG_CREAT . . . . .	15
	NDG_CREP1 . . . . .	17
	NDG_CREXP . . . . .	18
	NDG_GTSUP . . . . .	19
	NDF_MOREG . . . . .	20
	NDG_NDFAS . . . . .	21
	NDG_NDFCO . . . . .	22
	NDG_NDFCP . . . . .	23
	NDG_NDFCR . . . . .	24
	NDG_NDFPL . . . . .	25
	NDG_NDFPR . . . . .	26
	NDG_PROP1 . . . . .	27
	NDG_PTSUP . . . . .	28
	NDG_SETSZ . . . . .	29
<b>C</b>	<b>Routines for accessing NDF Provenance</b>	<b>30</b>
	C.1 Functions Provides: . . . . .	30
	C.2 The PROVENANCE Extension: . . . . .	31
	C.3 The Pre V6 PROVENANCE Extension: . . . . .	31
	C.4 Full Fortran Function Specifications . . . . .	33
	NDG_ADDPROV . . . . .	34
	NDG_BEGPV . . . . .	35
	NDG_COPYPROV . . . . .	36
	NDG_COUNTPROV . . . . .	37
	NDG_ENDPV . . . . .	38
	NDG_FORMATPROV . . . . .	39
	NDG_FREEPROV . . . . .	41
	NDG_GETPROV . . . . .	42
	NDG_HIDEPROV . . . . .	44

NDG_HLTPV . . . . .	45
NDG_ISHIDDENPROV . . . . .	46
NDG_MODIFYPROV . . . . .	47
NDG_PUTPROV . . . . .	48
NDG_READPROV . . . . .	49
NDG_REMOVEPROV . . . . .	50
NDG_ROOTPROV . . . . .	51
NDG_UNHASHPROV . . . . .	52
NDG_UNHIDEPROV . . . . .	53
NDG_WRITEPROV . . . . .	54
C.5 Full C Function Specifications . . . . .	55
ndgAddProv . . . . .	56
ndgCopyProv . . . . .	57
ndgCountProv . . . . .	58
ndgFormatProv . . . . .	59
ndgFreeProv . . . . .	61
ndgGetProv . . . . .	62
ndgHideProv . . . . .	64
ndgIsHiddenProv . . . . .	65
ndgModifyProv . . . . .	66
ndgPutProv . . . . .	67
ndgReadProv . . . . .	68
ndgReadVotProv . . . . .	69
ndgRemoveProv . . . . .	70
ndgRootProv . . . . .	71
ndgUnhashProv . . . . .	72
ndgUnhideProv . . . . .	73
ndgWriteProv . . . . .	74
ndgWriteVotProv . . . . .	75
<b>D Changes Introduced in NDG Version 7.1</b>	<b>76</b>
<b>E Changes Introduced in NDG Version 7.0</b>	<b>76</b>
<b>F Changes Introduced in NDG Version 5.8</b>	<b>76</b>
<b>G Changes Introduced in NDG Version 5.7</b>	<b>76</b>
<b>H Changes Introduced in NDG Version 5.6</b>	<b>77</b>
<b>I Changes Introduced in NDG Version 5.5</b>	<b>77</b>
<b>J Changes Introduced in NDG Version 5.4</b>	<b>77</b>
<b>K Changes Introduced in NDG Version 5.3</b>	<b>77</b>
<b>L Changes Introduced in NDG Version 5.2</b>	<b>77</b>
<b>M Changes Introduced in NDG Version 5.1</b>	<b>77</b>

## 1 Introduction

If an application prompts the user for an NDF using the facilities of the NDG\_ system (see SUN/33), the user may only reply with the name of a single NDF. Some applications allow many input NDFs to be specified and the need to type in every NDF name explicitly each time the program is run can become time consuming. The NDG package provides a means of giving the user the ability to specify a list (or “Group”) of NDFs as a reply to a single prompt for an parameter.

The current version of NDG can process NDFs which are stored as components within an HDS container file, and can also process foreign data formats using the system described in SSN/20.

## 2 Interaction Between NDG and GRP

NDG uses the facilities of the GRP package and users of NDG should be familiar with the content of SUN/150 which describes the GRP package. Groups created by NDG routines should be deleted when no longer needed using GRP\_DELETE.

## 3 General overview of the NDG\_ system

As a broad outline, applications use the NDG\_ package as follows:

- (1) A call is made to NDG\_ASSOC which causes the user to be prompted for a single parameter. This parameter can be of any type. The user replies with a “group expression” (see SUN/150), which contains the names of a group of *existing* NDFs to be used as inputs by the application<sup>1</sup>. For instance, the group expression may be

```
m51_b[23]s1_ds,m51_b[23]s2_ds,m51_b[23]s2?_ds,^files.lis
```

This is a complicated example, probably more complicated than would be used in practice, but it highlights the facilities of the GRP and NDG packages, e.g. wild cards (“?”, “\*” or “[.]” ), lists of files, or indirection through a text file (“^”).

The NDG\_ASSOC routine produces a list of explicit NDF names, which are stored internally within the GRP system.

- (2) What happens next depends on the application, but a common example may be the initiation of a DO loop to loop through the input NDFs (NDG\_ASSOC returns the total number of NDF names in the group).

---

<sup>1</sup>The routine NDG\_ASEXP performs the same function but does not use the parameter system - it expects the group expression to be supplied by the calling routine.

- (3) To access a particular NDF, the application calls routine NDG\_NDFAS supplying an index,  $n$ , within the group (i.e  $n$  is an integer in the range 1 to the group size returned by NDG\_ASSOC). NDG\_NDFAS returns an NDF identifier to the  $n$ th NDF in the group. This identifier can then be used to access the NDF in the normal manner using the NDF\_routines (SUN/33). The identifier should be annulled when it is no longer needed using NDF\_ANNUL in the normal way.
- (4) Once the application has finished processing the group of NDFs, it calls GRP\_DELET which deletes the group, releasing all resources reserved by the group.
- (5) Routine NDG\_ASSOC can also be used to append a list of NDF names obtained from the environment, to a previously defined group.

The routine NDG\_CREAT produces a group containing the names of NDFs which are to be created by the application. The routine NDG\_NDFCR will create a new NDF with a name given by a group member, and returns an NDF identifier to it. Routine NDG\_NDFPR creates a new NDF by propagation from a previously existing NDF, in a similar manner to the NDF routine NDF\_PROP (see SUN/33).

The names of output NDFs given by users usually relate to the input NDF names. When NDG\_CREAT is called, it creates a group of NDF names either by modifying all the names in a specified input group using a “modification element” (see SUN/150), or by getting a list of new names from the user.

- (6) Applications which produce a group of output NDFs could also produce a text file holding the names of the output NDFs. Such a file can be used as input to the next application, using the indirection facility. A text file listing of all the NDFs in a group can be produced by routine GRP\_LIST (or GRP\_LISTF).

See the detailed descriptions of NDG\_ASSOC and NDG\_CREAT below for details of the processing of existing and new NDF names.

## 4 An example NDG application

The following gives a short example of how NDG routines might be used within an ADAM task.

```

SUBROUTINE COPY( STATUS )

* Global Constants:
  INCLUDE 'GRP_PAR'           ! Standard GRP constants

* Local Variables:
  INTEGER GIDIN               ! GRP identifier for group of input NDFs
  INTEGER GIDOUT              ! GRP identifier for group of output NDFs
  INTEGER I                   ! Loop counter
  INTEGER NDFIN               ! NDF identifier for input NDF
  INTEGER NDFOUT              ! NDF identifier for output NDF
  INTEGER NUMIN               ! Number of input NDFs
  INTEGER NUMOUT              ! Number of output NDFs

```

```

        INTEGER STATUS           ! The global status
        LOGICAL FLAG            ! Has group ended with flag character?
*.
*  Inialise the group identifiers to indicate that groups do not have
*  any initial members.
        GIDIN = GRP__NOID
        GIDOUT = GRP__NOID
*  Create group of input NDFs using the parameter IN.
        CALL NDG_ASSOC( 'IN', .TRUE., GIDIN, NUMIN, FLAG, STATUS )
*  Create group of output NDFs using the parameter OUT, which possibly
*  works by modifying the values in the input group.
        CALL NDG_CREAT( 'OUT', GIDIN, GIDOUT, NUMOUT, FLAG, STATUS )
*  Loop over group members.
        DO I = 1, NUMIN
*  Get the identifier for an existing NDF from the input group.
            CALL NDG_NDFAS( GIDIN, I, 'READ', NDFIN, STATUS )
*  Create a new NDF by propagation using the name in the output group.
            CALL NDG_NDFPR( NDFIN, 'Data,Variance,Quality,WCS', GIDOUT, I,
                :           NDFOUT, STATUS )
*  Release NDF resources.
            CALL NDF_ANNUL( NDFIN, STATUS )
            CALL NDF_ANNUL( NDFOUT, STATUS )
        END DO
*  Release GRP resources.
        CALL GRP_DELET( GIDIN, STATUS )
        CALL GRP_DELET( GIDOUT, STATUS )

        END

```

When this program is compiled and run, the user is asked for two ADAM parameters, IN and OUT. Each NDF in the list specified by the IN parameter is simply copied to the corresponding name in the list specified by the OUT parameter. For instance, running

```
copy in=data[12] out=*-new
```

would write new NDFs `data1-new.sdf` and `data2-new.sdf` which were copies of the existing files `data1.sdf` and `data2.sdf`. If `data1` and `data2` do not represent NDF structures, an error will be signalled and the user will be prompted to enter a different value for IN. If any of the elements of the IN list represents an HDS container file which holds multiple NDF structures directly within it, each of these will be added to the list of NDFs to be processed, and the output list will be constructed with a corresponding structure. For instance, if the HDS file `obs.sdf` contains three NDFs at its top level called O1, O2 and O3, then invoking

```
copy in=obs out=obs-copy
```

will result in a new container file `obs-copy.sdf` being written, which contains three NDF structures called O1, O2 and O3.

Note that a few corners have been cut in the above code, in particular checking that the input and output groups have the same size and STATUS testing. Additionally, no action is taken when the FLAG character is given at the end of a group specification – conventionally this would indicate that the user should be allowed to add further members.

## A List of Routines

**CALL NDG\_ASEXP( GRPEXP, VERB, IGRP1, IGRP2, SIZE, FLAG, STATUS )**

*Store names of existing NDFs supplied as a group expression by the calling routine.*

**CALL NDG ASSO1( PARAM, VERB, MODE, INDF, FIELDS, STATUS )**

*Obtain an identifier for a single existing NDF using a specified parameter*

**CALL NDG\_ASSOC( PARAM, VERB, IGRP, SIZE, FLAG, STATUS )**

*Store names of existing NDFs specified through the environment*

**CALL NDG\_CREA1( PARAM, FTYPE, NDIM, LBND, UBND, INDF, NAME, STATUS )**

*Create a single new simple NDF using a specified parameter*

**CALL NDG\_CREAT( PARAM, IGRP0, IGRP, SIZE, FLAG, STATUS )**

*Obtain the names of a group of NDF to be created from the environment*

**CALL NDG\_CREP1( PARAM, FTYPE, NDIM, UBND, INDF, NAME, STATUS )**

*Create a single new primitive NDF using a specified parameter*

**CALL NDG\_GTSUP( IGRP, I, FIELDS, STATUS )**

*Get supplemental information for an NDF*

**CALL NDG\_NDFAS( IGRP, INDEX, MODE, INDF, STATUS )**

*Obtain an NDF identifier for an existing NDF*

**CALL NDG\_NDFCO( INDF1, IGRP, INDEX, INDF2, STATUS )**

*Obtain an NDF identifier for a new NDF created by copying an existing NDF*

**CALL NDG\_NDFCP( IGRP, INDEX, FTYPE, NDIM, UBND, INDF, STATUS )**

*Obtain an NDF identifier for a new primitive NDF*

**CALL NDG\_NDFCR( IGRP, INDEX, FTYPE, NDIM, LBND, UBND, INDF, STATUS )**

*Obtain an NDF identifier for a new simple NDF*

**CALL NDG\_NDFPL( IGRP, INDEX, PLACE, STATUS )**

*Obtain an placeholder for a new NDF*

**CALL NDG\_NDFPR( INDF1, CLIST, IGRP, INDEX, INDF2, STATUS )**

*Obtain an NDF identifier for a new NDF created by propagation from an existing NDF*

**CALL NDG\_PROP1( INDF1, CLIST, PARAM, INDF2, NAME, STATUS )**

*Create a single new NDF by propagation using a specified parameter*

**CALL NDG\_PTSUP( IGRP, I, FIELDS, STATUS )**

*Store supplemental information for an NDF*

**CALL NDG\_SETSZ( IGRP, SIZE, STATUS )**

*Reduces the size of an NDG group*

## **B Routines for accessing groups of NDFs**

---

## NDG\_ABPTH

### Ensure all NDFs in a group have absolute paths

---

**Description:**

Any NDFs in the group that have relative paths are converted to absolute paths on exit.

**Invocation:**

```
CALL NDG_ABPTH( IGRP, STATUS )
```

**Arguments:**

**IGRP = INTEGER (Given)**

The NDG group as returned by NDG\_ASSOC, etc.

**STATUS = INTEGER (Given and Returned)**

The global status.

---

## NDG\_ASEXP

### Store names of existing NDFs supplied as a group expression

---

**Description:**

The supplied group expression is parsed (using the facilities of the GRP routine GRP\_GROUP, see SUN/150) to produce a list of explicit names for existing NDFs which are appended to the end of the supplied group (a new group is created if none is supplied). NDF identifiers for particular members of the group can be obtained using NDG\_NDFAS.

If any of the NDFs specified by the group expression cannot be accessed, an error is reported and STATUS is returned equal to NDG\_NOFIL. If this happens strings holding the name of each bad NDF are appended to the group identified by IGRP1 (so long as IGRP1 is not equal to GRP\_NOID).

**Invocation:**

```
CALL NDG_ASEXP( GRPEXP, VERB, IGRP1, IGRP2, SIZE, FLAG, STATUS )
```

**Arguments:****GRPEXP = CHARACTER \* ( \* ) (Given)**

The group expression specifying the NDF names to be stored in the group.

**VERB = LOGICAL (Given)**

If TRUE then errors which occur whilst accessing supplied NDFs are flushed so that the user can see the details ("verbose" mode). Otherwise, they are annulled and a general "Cannot access file xyz" message is reported instead.

**IGRP1 = INTEGER (Given)**

The identifier of a group to which the names of any inaccessible NDFs will be appended. The group should already have been created by a call to GRP\_NEW, and should be deleted when no longer needed by a call to GRP\_DELETE. If IGRP1 is supplied equal to symbolic constant GRP\_NOID, then no information is stored describing the bad NDFs.

**IGRP2 = INTEGER (Given and Returned)**

The identifier of the group in which the NDF names are to be stored. A new group is created if the supplied value is GRP\_NOID. It should be deleted when no longer needed using GRP\_DELETE.

**SIZE = INTEGER (Returned)**

The total number of NDF names in the returned group IGRP2.

**FLAG = LOGICAL (Returned)**

If the group expression was terminated by the GRP "flag character", then FLAG is returned .TRUE. Otherwise it is returned .FALSE. Returned .FALSE. if an error occurs.

**STATUS = INTEGER (Given and Returned)**

The global status.

**Notes:**

- Any file names containing wildcards are expanded into a list of NDF names. The supplied strings are interpreted by a shell (/bin/tcsh if it exists, otherwise /bin/csh, otherwise /bin/sh), and so may contain shell meta-characters (e.g. twiddle, \$HOME, even command substitution and pipes - but pipe characters "|" need to be escaped using a backslash "\ " to avoid them being interpreted as GRP editing characters).

- Each supplied name may include an HDS path. For instance, `"/home/dsb/mydata.a.c(1).b"` refers to an NDF stored in component `"a.c(1).b"` in the HDS container file `/home/dsb/mydata.sdf`. Note, wild cards are not allowed within HDS component paths (i.e. they are only allowed within the specification of the container file).
- If an HDS object is specified which is not an NDF, then the object will be searched for NDF components. This search is recursive, in that any components of the specified object are also searched. The supplied name will be expanded into a group of names, one for each NDF found within the specified HDS object. Note, NDFs are not themselves searched for other NDFs. That is, the expanded group of names will not include any NDF which is contained within another NDF (i.e. NDFs which are stored as an extension item of another NDF are not included in the group). For instance, if the string `"fred"` is given, the HDS file `fred.sdf` will be searched for NDFs and the returned group will contain references for all NDFs found within `fred.sdf`.
- If the environment variable `NDF_FORMATS_IN` is defined (see SSN/20) then all possible NDFs matching the supplied string are included in the returned group. For instance, if the string `"fred"` is supplied, then the returned group will contain references to all files with basename `fred` which also have a file type specified in `NDF_FORMATS_IN`. If a FITS file `"fred.fit"` exists, and HDS file `"fred.sdf"` also exists (and contains an NDF), then supplying the name `"fred"` will result in both being included in the returned group. If the file `"fred.sdf"` contains a component called `".fit"`, then this will be included in the returned group in place of `"fred.sdf"`.
- NDFs contained within HDS files are opened in order to ensure that they are valid NDFs. The NDF name is returned in IGRP1 if there are no valid NDFs matching a supplied name. No check is made that any foreign data files contain valid NDFs since this would involve a potentially expensive data conversion. So, for instance, `"*.fit"` could pick up FITS catalogues as well as FITS images. If a foreign data file does not contain a valid NDF, an error will be reported when the NDF is accessed using `NDG_NDFAS`.
- Each element in the returned group contains a full specification for an NDF. Several other groups are created by this routine, and are associated with the returned group by means of a GRP "owner-slave" relationship. These supplemental groups are automatically deleted when the returned group is deleted using `GRP_DELETE`. The returned group should not be altered using `GRP` directly because corresponding changes may need to be made to the supplemental groups. Routines `NDG_SETSZ`, `NDG_GTSUP` and `NDG_PTSUP` are provided to manipulate the entire chain of groups. The full chain (starting from the head) is as follows:
  - NDF slice specifications
  - HDS paths
  - File types
  - Base file names
  - Directory paths
  - Full NDF specification (this is the returned group IGRP)

---

## NDG\_ASSO1

### Obtain an identifier for a single existing NDF using a specified parameter

---

**Description:**

This routine is equivalent to NDF\_ASSOC except that it allows the NDF to be specified using a GRP group expression (for instance, its name may be given within a text file, etc). The first NDF in the group expression is returned. Any other names in the group expression are ignored. Supplemental information describing the separate fields in the NDF specification are also returned.

**Invocation:**

```
CALL NDG_ASSO1( PARAM, VERB, MODE, INDF, FIELDS, STATUS )
```

**Arguments:****PARAM = CHARACTER \* ( \* ) (Given)**

Name of the ADAM parameter.

**VERB = LOGICAL (Given)**

If TRUE then errors which occur whilst accessing supplied NDFs are flushed so that the user can see them before re-prompting for a new NDF ("verbose" mode). Otherwise, they are annulled and a general "Cannot access file xyz" message is displayed before re-prompting.

**MODE = CHARACTER \* ( \* ) (Given)**

Type of NDF access required: 'READ', 'UPDATE' or 'WRITE'.

**INDF = INTEGER (Returned)**

NDF identifier.

**FIELDS( 6 ) = CHARACTER \* ( \* ) (Given)**

Each element contains the following on exit:

- (1) NDF slice specifications
- (2) HDS paths
- (3) File types
- (4) Base file names
- (5) Directory paths
- (6) Full NDF specification (this is the returned group - IGRP)

**STATUS = INTEGER (Given and Returned)**

The global status.

---

## NDG\_ASSOC

### Store names of existing NDFs specified through the environment

---

**Description:**

A group expression is obtained from the environment using the supplied parameter. The expression is parsed (using the facilities of the GRP routine GRP\_GROUP, see SUN/150) to produce a list of explicit names for existing NDFs which are appended to the end of the supplied group (a new group is created if none is supplied). If an error occurs while parsing the group expression, the user is re-prompted for a new group expression. NDF identifiers for particular members of the group can be obtained using NDG\_NDFAS.

**Invocation:**

```
CALL NDG_ASSOC( PARAM, VERB, IGRP, SIZE, FLAG, STATUS )
```

**Arguments:****PARAM = CHARACTER\*(\*) (Given)**

The parameter with which to associate the group expression.

**VERB = LOGICAL (Given)**

If TRUE then errors which occur whilst accessing supplied NDFs are flushed so that the user can see them before re-prompting for a new NDF ("verbose" mode). Otherwise, they are annulled and a general "Cannot access file xyz" message is displayed before re-prompting.

**IGRP = INTEGER (Given and Returned)**

The identifier of the group in which the NDF names are to be stored. A new group is created if the supplied value is GRP\_NOID. It should be deleted when no longer needed using GRP\_DELETE.

**SIZE = INTEGER (Returned)**

The total number of NDF names in the returned group.

**FLAG = LOGICAL (Returned)**

If the group expression was terminated by the GRP "flag character", then FLAG is returned true. Otherwise it is returned false. Returned .FALSE. if an error occurs.

**STATUS = INTEGER (Given and Returned)**

The global status.

**Notes:**

- Any file names containing wildcards or "[.]" globbing patterns are expanded into a list of NDF names. The supplied strings are interpreted by a shell (/bin/tcsh if it exists, otherwise /bin/csh, otherwise /bin/sh), and so may contain shell meta-characters (e.g. twiddle, \$HOME, even command substitution and pipes - but pipe characters "|" need to be escaped using a backslash "\" to avoid them being interpreted as GRP editing characters).
- Each supplied name may include an HDS path. For instance, "/home/dsb/mydata.a.c(1).b" refers to an NDF stored in component "a.c(1).b" in the HDS container file /home/dsb/mydata.sdf. Note, wild cards are not allowed within HDS component paths (i.e. they are only allowed within the specification of the container file).
- If an HDS object is specified which is not an NDF, then the object will be searched for NDF components. This search is recursive, in that any components of the specified object are also searched. The supplied name will be expanded into a group of names, one for each NDF found within the specified HDS object. Note, NDFs are not themselves searched for other

NDFs. That is, the expanded group of names will not include any NDF which is contained within another NDF (i.e. NDFs which are stored as an extension item of another NDF are not included in the group). For instance, if the string "fred" is given, the HDS file fred.sdf will be searched for NDFs and the returned group will contain references for all NDFs found within fred.sdf.

- If the environment variable NDF\_FORMATS\_IN is defined (see SSN/20) then only the highest priority file with any give file name is included in the returned group. The priority of a file is determined by its file type. Native NDFs (.sdf) have highest priority. After that, priority decreases along the list of file types specified in NDF\_FORMATS\_OUT. If no file type is given by the user, the highest priority available file type is used. If an explicit file type is given, then that file type is used.
- Care should be taken if a trailing string enclosed in square brackets is appended to the end of the file name. These are interpreted first as a globbing pattern. Thus "fred[12]" would match files with base names "fred1" and "fred2". If the pattern does not match any existing files, then the trailing "[.]" string is next interpreted as a foreign extension specifier. Thus if fred.fit is a multi-extension FITS file, "fred[12]" would be interpreted as the twelfth image extension in fred.fit only if files cannot be found with basenames "fred1" or "fred2".
- NDFs contained within HDS files are opened in order to ensure that they are valid NDFs. The user is notified if there are no valid NDFs matching a supplied name, and they are asked to supply a replacement parameter value. No check is made that any foreign data files contain valid NDFs since this would involve a potentially expensive data conversion. So, for instance, "\*.fit" could pick up FITS catalogues as well as FITS images. If a foreign data file does not contain a valid NDF, an error will be reported when the NDF is accessed using NDG\_NDFAS.
- Each element in the returned group contains a full specification for an NDF. Several other groups are created by this routine, and are associated with the returned group by means of a GRP "owner-slave" relationship. These supplemental groups are automatically deleted when the returned group is deleted using GRP\_DELETE. The returned group should not be altered using GRP directly because corresponding changes may need to be made to the supplemental groups. Routines NDG\_SETSZ, NDG\_GTSUP and NDG\_PTSUP are provided to manipulate the entire chain of groups. The full chain (starting from the head) is as follows:
  - NDF slice specifications
  - HDS paths
  - File types
  - Base file names
  - Directory paths
  - Full NDF specification (this is the returned group IGRP)
- If an error is reported the group is returned unaltered. If no group is supplied, an empty group is returned.
- A null value (!) can be given for the parameter to indicate that no more NDFs are to be specified. The corresponding error is annulled before returning unless no NDFs have been added to the group.
- If the last character in the supplied group expression is a colon (:), a list of the NDFs represented by the group expression (minus the colon) is displayed, but none are actually added to the group. The user is then re-prompted for a new group expression.

---

## NDG\_COPY

### Copy a section of an existing NDG group to a new group

---

**Description:**

This routine extends the functionality of GRP\_COPY when copying elements from a group created by NDG. Such groups have "supplemental information" associated with them that holds further information about each NDF in the group. This function ensures that the returned group also has such supplemental information.

NDG's supplemental information is stored in a chain of "slave groups" that are attached to each other using the facilities of GRP (e.g. see GRP\_SOWN). The supplied group is the lowest level "slave" in this chain.

**Invocation:**

```
CALL NDG_COPY( IGRP, INDXLO, INDXHI, REJECT, IGRP2, STATUS )
```

**Arguments:****IGRP1 = INTEGER (Given)**

A GRP identifier for the input group.

**INDXLO = INTEGER (Given)**

The lowest index to reject or to copy.

**INDXHI = INTEGER (Given)**

The highest index to reject or to copy.

**REJECT = LOGICAL ( Given)**

If reject is .TRUE., then names in the given range are rejected. Otherwise, names in the given range are copied.

**IGRP2 = INTEGER (Returned)**

A GRP identifier for the created group. Returned equal to GRP\_\_NOID if an error occurs.

**STATUS = INTEGER (Given and Returned)**

The global status.

---

## **NDG\_CPSUP**

### **Copy supplemental information for an NDF**

---

**Description:**

Copies an entry with its supplemental information from one group to another, appending it to the end of the output group.

**Invocation:**

```
CALL NDG_CPSUP( IGRP1, I, IGRP2, STATUS )
```

**Arguments:****IGRP1 = INTEGER (Given)**

The NDG group as returned by NDG\_ASSOC, etc.

**I = INTEGER (Given)**

The index, within IGRP1, of the entry to copy.

**IGRP2 = INTEGER (Given)**

The NDG group to which the copied information should be appended.

**STATUS = INTEGER (Given and Returned)**

The global status.

---

## NDG\_CREA1

### Create a single new simple NDF using a specified parameter

---

**Description:**

This routine is equivalent to NDF\_CREAT except that it allows the NDF to be specified using a GRP group expression (for instance, its name may be given within a text file, etc). The first NDF in the group expression is returned. Any other names in the group expression are ignored. Any modification elements in the supplied group expression will be treated literally.

**Invocation:**

```
CALL NDG_CREA1( PARAM, FTYPE, NDIM, LBND, UBND, INDF, NAME, STATUS )
```

**Arguments:**

**PARAM = CHARACTER \* ( \* ) (Given)**

Name of the ADAM parameter.

**FTYPE = CHARACTER \* ( \* ) (Given)**

Full data type of the NDF's DATA component (e.g. '\_DOUBLE' or 'COMPLEX\_REAL').

**NDIM = INTEGER (Given)**

Number of NDF dimensions.

**LBND( NDIM ) = INTEGER (Given)**

Lower pixel-index bounds of the NDF.

**UBND( NDIM ) = INTEGER (Given)**

Upper pixel-index bounds of the NDF.

**INDF = INTEGER (Returned)**

NDF identifier.

**NAME = CHARACTER \* ( \* ) (Returned)**

The full file specification for the NDF.

**STATUS = INTEGER (Given and Returned)**

The global status.

---

## NDG\_CREAT

### Obtain the names of a group of NDF to be created from the environment

---

**Description:**

A group expression is obtained from the environment using the supplied parameter. The expression is parsed (using the facilities of the GRP routine GRP\_GROUP, see SUN/150) to produce a list of explicit NDF names. These names are appended to the group identified by IGRP. The user is re-prompted if an error occurs while parsing the group expression. If IGRP has the value GRP\_NOID on entry, then a new group is created and IGRP is returned holding the new group identifier.

If IGRP0 holds a valid group identifier on entry, then the group identified by IGRP0 is used as the basis for any modification element contained in the group expression obtained from the environment. If IGRP0 holds an invalid identifier (such as GRP\_NOID) on entry then modification elements are included literally in the output group.

**Invocation:**

```
CALL NDG_CREAT( PARAM, IGRP0, IGRP, SIZE, FLAG, STATUS )
```

**Arguments:****PARAM = CHARACTER\*(\*) (Given)**

The parameter with which to associate the group.

**IGRP0 = INTEGER (Given)**

The GRP identifier for the group to be used as the basis for any modification elements. If a valid GRP identifier is supplied, and if the supplied group expression contains a modification element, then:

- the basis token (an asterisk) is replaced by the file basename associated with the corresponding element of the basis group (the "basis NDF").
- if no directory specification is included in the group expression, the directory specification associated with the basis NDF is used.
- if no HDS component path is included in the group expression, the HDS component path associated with the basis NDF (if any) is used. Any required higher level HDS objects are created in the output HDS file by copying the structure of the HDS file containing the basis NDF. Thus if, the basis NDF is fred.a.b(2).c, and the group expression is "\*\_a", then an HDS container file called "fred\_a.sdf" is created by copying fred.sdf and then deleting all NDFs from fred\_a.sdf (unless this has already been done while creating a previous member of the returned group). Other non-NDF components in fred\_a.sdf are retained. This ensures that all necessary structure exists in fred\_a.sdf, so that the NDF fred\_a.a.b(2).c can be created when necessary.

The supplied group will often be created by NDG\_ASSOC, but groups created "by hand" using GRP directly can also be used (i.e. without the supplemental groups created by NDG). In this case, there are no defaults for directory path, file type, or HDS component path, and the basis token ("\*") in the group expression represents the full basis file specification supplied in IGRP0, not just the file basename.

**IGRP = INTEGER (Given and Returned)**

The GRP identifier for the group to which the supplied .sdf files are to be appended.

**SIZE = INTEGER (Returned)**

The total number of file names in the returned group.

**FLAG = LOGICAL (Returned)**

If the group expression was terminated by the GRP "flag" character, then FLAG is returned .TRUE. Otherwise it is returned .FALSE. Returned .FALSE. if an error occurs.

**STATUS = INTEGER (Given and Returned)**

The global status.

**Notes:**

- If an error is reported the group is returned unaltered.
- A null value (!) can be given for the parameter to indicate that no more NDFs are to be specified. The corresponding error is annulled before returning unless no NDFs have been added to the group.
- Explicit file types are included in all the elements of the returned group. This is done because the name may be passed out to a script (eg POLPACK:POLKA) which may change the value of NDF\_FORMATS\_OUT before using the NDF name. If no file type is supplied in the group expression, then the first file type listed in the current value of the NDF\_FORMATS\_OUT environment variable (see SSN/20) is used. If this is "\*" then the file type is copied from the corresponding input file if a modification element was used to specify the output file name (if the NDF was not specified by a modification element, the second file type in NDF\_FORMATS\_OUT is used).
- If the last character in the supplied group expression is a colon (:), a list of the NDFs represented by the group expression (minus the colon) is displayed, but none are actually added to the group. The user is then re-prompted for a new group expression.
- The returned group has no associated groups holding supplemental information (unlike the group returned by NDG\_ASSOC).

---

## NDG\_CREP1

### Create a single new primitive NDF using a specified parameter

---

**Description:**

This routine is equivalent to NDF\_CREP except that it allows the NDF to be specified using a GRP group expression (for instance, its name may be given within a text file, etc). The first NDF in the group expression is returned. Any other names in the group expression are ignored. Any modification elements in the supplied group expression will be treated literally.

**Invocation:**

```
CALL NDG_CREP1( PARAM, FTYPE, NDIM, UBND, INDF, NAME, STATUS )
```

**Arguments:**

**PARAM = CHARACTER \* ( \* ) (Given)**

Name of the ADAM parameter.

**FTYPE = CHARACTER \* ( \* ) (Given)**

Type of the NDF's DATA component (e.g. '\_REAL'). Note that complex types are not permitted when creating a primitive NDF.

**NDIM = INTEGER (Given)**

Number of NDF dimensions.

**UBND( NDIM ) = INTEGER (Given)**

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

**INDF = INTEGER (Returned)**

NDF identifier.

**NAME = CHARACTER \* ( \* ) (Returned)**

The full file specification for the NDF.

**STATUS = INTEGER (Given and Returned)**

The global status.

---

## NDG\_CREXP

### Store the names of a specified group of NDF to be created

---

**Description:**

The supplied group expression is parsed (using the facilities of the GRP routine GRP\_GROUP, see SUN/150) to produce a list of explicit NDF names. No check is made to see if these NDFs exist or not, and any wild-cards in the NDF names are ignored. The names are appended to the group identified by IGRP. If IGRP has the value GRP\_\_NOID on entry, then a new group is created and IGRP is returned holding the new group identifier.

If IGRP0 holds a valid group identifier on entry, then the group identified by IGRP0 is used as the basis for any modification element contained in the supplied group expression. If IGRP0 holds an invalid identifier (such as GRP\_\_NOID) on entry then modification elements are included literally in the output group.

**Invocation:**

```
CALL NDG_CREXP( GRPEXP, IGRP0, IGRP, SIZE, FLAG, STATUS )
```

**Arguments:****GRPEXP = CHARACTER\*(\*) (Given)**

The group expression specifying the NDF names to be stored in the group.

**IGRP0 = INTEGER (Given)**

The GRP identifier for the group to be used as the basis for any modification elements.

**IGRP = INTEGER (Given and Returned)**

The GRP identifier for the group to which the supplied NDF names are to be appended.

**SIZE = INTEGER (Returned)**

The total number of NDF names in the returned group.

**FLAG = LOGICAL (Returned)**

If the group expression was terminated by the GRP "flag" character, then FLAG is returned .TRUE. Otherwise it is returned .FALSE. Returned .FALSE. if an error occurs.

**STATUS = INTEGER (Given and Returned)**

The global status.

---

## **NDG\_GTSUP**

### **Get supplemental information for an NDF**

---

**Description:**

Returns the supplemental information associated with a given entry in an NDG group.

**Invocation:**

```
CALL NDG_GTSUP( IGRP, I, FIELDS, STATUS )
```

**Arguments:****IGRP = INTEGER (Given)**

The NDG group as returned by NDG\_ASSOC, etc.

**I = INTEGER (Given)**

The index of the required entry.

**FIELDS( 6 ) = CHARACTER \* ( \* ) (Returned)**

The supplemental information associated with the entry specified by I. Each element of the returned array contains the following:

1 - NDF slice specification (if any) 2 - HDS path (if any) 3 - File type 4 - Base file name 5 - Directory path 6 - Full NDF specification

If the supplied group is the last group in a GRP owner-slave chain, then this information is obtained from these groups. If any of these groups do not exist, the corresponding elements of the above array are returned holding values formed by parsing the full file specifications in the supplied group. Note, Element 6, the full NDF specification, is obtained directly from the supplied group.

**STATUS = INTEGER (Given and Returned)**

The global status.

---

## NDF\_MOREG

### Search for NDFs within the extensions of a supplied NDF

---

**Description:**

Each extension within the supplied NDF is searched to see if it contains any NDFs. The paths to any such NDFs are appended to the end of the supplied group (a new group is created if none is supplied). NDF identifiers for particular members of the group can be obtained using NDG\_NDFAS.

**Invocation:**

```
CALL NDF_MOREG( INDF, IGRP, SIZE, STATUS )
```

**Arguments:****INDF = INTEGER (Given)**

The identifier for the NDF to be searched.

**IGRP = INTEGER (Given and Returned)**

The identifier of the group in which NDF names are to be stored. A new group is created if the supplied value is GRP\_NOID. It should be deleted when no longer needed using GRP\_DELET.

**SIZE = INTEGER (Returned)**

The total number of NDF names in the returned group IGRP.

**STATUS = INTEGER (Given and Returned)**

The global status.

**Notes:**

- Each element in the returned group contains a full specification for an NDF. Several other groups are created by this routine, and are associated with the returned group by means of a GRP "owner-slave" relationship. These supplemental groups are automatically deleted when the returned group is deleted using GRP\_DELET. The returned group should not be altered using GRP directly because corresponding changes may need to be made to the supplemental groups. Routines NDG\_SETSZ, NDG\_GTSUP and NDG\_PTSUP are provided to manipulate the entire chain of groups. The full chain (starting from the head) is as follows:
- NDF slice specifications
- HDS paths
- File types
- Base file names
- Directory paths
- Full NDF specification (this is the returned group IGRP)

---

## **NDG\_NDFAS**

### **Obtain an NDF identifier for an existing NDF**

---

**Description:**

The routine returns an NDF identifier for an existing NDF. The name of the NDF is held at a given index within a given group. It is equivalent to NDF\_ASSOC.

**Invocation:**

```
CALL NDG_NDFAS( IGRP, INDEX, MODE, INDF, STATUS )
```

**Arguments:****IGRP = INTEGER (Given)**

A GRP identifier for a group holding the names of NDFs. This will often be created using NDF\_ASSOC, but groups created "by hand" using GRP directly (i.e. without the supplemental groups created by NDF\_ASSOC) can also be used.

**INDEX = INTEGER (Given)**

The index within the group at which the name of the NDF to be accessed is stored.

**MODE = CHARACTER \* ( \* ) (Given)**

Type of NDF access required: 'READ', 'UPDATE' or 'WRITE'.

**INDF = INTEGER (Returned)**

NDF identifier.

**STATUS = INTEGER (Given and Returned)**

The global status.

**Notes:**

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

---

**NDG\_NDFCO****Obtain an NDF identifier for a new NDF created by copying an existing NDF**

---

**Description:**

The routine returns an NDF identifier for a new NDF created by copying an existing NDF. The name of the new NDF is held at a given index within a given group. It is equivalent to NDF\_PROP.

**Invocation:**

```
CALL NDG_NDFCO( INDF1, IGRP, INDEX, INDF2, STATUS )
```

**Arguments:****INDF1 = INTEGER (Given)**

Identifier for an existing NDF (or NDF section) to act as a template.

**IGRP = INTEGER (Given)**

A GRP identifier for a group holding the names of NDFs. This will often be created using NDG\_ASSOC, but groups created "by hand" using GRP directly (i.e. without the supplemental groups created by NDG\_ASSOC) can also be used.

**INDEX = INTEGER (Given)**

The index within the group at which the name of the NDF to be accessed is stored.

**INDF2 = INTEGER (Returned)**

Identifier for the new NDF.

**STATUS = INTEGER (Given and Returned)**

The global status.

**Notes:**

- If this routine is called with STATUS set, then a value of NDF\_\_NOID will be returned for the INDF2 argument, although no further processing will occur. The same value will also be returned if the routine should fail for any reason. The NDF\_\_NOID constant is defined in the include file NDF\_PAR.

---

## NDG\_NDFCP

### Obtain an NDF identifier for a new primitive NDF

---

**Description:**

The routine returns an NDF identifier for a new primitive NDF created with the specified attributes. The name of the new NDF is held at a given index within a given group. It is equivalent to NDF\_CREP.

**Invocation:**

```
CALL NDG_NDFCP( IGRP, INDEX, FTYPE, NDIM, UBND, INDF, STATUS )
```

**Arguments:****IGRP = INTEGER (Given)**

A GRP identifier for a group holding the names of NDFs. This will often be created using NDG\_ASSOC, but groups created "by hand" using GRP directly (i.e. without the supplemental groups created by NDG\_ASSOC) can also be used.

**INDEX = INTEGER (Given)**

The index within the group at which the name of the NDF to be created is stored.

**FTYPE = CHARACTER \* ( \* ) (Given)**

Type of the NDF's DATA component (e.g. '\_REAL'). Note that complex types are not permitted when creating a primitive NDF.

**NDIM = INTEGER (Given)**

Number of NDF dimensions.

**UBND( NDIM ) = INTEGER (Given)**

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

**INDF = INTEGER (Returned)**

NDF identifier.

**STATUS = INTEGER (Given and Returned)**

The global status.

---

## NDG\_NDFCR

### Obtain an NDF identifier for a new simple NDF

---

**Description:**

The routine returns an NDF identifier for a new simple NDF created with the specified attributes. The name of the new NDF is held at a given index within a given group. It is equivalent to NDF\_CREAT.

**Invocation:**

```
CALL NDG_NDFCR( IGRP, INDEX, FTYPE, NDIM, LBND, UBND, INDF, STATUS )
```

**Arguments:****IGRP = INTEGER (Given)**

A GRP identifier for a group holding the names of NDFs. This will often be created using NDF\_CREAT, but groups created "by hand" using GRP directly can also be used.

**INDEX = INTEGER (Given)**

The index within the group at which the name of the NDF to be created is stored.

**FTYPE = CHARACTER \* ( \* ) (Given)**

Full data type of the NDF's DATA component (e.g. '\_DOUBLE' or 'COMPLEX\_REAL').

**NDIM = INTEGER (Given)**

Number of NDF dimensions.

**LBND( NDIM ) = INTEGER (Given)**

Lower pixel-index bounds of the NDF.

**UBND( NDIM ) = INTEGER (Given)**

Upper pixel-index bounds of the NDF.

**INDF = INTEGER (Returned)**

NDF identifier.

**STATUS = INTEGER (Given and Returned)**

The global status.

---

## NDG\_NDFPL

### Obtain a placeholder for a new NDF

---

**Description:**

The routine returns a placeholder for a new NDF. The name of the new NDF is held at a given index within a given group. It is equivalent to NDF\_CREPL.

**Invocation:**

```
CALL NDG_NDFPL( IGRP, INDEX, PLACE, STATUS )
```

**Arguments:****IGRP = INTEGER (Given)**

A GRP identifier for a group holding the names of NDFs. This will often be created using NDG\_CREAT, but groups created "by hand" using GRP directly can also be used.

**INDEX = INTEGER (Given)**

The index within the group at which the name of the NDF to be created is stored.

**PLACE = INTEGER (Returned)**

NDF placeholder.

**STATUS = INTEGER (Given and Returned)**

The global status.

---

## NDG\_NDFPR

### Obtain an NDF identifier for a new NDF created by propagation from an existing NDF

---

**Description:**

The routine returns an NDF identifier for a new NDF created by propagation from an existing NDF. The name of the new NDF is held at a given index within a given group. It is equivalent to NDF\_PROP.

**Invocation:**

```
CALL NDG_NDFPR( INDF1, CLIST, IGRP, INDEX, INDF2, STATUS )
```

**Arguments:****INDF1 = INTEGER (Given)**

Identifier for an existing NDF (or NDF section) to act as a template.

**CLIST = CHARACTER \* ( \* ) (Given)**

A comma-separated list of the NDF components which are to be propagated to the new data structure. By default, the HISTORY, LABEL and TITLE components and all extensions are propagated.

**IGRP = INTEGER (Given)**

A GRP identifier for a group holding the names of NDFs. This will often be created using NDG\_ASSOC, but groups created "by hand" using GRP directly (i.e. without the supplemental groups created by NDG\_ASSOC) can also be used.

**INDEX = INTEGER (Given)**

The index within the group at which the name of the NDF to be accessed is stored.

**INDF2 = INTEGER (Returned)**

Identifier for the new NDF.

**STATUS = INTEGER (Given and Returned)**

The global status.

**Notes:**

- If this routine is called with STATUS set, then a value of NDF\_\_NOID will be returned for the INDF2 argument, although no further processing will occur. The same value will also be returned if the routine should fail for any reason. The NDF\_\_NOID constant is defined in the include file NDF\_PAR.

---

## NDG\_PROP1

### Create a single new NDF by propagation using a specified parameter

---

**Description:**

This routine is equivalent to NDF\_PROP except that it allows the NDF to be specified using a GRP group expression (for instance, its name may be given within a text file, etc). The first NDF in the group expression is returned. Any other names in the group expression are ignored. Modification elements use the name of the supplied NDF as the basis name.

**Invocation:**

```
CALL NDG_PROP1( INDF1, CLIST, PARAM, INDF2, NAME, STATUS )
```

**Arguments:****INDF1 = INTEGER (Given)**

Identifier for an existing NDF (or NDF section) to act as a template.

**CLIST = CHARACTER \* ( \* ) (Given)**

A comma-separated list of the NDF components which are to be propagated to the new data structure. By default, the HISTORY, LABEL and TITLE components and all extensions are propagated.

**PARAM = CHARACTER \* ( \* ) (Given)**

Name of the ADAM parameter for the new NDF.

**INDF2 = INTEGER (Returned)**

Identifier for the new NDF.

**NAME = CHARACTER \* ( \* ) (Returned)**

The full file specification for the NDF.

**STATUS = INTEGER (Given and Returned)**

The global status.

---

## NDG\_PTSUP

### Store supplemental information for an NDF

---

**Description:**

Stores the supplied items of supplemental information for a given entry in an NDG group. The GRP groups needed to store this supplemental information are created if they do not already exist, and associated with the supplied group by means of a chain of GRP "owner-slave" relationships. They will be deleted automatically when the supplied group is deleted using GRP\_DELETE.

**Invocation:**

```
CALL NDG_PTSUP( IGRP, I, FIELDS, STATUS )
```

**Arguments:****IGRP = INTEGER (Given)**

The NDG group as returned by NDG\_ASSOC, etc. This should be the last group in a GRP owner-slave chain.

**I = INTEGER (Given)**

The index of the required entry.

**FIELDS( 6 ) = CHARACTER \* ( \* ) (Given)**

The supplemental information to be stored with the entry specified by I. Each element of the supplied array should contain the following:

- (1) NDF slice specification (if any)
- (2) HDS path (if any)
- (3) File type
- (4) Base file name
- (5) Directory path
- (6) Full NDF specification

**STATUS = INTEGER (Given and Returned)**

The global status.

---

## NDG\_SETSZ

### Reduces the size of an NDG group

---

**Description:**

This routine should be used instead of GRP\_SETSZ to set the size of a group created by NDG. It sets the size of the supplied group, and also sets the size of each of the supplemental groups associated with the supplied group.

**Invocation:**

```
CALL NDG_SETSZ( IGRP, SIZE, STATUS )
```

**Arguments:****IGRP = INTEGER (Given)**

The NDG group as returned by NDG\_ASSOC, etc. This should be the last group in a GRP owner-slave chain.

**SIZE = INTEGER (Given)**

The new group size. Must be less than or equal to the size of the smallest group in the chain.

**STATUS = INTEGER (Given and Returned)**

The global status.

## C Routines for accessing NDF Provenance

This section describes all the functions used for reading, writing, modifying and querying provenance information in an NDF.

The provenance information in an NDF encapsulates details of all the other NDFs that were used in the creation of the NDF. The information is hierarchical and includes parents, grandparents, great-grandparents, etc., all the way back to “root ancestors” (a root ancestor is an ancestor NDF that has no recorded parents).

On disk, the provenance information is stored in an NDF extension called “PROVENANCE” (for details see the section “The PROVENANCE Extension” below). The `ndgReadProv` function reads this information and copies it into an in-memory structure for faster access. All the other public functions defined by this module accept an identifier for such an in-memory structure as their first argument. The `ndgWriteProv` function can be used to write the in-memory structure back out to disk as a PROVENANCE extension in an NDF. The in-memory structure should be freed when no longer needed, using `ndgFreeProv`.

### C.1 Functions Provides:

The following public functions are available. There is an equivalent set of F77 routines with names formed by converting the C name to upper case and inserting an underscore after the initial “NDG” string (C and F77 versions are - for the most part - documented individually in separate prologues below):

- `ndgCopyProv`: Create a deep copy of a provenance structure.
- `ndgBegpv`: Begin an NDF provenance block.
- `ndgCountProv`: Return the number of ancestors in a provenance structure.
- `ndgFormatProv`: Format all the information in a provenance structure.
- `ndgEndpv`: End an NDF provenance block.
- `ndgFreeProv`: Free the resources used by a provenance structure.
- `ndgGetProv`: Get information about a specific ancestor.
- `ndgHideProv`: Hide a specific ancestor.
- `ndgHltpv`: Temporarily stop recording NDFs within a provenance block
- `ndgIsHiddenProv`: See if an ancestor is hidden.
- `ndgModifyProv`: Modify information stored for a specific ancestor.
- `ndgPutProv`: Add a new ancestor NDF into a provenance structure.
- `ndgReadProv`: Create a new provenance structure by reading a given NDF.
- `ndgRemoveProv`: Remove ancestors from a provenance structure.
- `ndgRootProv`: Identify root ancestors in a provenance structure.
- `ndgUnhashProv`: Clear the hash code describing the creation of the Provenance.
- `ndgUnhideProv`: Ensure an ancestor is not hidden.
- `ndgWriteProv`: Writes a provenance structure out to an NDF.

Note, within a provenance block, it is possible to prevent selected NDFs receiving provenance by using `ndgHltpv` (`NDG_HLTPV`) to stop the recording of NDF names before the NDFs are accessed, and then calling `ndgHltpv` again afterwards to re-establish the recording of NDF names.

## C.2 The PROVENANCE Extension:

This section describes the format of the NDF extension used to store provenance information in NDG version 6.0 and later (for the pre-V6.0 format, see “The Pre V6 PROVENANCE Extension:”. The PROVENANCE extension in an NDF contains the following components:

**DATA** — A one-dimensional integer array containing descriptions of all the NDFs that were used to create the main NDF. These descriptions are encoded into an opaque set of integer values in order to save time and space, but represent the same set of items described in “The Pre V6 PROVENANCE Extension:” below.

In version 6 of NDG, the provenance extension contained an additional component called “MORE”, which held an optional one-dimensional array of structures containing arbitrary extra information about selected ancestor NDFs. If present, each element of this array contained supplemental information for a single ancestor NDF, and the DATA array contained indices into the MORE array for those ancestors which had additional information.

As of version 7, the information previous held in MORE is now held in the main DATA array.

## C.3 The Pre V6 PROVENANCE Extension:

The format in which provenance information is stored within an NDF’s PROVENANCE extension changed radically at NDG version 6.0, with another more minor change at version 7. Prior to v6.0, the separate numerical values, strings, etc, that form the provenance information were stored in separate HDS components. But for large provenance systems this proved to be inefficient in terms of both processing time and disk space. Therefore, as of NDG v6.0, the numerical values, strings, etc, forming the information are encoded into a single array of integers as described in the previous section. The current version of NDG will read both formats of provenance extension, but always writes the new integer-encoded format.

The rest of this section describes the old format. In addition to documenting the old format, this description serves to illustrate the concepts behind the provenance system. These concepts have not changed - the only thing that has changed is how these concepts are stored within an HDS object.

Pre-V6.0 PROVENANCE extensions in an NDF contains four components: “PARENTS”, “ANCESTORS”, “CREATOR”, “DATE” and “HASH”. The DATE component is a character string holding the date and time at which the information in the provenance extension was last modified. The date is UTC formatted by PSX\_ASCTIME. The ANCESTORS component is a 1D array of “PROV” structures (described below). Each element describes a single NDF that was used in the creation of the main NDF, either directly or indirectly. The PARENTS component is a 1D integer array holding the indices within the ANCESTORS array of the NDFs that are the direct parents of the main NDF. The CREATOR component holds an arbitrary identifier for the software that created the main NDF. The HASH component is an integer that identifies the contents of the current History record in the NDF at the time the PROVENANCE extension was created. This is used to determine which history records to copy into the PROVENANCE extension if the main NDF is used in the creation of another NDF.

Each PROV structure describes a single NDF that was used in the creation of the main NDF, and can contain the following components; “PARENTS”, “DATE”, “PATH”, “CREATOR”, “HISTORY” and “MORE”. If present, the PARENTS component is a 1D integer array holding the the indices within the ANCESTORS array of the direct parents of the ancestor NDF. If PARENTS is not present, the ancestor NDF is a “root” NDF (that is, it has no known parents). If present, the DATE component is a string holding the formatted UTC date at which the provenance information for the ancestor NDF was determined. If this date is not known, the DATE component will not be present (this will be the case, for instance, for all root NDFs). The PATH component will always be present, and is a string holding the full path to the ancestor NDF. This includes any HDS path within the container file, but will not include any NDF or HDS section specifier. Neither will it include the trailing “.sdf” suffix. If present, the MORE component is an arbitrary

HDS structure in which any extra information about the ancestor NDF can be stored. The CREATOR component holds an arbitrary identifier for the software that created the ancestor NDF. The HISTORY component is an array of "HISREC" structures, each containing a copy of a single History record from the NDF described by the PROV structure. Only History records that describe operations performed on the NDF itself are stored (including the record that describes the creation of the NDF). That is, History records inherited from the NDF's own parents are not included.

Each HISREC structure contains the following components (all taken from the corresponding items in the NDF History record): DATE, COMMAND, USER and TEXT. If the history record was created by the default NDF history writing mechanism, the TEXT component will contain a list of environment parameter values used by (or created by) the corresponding command, and another statement of the software that performed the action.

## **C.4 Full Fortran Function Specifications**

---

## NDG\_ADDPROV

### Record multiple input NDFs as ancestors in an output NDF

---

**Description:**

This routine reads provenance from the specified output NDF, and then records each of the specified input NDFs as ancestors within the output provenance. It then writes the modified provenance back out to the output NDF.

It is a simplified wrapper for NDG\_READPROV, NDG\_PUTPROV AND NDG\_WRITEPROV. It is more restrictive than use of NDF\_PUTPROV since it stores no extra information ("MORE") with any of the ancestors, and does not force any of the ancestors to be root ancestors.

**Invocation:**

```
CALL NDG_ADDPROV( INDF, CREATOR, NNDF, NDFS, STATUS )
```

**Arguments:****INDF = INTEGER (Given)**

An identifier for the output NDF.

**CREATOR = CHARACTER \* ( \* ) (Given)**

A text identifier for the software that created INDF (usually the name of the calling application). The format of the identifier is arbitrary, but the form "PACKAGE:COMMAND" is recommended.

**NNDF = INTEGER (Given)**

The number of input NDFs.

**NDFS( NNDF ) = INTEGER (Given)**

An array of identifiers for the input NDFs.

**STATUS = INTEGER (Given and Returned)**

The global status.

---

## **NDG\_BEGPV**

### **Begin an NDF provenance block**

---

**Description:**

This routine should be called to mark the start of an NDF provenance block. The block should be ended by a matching call to `NDG_ENDPV`. See `NDG_ENDPV` for more details.

Note - provenance blocks must not be nested.

**Invocation:**

```
CALL NDG_BEGPV( STATUS )
```

**Arguments:**

**STATUS = INTEGER (Given and Returned)**

The global status.

---

## NDG\_COPYPROV

### Copy a Provenance structure, optionally removing any hidden ancestors

---

**Description:**

This routine produces a deep copy of the supplied Provenance structure, and then optionally uses NDG\_REMOVEPROV to remove any hidden ancestors from the copy.

**Invocation:**

```
CALL NDG_COPYPROV( IPROV, CLEANSE, IPROV2, STATUS )
```

**Arguments:****IPROV = INTEGER (Given)**

An identifier for a structure holding the provenance information read from an NDF, as returned by NDG\_READPROV.

**CLEANSE = LOGICAL (Given)**

If `.TRUE.`, then any ancestors which have been hidden using NDG\_HIDEPROV are removed from the returned Provenance structure (see NDG\_REMOVEPROV).

**IPROV2 = INTEGER (Returned)**

An identifier for the new Provenance structure, which should be freed using NDG\_FREEPROV when no longer needed.

**STATUS = INTEGER (Given and Returned)**

The global status.

---

## NDG\_COUNTPROV

### Return the number of ancestors in a provenance structure

---

**Description:**

This routine returns the number of ancestors described in the supplied provenance structure.

**Invocation:**

```
CALL NDG_COUNTPROV( IPROV, COUNT, STATUS )
```

**Arguments:****IPROV = INTEGER (Given)**

An identifier for a structure holding the provenance information read from an NDF, as returned by NDG\_READPROV.

**COUNT = INTEGER (Returned)**

The number of ancestors in the supplied provenance structure.

**STATUS = INTEGER (Given and Returned)**

The global status.

---

## NDG\_ENDPV

### End an NDF provenance block

---

**Description:**

This routine should be called to mark the end of an NDF provenance block. The block should have been started by a matching call to NDG\_BEGPV. Note, provenance blocks must not be nested.

During a provenance block, a list is maintained of all the existing NDFs that have had their Data array mapped (either in read or update mode) during the block. Another list is maintained of all the NDFs that have been written (either existing NDFs accessed in update mode or new NDFs) during the block.

When the block ends, the provenance information within each NDF in the second may be modified to include all the NDFs in the first list as parents. Whether or not this occurs is controlled by the AUTOPROV environment variable. If AUTOPROV is set to '1' then the input NDFs are added to the provenance information in the output NDF. If AUTOPROV is set to anything other than '1' then the output provenance is not updated. If AUTOPROV is not set at all, the default behaviour is different for native and foreign format NDFs. For native format output NDFs, the provenance will be updated if one or more of the input NDFs contains a PROVENANCE extension (foreign format input NDFs are ignored). The provenance within foreign format output NDFs is never updated unless AUTPOPROV is set explicitly to '1' (this is to avoid the potentially costly process of format conversion).

**Invocation:**

```
CALL NDG_ENDPV( CREATR, STATUS )
```

**Arguments:****CREATR = CHARACTER \* ( \* ) (Given)**

An identifier for the software that created INDF1 (usually the name of the calling application). The format of the identifier is arbitrary, but the form "PACKAGE:COMMAND" is recommended.

**STATUS = INTEGER (Given and Returned)**

The global status.

---

## NDG\_FORMATPROV

### Format the information in a provenance structure

---

**Description:**

This routine returns an AST KeyMap holding a set of text strings containing information taken from the supplied provenance structure.

The returned KeyMap has an entry with key "0" that describes the NDF from which the provenance was read. It also has an entry describing each ancestor NDF. These entries have keys "1", "2", "3", etc, up to the number of ancestors in the NDF.

Each of these entries contains a pointer to another AST KeyMap which may contain any subset of the following entries (all of which, except for HISTORY, are strings):

"ID" - the integer index within the ancestors array (zero for the main NDF).

"PATH" - The full path or base name for the NDF (see "base").

"DATE" - The date of creation of the NDF.

"CREATOR" - The software item that created the NDF.

"PARENTS" - A comma-separated list of indices into the ancestors array that identifies the direct parents of the NDF.

"MORE" - A summary of the contents of the MORE structure associated with the NDF.

"HISTORY" - A vector entry holding one or more KeyMaps. Each KeyMap contains items that describe an action performed on the ancestor. The actions are stored in chronological order within the vector entry. The last KeyMap in the vector describes the action that created the ancestor NDF. Any earlier KeyMaps in the vector describe any subsequent actions performed on the ancestor NDF prior to it being used in the creation of its parent. Each KeyMap contains the following scalar character entries (all taken from the corresponding record in the NDF HISTORY component):

- "DATE": The date and time of the action (e.g. "2009-JUN-24 14:00:53.752" ).
- "COMMAND": An indication of the command that performed the action (e.g. "WCSATTRIB (KAPPA 1.10-6)" ).
- "USER": The user name that performed the action (e.g. "dsb").
- "TEXT": The full text of the NDF history record. This is arbitrary, but for NDFs created by Starlink software it will usually include environment parameter values, and the full path of the command that performed the action.

Finally, the returned KeyMap has an entry with key "MXLEN" that is again a pointer to another KeyMap with the same entries listed above (except that it has no "HISTORY" entry). However, this time the entries are integers, not strings, and holds the maximum field width used to format the strings.

**Invocation:**

```
CALL NDG_FORMATPROV( IPROV, BASE, KEYMAP, STATUS )
```

**Arguments:****IPROV = INTEGER (Given)**

An identifier for a structure holding the provenance information read from an NDF, as returned by NDG\_READPROV.

**BASE = LOGICAL (Given)**

If .TRUE., then the PATH field in the returned KeyMap holds the base name of each NDF rather than the full path.

**KEYMAP = INTEGER (Returned)**

A pointer to the returned AST KeyMap.

**STATUS = INTEGER (Given and Returned)**

The global status.

---

## **NDG\_FREEPROV**

### **Free a structure holding provenance information**

---

**Description:**

This routine frees the resources used to hold a provenance structure.

**Invocation:**

```
CALL NDG_FREEPROV( IPROV, STATUS )
```

**Arguments:****IPROV = INTEGER (Given and Returned)**

An identifier for a structure holding the provenance information read from an NDF, as returned by NDG\_READPROV. Returned holding NDG\_\_NULL.

**STATUS = INTEGER (Given and Returned)**

The global status.

**Notes:**

- This function attempts to execute even if an error has already occurred.

---

## NDG\_GETPROV

### Create a KeyMap holding information about an ancestor

---

**Description:**

This routine returns information about a specified ancestor in the supplied provenance structure.

**Invocation:**

```
CALL NDG_GETPROV( IPROV, IANC, KM, STATUS )
```

**Arguments:****IPROV = INTEGER (Given)**

An identifier for a structure holding the provenance information read from an NDF, as returned by NDG\_READPROV.

**IANC = INTEGER (Given)**

The index of the ancestor NDF for which information should be returned. A value of zero will result in information being returned that describes the NDF from which the provenance information was read. Otherwise, the IANC value is used as an index into the ANCESTORS array. No error is reported if IANC is too large, but a null identifier will be returned as the function value.

**KM = INTEGER (Returned)**

A pointer to an AST KeyMap containing entries with the following keys and values:

- "PATH": A string holding the path of the ancestor NDF.
- "DATE": A string holding the formatted UTC date and time at which the provenance information for the ancestor NDF was recorded.
- "CREATOR": A string identifying the software that created the ancestor NDF.
- "PARENTS": A 1D vector of integers that are the indices of the immediate parents of the ancestor.
- "MORE": A KeyMap containing any extra information that has been stored with the ancestor.
- "HISTORY": A vector entry holding one or more KeyMaps. Each KeyMap contains items that describe an action performed on the ancestor. The actions are stored in chronological order within the vector entry. The last KeyMap in the vector describes the action that created the ancestor NDF. Any earlier KeyMaps in the vector describe any subsequent actions performed on the ancestor NDF prior to it being used in the creation of its parent. Each KeyMap contains the following scalar character entries (all taken from the corresponding record in the NDF HISTORY component):
  - "DATE": The date and time of the action (e.g. "2009-JUN-24 14:00:53.752" ).
  - "COMMAND": An indication of the command that performed the action (e.g. "WCSATTRIB (KAPPA 1.10-6)" ).
  - "USER": The user name that performed the action (e.g. "dsb").
  - "TEXT": The full text of the NDF history record. This is arbitrary, but for NDFs created by Starlink software it will usually include environment parameter values, and the full path of the command that performed the action.

If the specified ancestor does not have any of these items of information, then the corresponding entry will not be present in the returned KeyMap. For instance, if the ancestor has no immediate parent NDFs, then the "PARENTS" entry will not be present in the KeyMap. A NULL pointer

will be returned if the NDF has no provenance extension, or if "ianc" is outside the bounds of the ANCESTORS array (and is not zero). The returned KeyMap pointer should be annulled when it is no longer needed, either by calling `astAnnul` explicitly, or by relying on `astEnd` to annul it (together with all the other AST Objects created in the current AST Object context).

**STATUS = INTEGER (Given and Returned)**

The global status.

---

## NDG\_HIDEPROV

### Hide an ancestor in a provenance structure

---

**Description:**

This function flags a specified ancestor as "hidden". The only effect this has is that the ancestor will not be included in Provenance structures created by the NDG\_COPYPROV function.

**Invocation:**

```
CALL NDG_HIDEPROV( IPROV, IANC, STATUS )
```

**Arguments:****IPROV = INTEGER (Given)**

An identifier for a structure holding the provenance information read from an NDF, as returned by NDG\_READPROV.

**IANC = INTEGER (Given)**

The index of the ancestor NDF to be hidden. The value is used as an index into the ANCESTORS array. An error will be reported if the value is too large, or is less than 1 (the main NDF cannot be hidden).

**STATUS = INTEGER (Given and Returned)**

The global status.

---

## NDG\_HLTPV

### Temporarily halt an NDF provenance block

---

**Description:**

This routine can be called to stop subsequently accessed NDFs being added to the list of NDFs that will receive updated provenance information when NDG\_ENDPV is called to end the current provenance block.

**Invocation:**

```
CALL NDG_HLTPV( NEW, OLD, STATUS )
```

**Arguments:****NEW = LOGICAL (Read)**

The new required provenance-recording state.

**OLD = LOGICAL (Returned)**

The provenance-recording state on entry to this routine.

**STATUS = INTEGER (Given and Returned)**

The global status.

**Notes:**

- A .FALSE. provenance-recording state means that any subsequently accessed NDFs will not be added to the list of NDFs to receive updated provenance information when NDG\_ENDPV is called.
- A .TRUE. provenance-recording state means that any subsequently accessed NDFs are added to the list of NDFs to receive updated provenance information when NDG\_ENDPV is called.

---

## NDG\_ISHIDDENPROV

### See if an ancestor in a provenance structure is hidden

---

**Description:**

This function returns a `TRUE` value for `HIDDEN` if the specified ancestor has been hidden. See `NDG_HIDEPROV` and `NDG_COPYPROV`.

**Invocation:**

```
CALL NDG_ISHIDDENPROV( IPROV, IANC, HIDDEN, STATUS )
```

**Arguments:****IPROV = INTEGER (Given)**

An identifier for a structure holding the provenance information read from an NDF, as returned by `NDG_READPROV`.

**IANC = INTEGER (Given)**

The index of the ancestor NDF to be checked. The value is used as an index into the `ANCESTORS` array. An error will be reported if the value is too large, or is less than 0.

**HIDDEN = LOGICAL (Returned)**

`.TRUE.` if the ancestor is hidden.

**STATUS = INTEGER (Given and Returned)**

The global status.

---

## **NDG\_MODIFYPROV**

### **Modify the information stored for a particular ancestor**

---

**Description:**

This routine modifies the information stored for a given ancestor in the supplied provenance structure. The new values to store are supplied in an AST KeyMap such as returned by `NDG_GETPROV`.

**Invocation:**

```
CALL NDG_MODIFYPROV( IPROV, IANC, KM, STATUS )
```

**Arguments:****IPROV = INTEGER (Given)**

An identifier for a structure holding the provenance information read from an NDF, as returned by `NDG_READPROV`.

**IANC = INTEGER (Given)**

The index of the ancestor NDF for which information should be modified. A value of zero will result in information about the NDF specified by `INDF` being modified. Otherwise, the `IANC` value is used as an index into the `ANCESTORS` array. An error is reported if `IANC` is too large.

**KM = INTEGER (Given)**

A pointer to an AST KeyMap containing the values to store. Entries with the following keys are recognised:

- "PATH": A string holding the path of the ancestor NDF.
- "DATE": A string holding the formatted UTC date and time at which the provenance information for the ancestor NDF was recorded.
- "CREATOR": A string identifying the software that created the ancestor NDF.
- "MORE": A KeyMap containing extra information to store with the ancestor.

If the "DATE", "CREATOR" or "MORE" components are missing then corresponding item of information will be deleted from the provenance extension. An error is reported if the supplied KeyMap has no "PATH" entry. Note, the `PARENTS` list and `HISTORY` information stored with the specified ancestor cannot be modified (any "PARENTS" or "HISTORY" component in the supplied `HDS` structure will be ignored).

**STATUS = INTEGER (Given and Returned)**

The global status.

---

## NDG\_PUTPROV

### Add an NDF to the list of ancestors

---

**Description:**

This routine modifies the supplied provenance structure to indicate that a given NDF was used in the creation of the NDF associated with the supplied provenance structure.

**Invocation:**

```
CALL NDG_PUTPROV( IPROV, INDF, MORE, ISROOT, STATUS )
```

**Arguments:****IPROV = INTEGER (Given)**

An identifier for a structure holding the provenance information read from an NDF, as returned by NDG\_READPROV.

**INDF = INTEGER (Given)**

An identifier for an NDF that is to be added into the list of ancestor NDFs in the supplied provenance information.

**MORE = INTEGER (Given)**

A pointer to an AstKeyMap holding arbitrary additional information about the new ancestor NDF, and how it was used in the creation of the output NDF.

**ISROOT = LOGICAL (Given)**

If TRUE, then the new ancestor NDF will be treated as a root NDF. That is, any provenance information in the supplied NDF is ignored. If FALSE, then any provenance information in the NDF is copied into the supplied provenance structure. The new ancestor NDF is then only a root NDF if it contains no provenance information.

**STATUS = INTEGER (Given and Returned)**

The global status.

---

## NDG\_READPROV

### Read the provenance information from an NDF

---

**Description:**

This function reads the information stored in the "PROVENANCE" extension of an NDF, storing it in a memory-resident structure for faster access. An identifier for this structure is returned, and can be passed to other NDG provenance functions to manipulate the contents of the structure.

If the NDF has no provenance information (for instance, if it is a newly created NDF), or if no NDF is supplied, the returned structure will contain just the supplied creator name (which may be blank), and an empty ancestor list.

The structure should be freed when it is no longer needed by calling NDG\_FREEPROV.

The structure should be freed when it is no longer needed by calling NDG\_FREEPROV.

**Invocation:**

```
CALL NDG_READPROV( INDF, CREATOR, IPROV, STATUS )
```

**Arguments:****INDF = INTEGER (Given)**

An identifier for the NDF containing the provenance information to be read. This may be NDF\_NOID, in which case a new provenance structure with the supplied creator name and an empty ancestor list will be created and returned.

**CREATOR = CHARACTER \* ( \* ) (Given)**

A text identifier for the software that created INDF (usually the name of the calling application). The format of the identifier is arbitrary, but the form "PACKAGE:COMMAND" is recommended.

**IPROV = INTEGER (Returned)**

An identifier for the structure holding the provenance information read from the NDF. NDG\_NULL is returned if an error occurs.

**STATUS = INTEGER (Given and Returned)**

The global status.

---

## NDG\_REMOVEPROV

### Remove one or more ancestors from a provenance structure

---

**Description:**

This routine removes one or more ancestors from the supplied provenance structure. The direct parents of the removed ancestor are assigned to the direct children of the removed ancestor. Note, any history records stored in the removed ancestors are lost.

**Invocation:**

```
CALL NDG_REMOVEPROV( IPROV, NANC, IANC, STATUS )
```

**Arguments:****IPROV = INTEGER (Given)**

An identifier for a structure holding the provenance information read from an NDF, as returned by NDG\_READPROV.

**NANC = INTEGER (Given)**

The length of the ANC array.

**ANC( \* ) = INTEGER (Given)**

An array holding the indices of the ancestor NDFs to be removed. Each supplied value must be at least 1, and must be no more than the number of ancestors in the provenance extension (as returned by NDG\_COUNTPROV). An error is reported otherwise. The supplied list is sorted into decreasing order before use so that the highest index ancestor is removed first.

**STATUS = INTEGER (Given and Returned)**

The global status.

---

## **NDG\_ROOTPROV**

### **Identify the root ancestors in a provenance structure**

---

**Description:**

This routine searches the supplied provenance structure for root ancestors, and returns information about them. An ancestor is a root ancestor if it does not itself have any ancestors.

**Invocation:**

```
CALL NDG_ROOTPROV( IPROV, KM, STATUS )
```

**Arguments:****IPROV = INTEGER (Given)**

An identifier for a structure holding the provenance information read from an NDF, as returned by `NDG_READPROV`.

**KM = INTEGER (Returned)**

A pointer to an AST KeyMap containing an entry for each root ancestor. The key associated with each entry is the path to the NDF and the value of the entry is an integer that gives the position of the root ancestor within the list of all ancestors. This integer value can be supplied to `ndgGetProv` in order to get further information about the root ancestor. The first ancestor NDF has an index of one. An index of zero refers to the NDF from which the provenance information was read.

**STATUS = INTEGER (Given and Returned)**

The global status.

---

## NDG\_UNHASHPROV

### Clear the hash code describing the creation of the Provenance

---

**Description:**

Each ancestor in a Provenance structure may contain a copy of the History information stored in the associated ancestor NDF. Storing the complete History component from each ancestor NDF would be very wasteful since the NDF History component will usually contain not only records of operations performed on the ancestor NDF, but also all History records inherited from the "primary" NDF (i.e. the NDF from which the ancestor was propagated). Since these inherited History records will already be stored with other ancestors in the Provenance structure, it is not necessary to store them again. However, this means that when we add a new parent into a Provenance structure using NDG\_PUTPROV, NDG needs some way of knowing which records within the new NDF are unique to the NDF (and should thus be stored in the Provenance structure), and which were inherited from earlier ancestors (and will thus already be stored in the Provenance structure). The solution is for each PROVENANCE extension to include a "creator" hash code for the History record that describes the creation of the NDF. When an NDF is supplied to NDG\_PUTPROV, each History record, starting with the most recent, is copied from the NDF into the Provenance structure, until a History record is found which has a hash code equal to the creator hash code in the NDF. The copying of history records then stops since all earlier history records will already be present in the Provenance structure.

This routine clears the creator hash code in the supplied Provenance structure, so that a new one will be calculated when the Provenance structure is written to an NDF using NDG\_WRITEPROV. This is useful for instance if the Provenance was written to the NDF using NDG\_WRITEPROV before the NDF History record was completed. In this case, you would probably want to re-read the Provenance from the NDF, use this function to clear the creator hash code, and then re-write the Provenance to the NDF, thus forcing a new creator hash code to be stored in the NDF.

**Invocation:**

```
CALL NDG_UNHASHPROV( IPROV, STATUS )
```

**Arguments:****IPROV = INTEGER (Given)**

An identifier for a structure holding the provenance information read from an NDF, as returned by NDG\_READPROV.

**STATUS = INTEGER (Given and Returned)**

The global status.

---

## **NDG\_UNHIDEPROV**

### **Un-hide an ancestor in a provenance structure**

---

**Description:**

This function ensures that a given ancestor is not flagged as "hidden". See *NDG\_HIDEPROV* and *NDG\_COPYPROV*.

**Invocation:**

```
CALL NDG_UNHIDEPROV( IPROV, IANC, STATUS )
```

**Arguments:****IPROV = INTEGER (Given)**

An identifier for a structure holding the provenance information read from an NDF, as returned by *ndgReadProv*

**IANC = INTEGER (Given)**

The index of the ancestor NDF to be un-hidden. The value is used as an index into the *ANCESTORS* array. An error will be reported if the value is too large, or is less than 0.

**STATUS = INTEGER (Given and Returned)**

The global status.

**Notes:**

- No error is reported if the specified ancestor is not currently hidden (in which case this function returns without action).

---

## NDG\_WRITEPROV

### Write provenance information to an NDF

---

**Description:**

This routine writes the contents of the supplied provenance structure out to a given NDF, replacing any existing provenance information in the NDF.

**Invocation:**

```
CALL NDG_WRITEPROV( IPROV, INDF, WHDEF, STATUS )
```

**Arguments:****IPROV = INTEGER (Given)**

An identifier for a structure holding the provenance information read from an NDF, as returned by NDG\_READPROV.

**INDF = INTEGER (Given)**

Identifier for the NDF in which to store the provenance information.

**WHDEF = INTEGER (Given)**

The correct recording of history information within the PROVENANCE extension requires that the current history record within the supplied NDF at the time this function is called, describes the creation of the NDF. Very often, an application will not itself add any history to the NDF, but will instead rely on the automatic recording of default history provided by the NDF library. Normally, default history is recorded when the NDF is released from the NDF system (e.g. using NDF\_ANNUL or NDF\_END). So if this function is called prior to the release of the NDF (which it normally will be), then the default history information will not yet have been recorded, resulting in incorrect information being stored in the PROVENANCE extension. For this reason, the WHDEF argument is supplied. If it is set to `.TRUE.`, a check is made to see if default history has already been stored in the NDF. If `.FALSE.`, default history is stored in the NDF before going on to create the PROVENANCE extension. Applications that do not use the default history recording mechanism, but instead store their own history information, should supply `.FALSE.` for WHDEF, and should also ensure that history information has been stored in the NDF before calling this routine.

**STATUS = INTEGER (Given and Returned)**

The global status.

## **C.5 Full C Function Specifications**

---

## **ndgAddProv**

### **Record multiple input NDFs as ancestors in an output NDF**

---

**Description:**

This routine reads provenance from the specified output NDF, and then records each of the specified input NDFs as ancestors within the output provenance. It then writes the modified provenance back out to the output NDF.

It is a simplified wrapper for `ndgReadProv`, `ndgPutProv` and `ndgWriteProv`. It is more restrictive than use of `ndgPutProv` since it stores no extra information ("MORE") with any of the ancestors, and does not force any of the ancestors to be root ancestors.

**Invocation:**

```
ndgAddProv( int indf, const char *creator, int nndf, int *ndfs, int *status )
```

**Arguments:****indf**

An identifier for the output NDF.

**creator**

A text identifier for the software that created INDF (usually the name of the calling application). The format of the identifier is arbitrary, but the form "PACKAGE:COMMAND" is recommended. This value is only used if the the NDF does not contain any existing provenance information.

**nndf**

The number of input NDFs.

**ndfs**

A pointer to an array of identifiers for the input NDFs.

**status**

The global status.

---

## **ndgCopyProv**

### **Copy a Provenance structure, optionally removing any hidden ancestors**

---

**Description:**

This function produces a deep copy of the supplied Provenance structure, and then optionally uses `ndgRemoveProv` to remove any hidden ancestors from the copy. A pointer to the copy is returned.

**Invocation:**

```
NdgProvenance *ndgCopyProv( NdgProvenance *prov, int cleanse, int *status )
```

**Arguments:****prov**

A pointer to the provenance information to be copied.

**cleanse**

If non-zero, then any ancestors which have been hidden using `ndgHideProv` are removed from the returned Provenance structure (see `ndgRemoveProv`).

**status**

The global status.

**Returned Value:**

A pointer to the new Provenance structure, which should be freed using `ndgFreeProv` when no longer needed.

---

## **ndgCountProv**

### **Return the number of ancestors in a provenance structure**

---

**Description:**

This function returns the number of ancestors described in the supplied provenance structure.

**Invocation:**

```
result = ndgCountProv( NdgProvenance *prov, int *status )
```

**Arguments:****prov**

An identifier for a structure holding the provenance information as returned by `ndgReadProv` or `ndgReadVotProv`.

**status**

The global status.

**Returned Value:**

The number of ancestor NDFs stored in the supplied provenance structure, or zero if an error occurs.

---

## ndgFormatProv

### Format the information in a provenance structure

---

**Description:**

This function returns an AST KeyMap holding a set of text strings containing information taken from the supplied provenance structure.

The returned KeyMap has an entry with key "0" that describes the NDF from which the provenance was read. It also has an entry describing each ancestor NDF. These entries have keys "1", "2", "3", etc, up to the number of ancestors in the NDF.

Each of these entries contains a pointer to another AST KeyMap which may contain any subset of the following entries (all of which are strings):

"ID" - the integer index within the ancestors array (zero for the main NDF).

"PATH" - The full path or base name for the NDF (see "base").

"DATE" - The date of creation of the NDF.

"CREATOR" - The software item that created the NDF.

"PARENTS" - A comma-separated list of indices into the ancestors array that identifies the direct parents of the NDF.

"MORE" - A summary of the contents of the MORE structure associated with the NDF.

"HISTORY" - A vector entry holding one or more KeyMaps. Each KeyMap contains items that describe an action performed on the ancestor. The actions are stored in chronological order within the vector entry. The last KeyMap in the vector describes the action that created the ancestor NDF. Any earlier KeyMaps in the vector describe any subsequent actions performed on the ancestor NDF prior to it being used in the creation of its parent. Each KeyMap contains the following scalar character entries (all taken from the corresponding record in the NDF HISTORY component):

- "DATE": The date and time of the action (e.g. "2009-JUN-24 14:00:53.752" ).
- "COMMAND": An indication of the command that performed the action (e.g. "WCSATTRIB (KAPPA 1.10-6)" ).
- "USER": The user name that performed the action (e.g. "dsb").
- "TEXT": The full text of the NDF history record. This is arbitrary, but for NDFs created by Starlink software it will usually include environment parameter values, and the full path of the command that performed the action.

Finally, the returned KeyMap has an entry with key "MXLEN" that is again a pointer to another KeyMap with the same entries listed above (except that it has no "HISTORY" entry). However, this time the entries are integers, not strings, and holds the maximum field width used to format the strings.

**Invocation:**

```
void ndgFormatProv( NdgProvenance *prov, int base, AstKeyMap **keymap, int *status
)
```

**Arguments:****prov**

An identifier for a structure holding the provenance information as returned by `ndgReadProv` or `ndgReadVotProv`.

**base**

If non-zero, then the PATH field in the returned KeyMap holds the base name of each NDF rather than the full path.

**keymap**

A location at which to returned a pointer to the returned AST KeyMap.

**status**

The global status.

---

## **ndgFreeProv**

### **Free a structure holding provenance information**

---

**Description:**

This function frees the resources used to hold a provenance structure.

**Invocation:**

```
NdgProvenance *ndgFreeProv( NdgProvenance *prov, int *status )
```

**Arguments:****prov**

A pointer to the provenance information to be freed.

**status**

The global status.

**Returned Value:**

A NULL pointer is returned.

**Notes:**

- This function attempts to execute even if an error has already occurred.

---

## **ndgGetProv**

### **Create a KeyMap holding information about an ancestor**

---

**Description:**

This function returns information about a specified ancestor in the supplied provenance structure.

**Invocation:**

```
result = ndgGetProv( NdgProvenance *prov, int ianc, int *status )
```

**Arguments:****prov**

An identifier for a structure holding the provenance information as returned by `ndgReadProv` or `ndgReadVotProv`.

**ianc**

The index of the ancestor NDF for which information should be returned. A value of zero will result in information being returned that describes the NDF from which the provenance information was read. Otherwise, the "ianc" value is used as an index into the ANCESTORS array. No error is reported if "ianc" is too large, but a NULL pointer will be returned as the function value.

**status**

The global status.

**Returned Value:****A pointer to an AST KeyMap**

The keymap contains entries with the following keys and values:

- "PATH": A string holding the path of the ancestor NDF.
- "DATE": A string holding the formatted UTC date and time at which the provenance information for the ancestor NDF was recorded.
- "CREATOR": A string identifying the software that created the ancestor NDF.
- "PARENTS": A 1D vector of integers that are the indices of the immediate parents of the ancestor.
- "MORE": A KeyMap containing any extra information that has been stored with the ancestor.
- "HISTORY": A vector entry holding one or more KeyMaps. Each KeyMap contains items that describe an action performed on the ancestor. The actions are stored in chronological order within the vector entry. The last KeyMap in the vector describes the action that created the ancestor NDF. Any earlier KeyMaps in the vector describe any subsequent actions performed on the ancestor NDF prior to it being used in the creation of its parent. Each KeyMap contains the following scalar character entries (all taken from the corresponding record in the NDF HISTORY component):
  - "DATE": The date and time of the action (e.g. "2009-JUN-24 14:00:53.752" ).
  - "COMMAND": An indication of the command that performed the action (e.g. "WCSATTRIB (KAPPA 1.10-6)" ).
  - "USER": The user name that performed the action (e.g. "dsb").
  - "TEXT": The full text of the NDF history record. This is arbitrary, but for NDFs created by Starlink software it will usually include environment parameter values, and the full path of the command that performed the action.

If the specified ancestor does not have any of these items of information, then the corresponding entry will not be present in the returned KeyMap. For instance, if the ancestor has no immediate parent NDFs, then the "PARENTS" entry will not be present in the KeyMap. A NULL pointer will be returned if the NDF has no provenance extension, or if "ianc" is outside the bounds of the ANCESTORS array (and is not zero). The returned KeyMap pointer should be annulled when it is no longer needed, either by calling `astAnnul` explicitly, or by relying on `astEnd` to annul it (together with all the other AST Objects created in the current AST Object context).

---

## **ndgHideProv**

### **Hide an ancestor in a provenance structure**

---

**Description:**

This function flags a specified ancestor as "hidden". The only effect this has is that the ancestor will not be included in Provenance structures created by the `ndgCopyProv` function.

**Invocation:**

```
ndgHideProv( NdgProvenance *prov, int ianc, int *status )
```

**Arguments:****prov**

An identifier for a structure holding the provenance information as returned by `ndgReadProv` or `ndgReadVotProv`.

**ianc**

The index of the ancestor NDF to be hidden. The value is used as an index into the ANCESTORS array. An error will be reported if the value is too large, or is less than 1 (the main NDF cannot be hidden).

**status**

The global status.

---

## **ndgIsHiddenProv**

### **See if an ancestor in a provenance structure is hidden**

---

**Description:**

This function returns a non-zero value if the specified ancestor has been hidden. See `ndgHideProv` and `ndgCopyProv`.

**Invocation:**

```
int ndgIsHiddenProv( NdgProvenance *prov, int ianc, int *status );
```

**Arguments:****prov**

An identifier for a structure holding the provenance information as returned by `ndgReadProv` or `ndgReadVotProv`.

**ianc**

The index of the ancestor NDF to be checked. The value is used as an index into the ANCESTORS array. An error will be reported if the value is too large, or is less than 0.

**status**

The global status.

---

## **ndgModifyProv**

### **Modify the information stored for a particular ancestor**

---

**Description:**

This function modifies the information stored for a given ancestor in the supplied provenance structure. The new values to store are supplied in an Ast KeyMap such as returned by `ndgGetProv`.

**Invocation:**

```
void ndgModifyProv( NdgProvenance *prov, int ianc, AstKeyMap *km, int *status )
```

**Arguments:****prov**

An identifier for a structure holding the provenance information as returned by `ndgReadProv` or `ndgReadVotProv`.

**ianc**

The index of the ancestor NDF for which information should be modified. A value of zero will result in information being modified for the NDF from which the the supplied provenance structure was read. Otherwise, the "ianc" value is used as an index into the ANCESTORS array. An error is reported if "ianc" is too large.

**km** A pointer to an AST KeyMap containing the values to store. Entries with the following keys are recognised:

- "PATH": A string holding the path of the ancestor NDF.
- "DATE": A string holding the formatted UTC date and time at which the provenance information for the ancestor NDF was recorded.
- "CREATOR": A string identifying the software that created the ancestor NDF.
- "MORE": A KeyMap containing extra information to store with the ancestor.

If the "DATE", "CREATOR" or "MORE" components are missing then corresponding item of information will be deleted from the provenance extension. An error is reported if the supplied KeyMap has no "PATH" entry. Note, the PARENTS list and HISTORY information stored with the specified ancestor cannot be modified (any "PARENTS" or "HISTORY" component in the supplied HDS structure will be ignored).

**status**

The global status.

---

## **ndgPutProv**

### **Add an NDF to the list of ancestors**

---

**Description:**

This function modifies the supplied provenance structure to indicate that a given NDF was used in the creation of the NDF associated with the supplied provenance structure.

**Invocation:**

```
ndgPutProv( NdgProvenance *prov, int indf, AstKeyMap *more, int isroot, int *status
)
```

**Arguments:****prov**

An identifier for a structure holding the provenance information as returned by `ndgReadProv` or `ndgReadVotProv`.

**indf**

An identifier for an NDF that is to be added into the list of ancestor NDFs in the supplied provenance information.

**more**

A pointer to an `AstKeyMap` holding arbitrary additional information about the new ancestor NDF, and how it was used in the creation of the output NDF. A NULL pointer can be supplied if required.

**isroot**

If non-zero, then the new ancestor NDF will be treated as a root NDF. That is, any provenance information in the supplied NDF is ignored. If zero, then any provenance information in the NDF is copied into the supplied provenance structure. The new ancestor NDF is then only a root NDF if it contains no provenance information.

**status**

The global status.

---

## **ndgReadProv**

### **Read the provenance information from an NDF**

---

**Description:**

This function reads the information stored in the "PROVENANCE" extension of an NDF, storing it in a memory-resident structure for faster access. A pointer that identifies this structure is returned, and can be passed to other NDG provenance functions to manipulate the contents of the structure.

If the NDF has no provenance information (for instance, if it is a newly created NDF), or if no NDF is supplied, the returned structure will contain just the supplied creator name (which may be blank), and an empty ancestor list.

The structure should be freed when it is no longer needed by calling `ndgFreeProv`.

**Invocation:**

```
NdgProvenance *ndgReadProv( int indf, const char *creator, int *status )
```

**Arguments:****indf**

An identifier for the NDF containing the provenance information to be read. This may be `NDF_NOID`, in which case a new provenance structure with the supplied creator name and an empty ancestor list will be created and returned.

**creator**

A text identifier for the software that created INDF (usually the name of the calling application). The format of the identifier is arbitrary, but the form "PACKAGE:COMMAND" is recommended. This value is only used if the the NDF does not contain any existing provenance information.

**status**

The global status.

**Returned Value:**

A pointer that identifies the structure holding the provenance information read from the NDF. Note, this is not a genuine pointer to the structure and should not be de-referenced. A NULL pointer is returned if an error occurs.

---

## **ndgReadVotProv**

### **Read the provenance information from a VOTABLE**

---

**Description:**

This function reads provenance information from a string of XML text read from a VOTABLE, storing it in a memory-resident structure for faster access. A pointer that identifies this structure is returned, and can be passed to other NDG provenance functions to manipulate the contents of the structure.

If the XML text has no provenance information, the returned structure will contain just the supplied creator name (which may be blank), and an empty ancestor list.

The structure should be freed when it is no longer needed by calling `ndgFreeProv`.

**Invocation:**

```
NdgProvenance *ndgReadVotProv( const char *xml, const char *path, const char *creator,
int *status )
```

**Arguments:****xml**

Pointer to a null terminated string holding XML read from a VOTABLE. The provenance information is read from the first element found in the text that has the opening tag:

```
"<GROUP name="PROVENANCE" utype="hds_type:PROVENANCE">"
```

This is the form produced by function `ndgWriteVotProv`.

**path**

The path to the file from which the XML provenance text was read.

**creator**

A text identifier for the software that created the NDF with which the provenance is associated (usually the name of the calling application). The format of the identifier is arbitrary, but the form "PACKAGE:COMMAND" is recommended. This value is only used if the the supplied XML text does not contain any provenance information.

**status**

The global status.

**Returned Value:**

A pointer that identifies the structure holding the provenance information read from the VOTABLE. Note, this is not a genuine pointer to the structure and should not be de-referenced. A NULL pointer is returned if an error occurs.

## **ndgRemoveProv**

### **Remove one or more ancestors from a provenance structure**

---

**Description:**

This routine removes one or more ancestors from the supplied provenance structure. The direct parents of the removed ancestor are assigned to the direct children of the removed ancestor. Note, any history records stored in the removed ancestors are lost.

**Invocation:**

```
void ndgRemoveProv( NdgProvenance *prov, int nanc, int *anc, int *status )
```

**Arguments:****prov**

An identifier for a structure holding the provenance information as returned by ndgReadProv or ndgReadVotProv.

**nanc**

The length of the "anc" array.

**anc**

Pointer to an array holding the indices of the ancestor NDFs to be removed. Each supplied value must be at least 1, and must be no more than the number of ancestors in the provenance extension (as returned by ndgCountProv). An error is reported otherwise. The supplied list is sorted into decreasing order before use so that the highest index ancestor is removed first.

**status**

The global status.

---

## **ndgRootProv**

### **Identify the root ancestors in a provenance structure**

---

**Description:**

This function searches the supplied provenance structure for root ancestors, and returns information about them. An ancestor is a root ancestor if it does not itself have any ancestors.

**Invocation:**

```
AstKeyMap *ndgRootProv( NdgProvenance *prov, int *status )
```

**Arguments:****prov**

An identifier for a structure holding the provenance information as returned by `ndgReadProv` or `ndgReadVotProv`.

**status**

The global status.

**Returned Value:**

A pointer to an AST KeyMap containing an entry for each root ancestor. The key associated with each entry is the path to the NDF and the value of the entry is an integer that gives the position of the root ancestor within the list of all ancestors. This integer value can be supplied to `ndgGetProv` in order to get further information about the root ancestor. The first ancestor NDF has an index of one. An index of zero refers to the NDF from which the provenance information was read.

## ndgUnhashProv

### Clear the hash code describing the creation of the Provenance

---

**Description:**

Each ancestor in a Provenance structure may contain a copy of the History information stored in the associated ancestor NDF. Storing the complete History component from each ancestor NDF would be very wasteful since the NDF History component will usually contain not only records of operations performed on the ancestor NDF, but also all History records inherited from the "primary" NDF (i.e. the NDF from which the ancestor was propagated). Since these inherited History records will already be stored with other ancestors in the Provenance structure, it is not necessary to store them again. However, this means that when we add a new parent into a Provenance structure using `ndgPutProv`, NDG needs some way of knowing which records within the new NDF are unique to the NDF (and should thus be stored in the Provenance structure), and which were inherited from earlier ancestors (and will thus already be stored in the Provenance structure). The solution is for each PROVENANCE extension to include a "creator" hash code for the History record that describes the creation of the NDF. When an NDF is supplied to `ndgPutProv`, each History record, starting with the most recent, is copied from the NDF into the Provenance structure, until a History record is found which has a hash code equal to the creator hash code in the NDF. The copying of history records then stops since all earlier history records will already be present in the Provenance structure.

This routine clears the creator hash code in the supplied Provenance structure, so that a new one will be calculated when the Provenance structure is written to an NDF using `ndgWriteProv`. This is useful for instance if the Provenance was written to the NDF using `ndgWriteProv` before the NDF History record was completed. In this case, you would probably want to re-read the Provenance from the NDF, use this function to clear the creator hash code, and then re-write the Provenance to the NDF, thus forcing a new creator hash code to be stored in the NDF.

**Invocation:**

```
void ndgUnhashProv( NdgProvenance *prov, int *status )
```

**Arguments:****prov**

An identifier for a structure holding the provenance information as returned by `ndgReadProv` or `ndgReadVotProv`.

**status**

The global status.

---

## **ndgUnhideProv**

### **Un-hide an ancestor in a provenance structure**

---

**Description:**

This function ensures that a given ancestor is not flagged as "hidden". See `ndgHideProv` and `ndgCopyProv`.

**Invocation:**

```
ndgUnhideProv( NdgProvenance *prov, int ianc, int *status )
```

**Arguments:****prov**

An identifier for a structure holding the provenance information as returned by `ndgReadProv` or `ndgReadVotProv`.

**ianc**

The index of the ancestor NDF to be un-hidden. The value is used as an index into the ANCESTORS array. An error will be reported if the value is too large, or is less than 0.

**status**

The global status.

**Notes:**

- No error is reported if the specified ancestor is not currently hidden (in which case this function returns without action).

---

## **ndgWriteProv**

### **Write provenance information to an NDF**

---

**Description:**

This function writes the contents of the supplied provenance structure out to a given NDF, replacing any existing provenance information in the NDF.

**Invocation:**

```
void ndgWriteProv( NdgProvenance *prov, int indf, int whdef, int *status )
```

**Arguments:****prov**

A pointer to the provenance information to be written out.

**indf**

Identifier for the NDF in which to store the provenance information.

**whdef**

The correct recording of history information within the PROVENANCE extension requires that the current history record within the supplied NDF at the time this function is called, describes the creation of the NDF. Very often, an application will not itself add any history to the NDF, but will instead rely on the automatic recording of default history provided by the NDF library. Normally, default history is recorded when the NDF is released from the NDF system (e.g. using `ndfAnnul` or `ndfEnd`). So if this function is called prior to the release of the NDF (which it normally will be), then the default history information will not yet have been recorded, resulting in incorrect information being stored in the PROVENANCE extension. For this reason, the "whdef" argument is supplied. If it is set to a non-zero value, a check is made to see if default history has already been stored in the NDF. If not, default history is stored in the NDF before going on to create the PROVENANCE extension. Applications that do not use the default history recording mechanism, but instead store their own history information, should supply a zero value for "whdef" and should also ensure that history information has been stored in the NDF before calling this function.

**status**

The global status.

---

## **ndgWriteVotProv**

### **Write provenance information out to a VOTABLE**

---

**Description:**

This function writes the contents of the supplied provenance structure out as a text string holding an XML snippet suitable for inclusion in a VOTABLE.

**Invocation:**

```
const char *ndgWriteVotProv( NdgProvenance *prov, int *status )
```

**Arguments:****prov**

A pointer to the provenance information to be written out.

**status**

The global status.

**Returned Value:**

A pointer to a dynamically allocated string holding the XML text, or NULL if an error occurs. The string should be freed using `astFree` when it is no longer needed. The text will contain a single top level element with the following opening tag:

```
"<GROUP name="PROVENANCE" utype="hds_type:PROVENANCE">"
```

The HDS structure of an NDF PROVENANCE extension is replicated using PARAM elements to hold primitive values and GROUP elements to hold structures. The "utype" attributes are used to hold the corresponding HDS data types.

## D Changes Introduced in NDG Version 7.1

- A bug has been fixed that resulted in the NDF AUTOHISTORY facility being switched off inside an NDG “group history” block. For instance, this caused all KAPPA applications to ignore any value set for environment variable NDF\_AUTO\_HISTORY.
- New routine NDG\_ABPTH added to convert a group of NDF paths from relative to absolute (with respect to the current working directory).
- Routine NDG\_GTSUP now returns field values formed by parsing the file specifications in the supplied group, if the supplied group has no supplemental information. Previously, it returned blank field values in such cases.

## E Changes Introduced in NDG Version 7.0

- After provenance has been read into memory using `ndgReadProv`, any extra information associated with an ancestor NDF via the “MORE” component is now stored in an AST KeyMap rather than a temporary HDS structure.

*NOTE, THIS HAS REQUIRED CHANGES TO THE API OF THREE FUNCTIONS: `ndgGetProv`, `ndgPutProv` and `ndgModifyProv`. THESE FUNCTIONS NO LONGER HAVE A HDS LOCATOR ARGUMENT. In addition, `ndgAntmp` has been withdrawn.*

- The format in which provenance information is stored within an NDF PROVENANCE extension has changed. Any “MORE” information is now included in the opaque array of integers containing all the other information, rather than being stored in a separate HDS structure. Version 7 of NDG will read all earlier formats of provenance information but always writes the new format.

## F Changes Introduced in NDG Version 5.8

- Added functions for appending GRP group contents to default history records written by the NDF library.
- New function `ndgHltpv` (NDG\_HLTPV) allows selected NDFs to be exempted from a provenance block.

## G Changes Introduced in NDG Version 5.7

- The Provenance handling API has changed to use a temporary in-memory structure to describe a PROVENANCE extension, rather than every function accessing the NDF on disk.
- The PROVENANCE extension now allows the recording of NDF History with each ancestor.

## H Changes Introduced in NDG Version 5.6

- A new C function `ndgRmprvs` has been added, that removes multiple provenance ancestors from an NDF.
- A new C function `ndgGtprvk` has been added, that returns provenance information for a given ancestor in the form of an AST KeyMap.

## I Changes Introduced in NDG Version 5.5

- A new routine `NDG_MOREG` has been added which searches the extensions of a supplied NDF for encapsulated NDFs, appending the paths to such NDFs to a supplied GRP group.

## J Changes Introduced in NDG Version 5.4

- Added new routines: `NDG_MDPRV`.

## K Changes Introduced in NDG Version 5.3

- Added routines for storing and retrieving provenance information: `NDG_PTPRV`, `NDG_RTPRV`, `NDG_GTPRV`, `NDG_BEGPV` and `NDG_ENDPV`.

## L Changes Introduced in NDG Version 5.2

- The C routine `ndg1_regsb.c` has been changed to avoid warning message about "tmpnam" when linking applications on Redhat Linux systems.

## M Changes Introduced in NDG Version 5.1

- A new routine `NDG_ASEXP` has been added which allows a group of existing NDFs to be created from a group expression supplied as a subroutine argument. This is very similar to `NDG_ASSOC` except that `NDG_ASSOC` gets the group expression from the parameter system instead of from its argument list.