**SSN/20.3**

Starlink Project
Starlink System Note 20.3

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

17th July 2000

# Adding Format Conversion Facilities to the NDF Data Access Library
# Version 1.6
# Developer's Guide

## Abstract

The NDF (*Extensible N-Dimensional Data Format*) data access library (SUN/33) provides a programming interface and a data model for astronomical applications software. It is based on a flexible underlying data format HDS (SUN/92) and a set of conventions for structuring data within HDS.

This document describes facilities for extending the range of data formats accessible via the NDF library, to include any arbitrary "foreign" format for which a conversion utility can be defined. This gives NDF-based applications access to a potentially wide range of data formatting possibilities, including data compression. The intended readership of this document includes a) Developers working with data formats and associated conversion utilities; b) programmers who anticipate using the NDF library to access foreign data; and c) knowledgeable users who need to access data stored in new (*e.g.* personal) formats.

## Contents

# 1 INTRODUCTION

One of the best things about standards is the large number you have to choose from. "Standard" ways of storing data proliferate in all areas of computing, and astronomy is no exception.

Given this, there is a natural desire to write software that can cope with more than one data format, but this can be a major undertaking. This document describes features provided by the NDF library (SUN/33) that help to make it a little easier.

## 1.1 Philosophy

There are two main ways of writing software that can read and write multiple data formats. Perhaps the most obvious is to incorporate a knowledge of the data model used by each format into the data access library and to have it make the appropriate calls (*e.g.* to different lower-level data access libraries) according to the format being used. This approach is generally quite efficient, but it often presents serious difficulties in practice.

The main problem is that the data access software rapidly becomes extremely complex. Usually, a single person will maintain the data access library used by a suite of applications so, if multiple data formats are to be supported, that person must become expert in all the formats required. Since the resulting library will be the only means of access to these formats, it must be very sophisticated and anticipate every requirement (even if many of the features are, in fact, never used). Given the number and complexity of formats in use, the range of data models they present, and the rate at which they change, this is a near impossible task. It is not surprising, therefore, that few systems attempt to support more than a couple of formats in this way.

An alternative approach is to interpose format conversion software between the original data and the application. This is potentially less efficient, but modern computing equipment makes this less of a problem than it once was. The great advantage is that it decouples the problem of format conversion from that of data access. It also splits the provision of software for accessing each different data format into a series of separate tasks. This makes it possible to support a wide variety of formats relatively easily.

## 1.2 The Format Conversion Approach

The NDF data access library follows this latter approach by allowing format conversion utilities to be added to it, thereby allowing it to access a range of "foreign" data (*i.e.* data which are not stored in the native NDF format). This has the following advantages:

- The full range of normal NDF data access operations can be supported – reading, writing, updating, deleting, reshaping, *etc.*
- Format conversion utilities can be added to already-built software. Thus you can add the ability to read new data formats to standard applications, without having to re-build them.
- Format conversion utilities can be written and added by anyone, so the problem of understanding and accessing a range of different formats can be shared. Particular problems can be tackled by whoever best understands them. This makes the NDF library capable of accessing data in formats completely unknown to its original author.

- Because it is easy to add new format conversion utilities, they do not always need to be very sophisticated. Instead, they can be invented or adapted to tackle new problems as they arise. Many of the difficulties encountered when converting a complicated data format into another one with a different data model can be ignored, unless they happen to be relevant to the situation at hand.

The main reason that this approach can be used is because of the relatively sophisticated formatting possibilities and data model presented by the NDF library, with its in-built extensibility. This makes it possible to convert foreign data into NDF format and back again without losing information, while the opposite process is not always possible.

## 1.3 How Format Conversion Operates

To illustrate how this system works, suppose an application which uses the NDF library wants to access an existing NDF data structure, but the person running it only has data available in a foreign format. The following outlines the sequence of operations that might occur:

(1) The NDF library will first obtain the name of the dataset to be accessed in the normal way, *e.g.* by prompting (alternatively, the application could obtain the name and pass it to the library, but the two methods are equivalent here).

(2) The library will check whether the data are stored in native NDF format. If so, it will access them directly. If not, it will next identify which foreign format they are stored in. This is done by inspecting the extension on the file name (for instance, the file name `mydata.fit` might designate a dataset stored in FITS format).

(3) The library will then look to see if a format conversion utility has been defined to convert from the foreign format into native NDF format. Assuming one has, it will invoke it, causing the data to be converted and written into a scratch object in native NDF format.

(4) The scratch object will then be accessed as normal. The application need not know that it hasn't been given a normal NDF. In addition, all references which the application makes to the dataset name will use the original (foreign) file name, so the user usually need not be aware that conversion has occurred either.

(5) When the dataset is released by the application, the scratch object will be deleted (in fact, this is optional – see §3.2). If it has been modified, a format conversion utility will be sought, and invoked, to perform back-conversion of the modified data before this occurs.

A rather similar sequence of events might occur when creating a new dataset (*e.g.* as output from an application), except that the format conversion stage on input would not be required.

These steps are an exact analogue of the conversions that the NDF and HDS libraries perform transparently whenever an application attempts to (*e.g.*) access an integer data array as floating point, or to read data previously written on a machine which uses a different number representation. The only difference is that format conversion utilities are not a permanent part of the data access software, but are invoked as separate processes which communicate through files rather than via memory.[1] This makes it possible to add and remove them as required.

[1]With the file caching available on modern operating systems, this distinction is actually rather blurred.

# 2 SETTING UP FOR FORMAT CONVERSION

## 2.1 Name Your Formats

The first step in setting up the NDF library to access foreign data formats is to define a name for each foreign format to be recognised, and to associate a file extension with each of these names. The file extension will be used to determine which format a file is written in.

This is done by defining the environment variable called NDF_FORMATS_IN to contain a format list, such as the following:

```
setenv NDF_FORMATS_IN 'FITS(.fit),FIGARO(.dst),IRAF(.imh)'
```

This is a comma-separated list of format specifications, where each specification consists of a format name (*e.g.* FITS) with an associated file extension (*e.g.* '.fit') in parentheses.

This list serves two purposes. First, it defines the set of formats and associated file extensions to be recognised when accessing input[2] datasets. This means, for instance, that if a dataset name such as:

```
run66.fit
```

were given to the NDF library, it would recognise it as a FITS format file and try to carry out the appropriate conversion.

The list also defines a search order for foreign data formats. This means that if the dataset name supplied had been simply:

```
run66
```

then the NDF library would first look for a native format NDF with this name (*i.e.* in the file `run66.sdf`). If this was not found, it would then look for a file called `run66.fit`, then `run66.dst` and then `run66.imh`, stopping when the first one was found and associating the appropriate data format with it. If none of the files existed, a "file not found" error would result.

Note that the ability to select sections from pre-existing NDF datasets (see SUN/33) is also available when accessing foreign data files, so that entering:

```
run66.fit(100.0~50.0)
```

or

```
run66(100.0:200.0,10:512)
```

would result in the same actions as above to locate a suitable file and to convert its format, with the required section then being extracted from the converted NDF and passed to the application.

[2]Strictly speaking, NDF_FORMATS_IN defines the formats recognised when accessing *pre-existing* datasets. Although it is possible to update and write to such datasets, it is nevertheless convenient to refer to them as "input" datasets.

## 2.2 Rules and Regulations

You may define up to 50 foreign formats to be recognised in this way, and may give them any names and file extensions you like (apart from the format name NDF and the file extension '.sdf' which are reserved for the native NDF format). Format names are not case sensitive, although file extensions are if that makes sense for the host file system (*e.g.* they are case sensitive on UNIX). File extensions should always begin with a '.' and appear in parentheses following the associated format name. There is no individual limit on the length of a format name or file extension, but the entire format list is limited to 1024 characters.

Note that the same foreign format name and/or file extension may appear more than once in the NDF_FORMATS_IN list. The first occurrence takes precedence when searching for files. Thus, you could associate different file extensions with the same format name to define synonyms for file extensions.

## 2.3 Defining Conversion Commands

For each foreign format which appears in the NDF_FORMATS_IN list, you should also provide commands to perform the necessary format conversions to and/or from the native NDF format. These commands are also defined by means of environment variables.

Taking the FITS format (above) as an example, this means defining up to two commands – one for converting from FITS format to NDF format and the other for converting back again, such as the following:

```
setenv NDF_FROM_FITS 'fitsin in=^dir^name^type out=^ndf'
setenv NDF_TO_FITS   'fitsout in=^ndf out=^dir^name^type'
```

Here, the names of two environment variables have been formed by prefixing 'NDF_FROM_' and 'NDF_TO_' to the foreign format name (in upper case) and each of these variables has been set to contain a command which performs the appropriate format conversion (in this case by invoking two conversion utilities called "fitsin" and "fitsout", which we assume to exist).

Ideally, you would define both of these commands. However, if you only want to support conversion in one direction, then it is quite acceptable to omit either of them. The commands are only accessed when the occasion to use them arises, so no error will result if they are omitted but never used.

When needed, the conversion commands you define will be interpreted (in a separate process) by a command interpreter appropriate to the host operating system.[3] The commands are actually invoked by passing them to the C run time library "system" function, and they may therefore use any components of the environment which are inherited through that interface. Typically this means that such things as the default directory and environment variables are available to these commands.

Before the commands are invoked, the NDF library will perform token substitution on them, in order to insert the names of the actual datasets to be processed. The tokens used to represent these datasets are, in fact, *message tokens* – identical to those used by the MERS and EMS libraries (SUN/104 and SSN/4) and commonly used when reporting errors and other messages from

[3]On UNIX, this will be the "sh" (Bourne) shell.

within applications. They are used in conversion commands in exactly the same way (they appear in the example commands above prefixed with the '^' substitution character), and the NDF library defines a set of them for this purpose, as follows:

| **Token** | **Value** |
|---|---|
| **dir** | Directory in which the foreign file resides |
| **name** | Foreign file name (without directory or extension) |
| **type** | Foreign file extension (with leading '.') |
| **vers** | Foreign file version number (blank if not supported) |
| **fxs** | Foreign extension specifier (see §2.4 ) |
| **fxscl** | Clean version of **fxs** (all non-alphanumeric characters replaced by underscores) |
| **fmt** | Foreign format name (upper case) |
| **ndf** | Full name of the native NDF format copy of the dataset |

Note that the EMS library, which performs substitution of these tokens, imposes a limit of 200 characters on the resulting command. If long file names are in use this may present a problem unless the conversion command itself is short. Fortunately, this can always be arranged by wrapping it up in a simple script if necessary.

## 2.4 Accessing Sub-structures Within Foreign Data Files

The native HDS format allows multiple NDFs to be stored within a single disk file, and some foreign data formats provide somewhat similar facilities. As a concrete example, the FITS format allows images to be stored within *image extensions*, so a single FITS file may contain several images, each of which can be thought of as a foreign format NDF. When an NDF application is run, a specific NDF within such a FITS file can be selected by appending a *foreign extension specifier* (FXS) to the end of the file name. A foreign extension specifier consists of a string delimited by matching square brackets. The string identifies a sub-structure within the specified file, using syntax specific to the data format. So, for instance, the second image extension within a FITS file called `m51.fit` could be specified using the string "`m51.fit[2]`". Here, the sub-string "`[2]`" forms the foreign extension specifier, and uses the syntax expected by the CONVERT application FITS2NDF.

The foreign extension specifier is made available to external commands using a message token called **fxs**. Since this will certainly include square brackets (and possibly other non-alphanumeric characters), it cannot safely be included directly within the name of a file. You may want to do this for instance, when setting up the NDF_KEEP_ or NDF_TEMP_ environment variables. For this reason, a "cleaned" version of the foreign extension specifier is also available, in a message token called **fxscl**. This is equal to **fxs** except that all non-alphanumeric characters are replaced by underscores.

*Note, currently the NDF library only allows foreign extension specifiers to be given when accessing existing NDFs for read-only access. An error will be reported if an FXS is included in the name of an NDF to be created, or an existing NDF for which update or write access is required.*

## 2.5 Writing Format Conversion Utilities

In the previous section, the utilities "fitsin" and "fitsout" were presumed to exist to perform the necessary conversions. For commonly encountered formats, this is likely to be the case, and the CONVERT package (SUN/55) and other likely sources of conversion utilities should be investigated before embarking on writing your own. Don't forget that you can often adapt existing utilities (including those provided by the operating system) by combining them into a suitable script.

If you do need to write your own format conversion utilities from scratch, then the rules that apply are very few. It should obviously be possible to execute your utility by invoking a suitable command which includes the names of the input and output datasets. Your utility will also need to be able to interpret the NDF name it receives. This means that if you are writing a program, it should probably use the NDF library to access the NDF data (rather than, say, HDS, which cannot necessarily interpret the compound data structure names that will occur).[4] For a template example of a conversion utility that reads data from unformatted Fortran files, see SUN/33.

As far as possible, the NDF library will attempt to ensure that the output dataset to be written by a conversion command does not already exist, by deleting it first if necessary (your conversion utility should then create it). However, it may not always be wise to depend on this. In particular, recovery from error conditions (such as failed conversions) is likely to be more robust if conversion commands are able to cope when their output datasets already exist.

Unless you are debugging, you should also arrange for conversion utilities not to write to the standard output channel, as such output will otherwise appear whenever a conversion occurs. This is not normally wanted.

Beyond this, you have complete freedom to define and implement the conversion you want to perform. This may have whatever side effects you choose, so long as it results in the production of the requested output dataset, leaves its input dataset intact and returns an appropriate status value to the NDF library (see §3.6 for a discussion of error handling in conversion commands).

## 2.6 Defining Output Formats

As you might expect, you define the formats for output[5] datasets in rather the same way as for input datasets (§2.1), by means of a search list. However, the way this list is used is slightly different in this case.

The output format list is found by translating the environment variable NDF_FORMATS_OUT, which might typically have a definition such as:

```
      setenv NDF_FORMATS_OUT '.,FITS(.fit),FIGARO(.dst),IRAF(.imh)'
```

Ignoring, for the moment, the '.' at the start, this list defines the names of foreign data formats which are to be recognised when creating new datasets, and associates a file extension with each one. The syntax and restrictions are identical to the NDF_FORMATS_IN list (see §2.2).

[4]But also see §3.3 for ways of avoiding this restriction.

[5]As before, we really mean *new* datasets here (because you could write output to a pre-existing dataset, which is covered by the NDF_FORMATS_IN list), but thinking of them as "output" datasets is more convenient.

There is no requirement for the output formats to be the same as those used for input although, for obvious reasons, they will often be so. You could, however, give your formats different names or file extensions in the output list if you wanted.

The NDF library uses the same commands to perform format conversion for output datasets as for input ones (see §2.3), so the names of output formats should be chosen to select the environment variable containing the appropriate command. Note, however, that the "NDF_FROM_…" command will not be used in the case of output datasets.

## 2.7 Specifying an Output Format

With the output format list above, the following could be given to the NDF library when it is expecting the name of a new dataset:

```
newfile.fit
```

and it would recognise this as a request to write the new file in FITS format (performing the appropriate conversion when necessary).

If the name supplied were simply:

```
newfile
```

(*i.e.* if no file extension is specified), then the *first* format appearing in the output format list would be used. This is where the '.' in the earlier example (§2.6) comes in, as it stands for the native NDF format. Hence, a native format NDF would be written in this case. This is normally the required behaviour, so having '.' at the start of the format list is recommended.

However, if you wanted to work predominantly with a foreign format (say you were using NDF applications with another package which could not access NDF data directly), then you could put that format at the start of the output format list. For example:

```
setenv NDF_FORMATS_OUT 'IRAF(.imh),FITS(.fit)'
```

would cause all output files to be written in IRAF format and to have a file extension of '.imh' by default. You could still specify FITS format explicitly by giving a file extension of '.fit'.

## 2.8 Propagating Data Formats

The output format may also be determined according to the format of a related input dataset. This is achieved by putting the "wild-card" character '∗' at the start of the output format list, as follows:

```
setenv NDF_FORMATS_OUT '*,.,FIGARO(.dst),FITS(.fit),IRAF(.imh)
```

This affects new datasets which are created as a result of the *propagation* of information from an existing dataset (*e.g.* by applications calling the routines NDF_PROP or NDF_SCOPY, as described in SUN/33) and for which no explicit output file extension is given. In this instance, not only will data values be propagated, but so also will the dataset format. Thus, if a FIGARO format dataset (with file extension '.dst') had been accessed for input and propagated to create an output dataset, then a similar FIGARO format dataset would be created (also with a '.dst' file extension) unless an explicit output file extension were given.[6]

Note that output datasets which are created by applications without propagation from an existing dataset do not inherit any format information. In this case, any '∗' in the format list is ignored and the normal rules apply (in the example above, a native format NDF dataset would be created instead).

## 2.9 Resolving Naming Ambiguities

Unfortunately, because the '.' (dot) character is used both to separate a file extension from its file name and also to separate fields in an NDF (or HDS) object name, ambiguities can sometimes arise. For example, if the dataset name:

```
datafile.fit
```

is supplied, it might mean a foreign (FITS) data file with a '.fit' extension, or it might identify an NDF structure called `FIT` residing within an HDS file called `datafile.sdf`.

In such cases, the NDF library always uses the former interpretation. That is, it attempts to access (or create) a foreign format file whenever a file extension appears to be present and corresponds with a known foreign data format. For example, if '.xyz' is a recognised foreign file extension, then:

```
myfile.xyz
```

and

```
my.file.xyz
```

are both references to foreign format files rather than HDS objects (although they may not necessarily be valid file names on all operating systems). Conversely:

```
yourfile.abc
```

Would not identify a foreign file if '.abc' is not a recognised foreign file extension.

On UNIX, where the file system is case sensitive, it is possible to circumvent this behaviour by exploiting the case insensitivity of HDS component names. For instance, if '.img' (lower case) is a recognised foreign file extension, then the dataset name:

```
anyfile.IMG
```

with '.IMG' in upper case, refers to a native format NDF (an object called `IMG` contained within the HDS file `anyfile.sdf`). To leave this possibility open, it is recommended that foreign file extensions should always contain at least one lower case character.

[6]Note that in this case the *input* format description is being used to create the *output* dataset, so it would not strictly be necessary for the FIGARO format to appear in the NDF_FORMATS_OUT list.

### 2.10 Example: Setting Up a New Format

The following example shows the C shell commands that might be used on a UNIX system to give NDF-based applications access to a new data format. Typically, commands such as these would appear in a startup file, perhaps packaged as part of a "driver" that could be installed to give access to the data format in question:

```
#  Ensure that the new format and its file extension are recognised on
#  input.
      if ($?NDF_FORMATS_IN) then
         setenv NDF_FORMATS_IN $NDF_FORMATS_IN',NEW(.new)'
      else
         setenv NDF_FORMATS_IN 'NEW(.new)'
      endif

#  Similarly, ensure they are recognised on output.
      if ($?NDF_FORMATS_OUT) then
         setenv NDF_FORMATS_OUT $NDF_FORMATS_OUT',NEW(.new)'
      else
         setenv NDF_FORMATS_OUT '.,NEW(.new)'
      endif

#  Define commands to convert from the new format to NDF format and
#  vice versa.
      setenv NDF_FROM_NEW 'new2ndf in='\''^dir^name^type\'' out='\''^ndf\'
      setenv NDF_TO_NEW   'ndf2new in='\''^ndf\'' out='\''^dir^name^type\'
```

This example illustrates a couple of points which were not addressed earlier:

(1) We first check to see if the NDF_FORMATS_IN and NDF_FORMATS_OUT environment variables are already defined. If they are, we can append our new format description to them so as not to disturb any definitions already in use. Otherwise we must set them up from scratch.

(2) The environment variable definitions have been written so that single quote characters appear around the names of datasets. For example, the translation of the environment variable NDF_FROM_NEW would be:

```
new2ndf in='^dir^name^type' out='^ndf'
```

Although the syntax needed is a bit messy, this does mean that any special characters that appear in dataset names will be handled correctly (*i.e.* literally), and not expanded by the shell that interprets the command.

## 3 ADDITIONAL FACILITIES

### 3.1 Explicit Deletion Commands

When accessing files containing foreign format data, the NDF library will, on occasion, have to delete them (for instance, the routine NDF_DELET might have been called by an application).

Normally, this causes no problem, as a named file can easily be deleted when necessary. With some formats, however, this is not so simple. For example, data written in IRAF format will normally reside in two associated files – although the NDF library can delete the one it knows about, the other one would remain in existence.

To overcome this and other similar problems, it is possible to define an explicit deletion command for any foreign format which needs special treatment. If one is defined, it will over-ride any attempt by the NDF library to delete files which it knows are written in that format.

Taking the IRAF format as an example, the command would be defined via the environment variable NDF_DEL_IRAF in the same way as when defining format conversion commands. For example:

```
      setenv NDF_DEL_IRAF 'rm -f ^dir^name.imh ^dir^name.pix'
```

would unsure that both files associated with the dataset (with extensions '.imh' and '.pix') are deleted when necessary.

The deletion command is invoked in the normal way, by passing it to the C run time library "system" function, having first performed message token substitution on it (see §2.3). In this case, the NDF library defines the following tokens for use in the command:

| Token | Value |
|---|---|
| **dir** | Directory in which the foreign file resides |
| **name** | Foreign file name (without directory or extension) |
| **type** | Foreign file extension (with leading '.') |
| **vers** | Foreign file version number (blank if not supported) |
| **fxs** | Foreign extension specifier (see §2.4 ) |
| **fxscl** | Clean version of **fxs** (all non-alphanumeric characters replaced by underscores) |
| **fmt** | Foreign format name (upper case) |

## 3.2 Retaining Converted Data

Normally, when a foreign format dataset is converted to or from the native NDF format, the native copy of the data will be held in temporary file space (it will usually be written to a data structure held in the standard HDS scratch file), and this copy will be deleted when no longer required. Normally, this occurs when the dataset is released by the application.

Sometimes, however, it may be more convenient to retain the converted copy. For example, if you have foreign format data but plan to run several NDF-based applications on it, then retaining the native NDF copy the first time it is converted will save you having to re-convert the data on subsequent occasions.

To do this, the NDF library 'KEEP' tuning parameter should be set to 1. This can be done by calling the NDF_TUNE routine from within an application, but can also be done by setting the environment variable NDF_KEEP to '1' outside the application (see SUN/33), for example:

```
setenv NDF_KEEP 1
```

If this is done, then subsequent access to a foreign format dataset (say `galaxy.fit`) will create a corresponding native format NDF copy of the data in the default directory (in this case in a file called `galaxy.sdf`). This will be retained, and will then be accessed the next time `galaxy` is specified as a dataset name (remember, if no file extension is given, there is always an implicit search for a native format dataset before looking for a foreign one).

The 'KEEP' tuning parameter may be changed at any time, so control over individual datasets is possible if it is set from within an application. The value used will be that in effect when the dataset is first accessed.

## 3.3 Specifying Where the Native NDF is Stored

If the native NDF copy of a foreign dataset is not being kept, then the NDF library will, by default, store it within the HDS scratch file, as described earlier. This is generally most efficient. However, not all conversion utilities will necessarily be able to access such an NDF, particularly if they know nothing of the NDF or HDS data formats themselves. This would be the case with a general purpose data compression utility, for instance.

The NDF library therefore allows you to specify where the native NDF copy of the data should be stored. This is done by defining environment variables containing message tokens that evaluate to the name you want this NDF to have.

Up to two such environment variables may be defined for each foreign format. Their names are generated by prefixing 'NDF_KEEP_' and 'NDF_TEMP_' to the foreign format name (in upper case), and they correspond to the two cases (a) where the native copy of the NDF is being kept, and (b) where it is not. The two cases are handled separately, but the message tokens available are the same in both instances, as follows:

| **Token** | **Value** |
|---|---|
| **dir** | Directory in which the foreign file resides |
| **name** | Foreign file name (without directory or extension) |
| **type** | Foreign file extension (with leading '.') |
| **vers** | Foreign file version number (blank if not supported) |
| **fxs** | Foreign extension specifier (see §2.4 ) |
| **fxscl** | Clean version of **fxs** (all non-alphanumeric characters replaced by underscores) |
| **fmt** | Foreign format name (upper case) |

For example, if we were defining a COMPRESSED format and wanted to ensure that the native NDF data was always stored in its own file in the default directory, so that the UNIX "compress" utility could access it, we might use:

```
setenv NDF_KEEP_COMPRESSED ^name
setenv NDF_TEMP_COMPRESSED tmp_^name
```

Then, whatever the setting of the 'KEEP' tuning parameter, the name of an NDF in the default directory would be generated, so the HDS scratch file would never be used to hold the NDF copy of data from a COMPRESSED dataset. Note that specifying an explicit NDF name in this way does not affect whether the native NDF copy is deleted when the dataset is released. This is still determined by the 'KEEP' tuning parameter (see §3.2).

If, in the above example, the foreign dataset were called `/home/me/data/nebula.sdf.Z` (and the 'KEEP' tuning parameter was not set), then the NDF name would be `tmp_nebula` and this name would be passed (as the value of the '^ndf' message token) to any conversion commands that needed to be invoked. The NDF itself would reside in an HDS file called `tmp_nebula.sdf` (the '.sdf' extension being added automatically by HDS).

Note that the value given for the NDF_KEEP_COMPRESSED environment variable above is, in fact, the same as its default. You should generally be wary of setting this to anything except its default value because the user of an application might well be confused if he sets the NDF_KEEP environment variable to specify that the NDF should be kept, but it ends up with an unexpected name. This facility does, however, give control over which directory is used to store the file.

## 3.4 Efficiency Considerations

When deciding where to store the native NDF format copy of a dataset, it is wise to specify a location on a local file system wherever possible. This is, of course, always good practice where large datasets are concerned, as access to remote files is usually far less efficient and can generate considerable network traffic that may interfere with other people's work.

With NDF format conversion facilities, local file access is even more important. This is because the temporary datasets involved are always read immediately after being written, and very frequently deleted immediately after that. In this situation, an operating system with good file caching will often not actually write the data to a local file at all, but merely copy it to and from memory. This is far faster than waiting for actual data transfer to take place, which is what will normally happen if remote files are involved.

For this reason, you are recommended to configure format conversion software so that temporary datasets are stored in the user's default directory, in the expectation that this directory, at least, will be chosen sensibly and reside on a local file system. Users may, however, still need to be reminded of the need for this (*e.g.* in documentation). You may also need to explain how to change this behaviour if, for example, access to larger amounts of space for temporary files becomes necessary.

Note that, by default, temporary NDF datasets are stored in the standard HDS scratch file, which resides in a directory specified by the HDS_SCRATCH environment variable. If this variable is not explicitly set, the user's default directory is used.

## 3.5 Example: Data Compression

To illustrate the above, the following is a complete example of the C shell commands that might be used to allow access to compressed NDF data files (with file extension '.sdf.Z') on UNIX systems:

```
#  Define the COMPRESSED format to be recognised on input, with file
#  extension '.sdf.Z'.
```

```
      if ($?NDF_FORMATS_IN) then
         setenv NDF_FORMATS_IN $NDF_FORMATS_IN',COMPRESSED(.sdf.Z)'
      else
         setenv NDF_FORMATS_IN 'COMPRESSED(.sdf.Z)'
      endif

#  Similarly, recognise it on output.
      if ($?NDF_FORMATS_OUT) then
         setenv NDF_FORMATS_OUT $NDF_FORMATS_OUT',COMPRESSED(.sdf.Z)'
      else
         setenv NDF_FORMATS_OUT '.,COMPRESSED(.sdf.Z)'
      endif

#  Store the uncompressed data in the default directory.
      setenv NDF_KEEP_COMPRESSED ^name
      setenv NDF_TEMP_COMPRESSED tmp_^name

#  Use the "uncompress" and "compress" utilities to convert the data.
      setenv NDF_FROM_COMPRESSED 'uncompress -c -f ^dir^name^type >^ndf.sdf'
      setenv NDF_TO_COMPRESSED 'compress -c -f ^ndf.sdf >^dir^name^type;:'

#  Suppress processing of extension information for compressed data.
      setenv NDF_XTN_COMPRESSED ''
```

Note that the "compress" command has been followed by a null ":" command which does nothing. This is because an error status may be returned if the file being compressed does not get any smaller, so the ":" command ensures that the invoking NDF application always receives a success status. In a production system, more secure error handling than this would probably be required.

For an explanation of the final definition of the NDF_XTN_COMPRESSED environment variable, you should refer ahead to §4.4.

(**Warning:** *Users should be warned that it is unwise to archive compressed data unless thay are sure that the necessary decompression software will be available to them in future, possibly on different hardware and/or operating system platforms.*)

## 3.6 Handling Errors in Conversion Commands

When a command associated with access to foreign data completes, the NDF library checks to determine if it was successful.

It first looks at the status value returned by the C "system" call which invoked the command. These status values are operating-system dependent but, on most systems, there is provision for the command to return either a "success" or an "error" status to the command interpreter and for the invoking application to receive this. If the NDF library does not receive a success status back, it deduces that the command has failed and generates an appropriate error report. The NDF_ routine that was invoked then returns to the application with its STATUS argument set and the application would probably then abort and display the error message.

For commands that invoke conversion utilities (associated with either of the environment variables NDF_FROM_... or NDF_TO_...), the NDF library will also check to see that the output dataset from the conversion operation has been created. Where this dataset is a native format NDF, it will be opened to check that it contains a valid NDF data structure. Any problem will again result in an error report from the invoking application.

When writing data access commands or conversion utilities, the recommended course of action if an error occurs is for diagnostic error information to be written to the standard error channel, and for the invoked command to return with an "error" status value appropriate to the command interpreter in use.

It is generally wise to avoid having the conversion utility re-prompt for new input, as this can be confusing for the user who may not be aware that conversion is taking place. This can normally be arranged by appropriately redirecting the standard input and/or output channels to a null device so that the command will abort and control will return to the NDF library if an attempt is made to prompt for (or read) new input.

## 3.7 Avoiding Unwanted Recursion

When writing format conversion utilities, it is often convenient to use the NDF library to access the native NDF format version of the data (see §2.5). However, you should bear in mind that the NDF library's ability to invoke format conversion commands will still be active unless you take action to switch it off. This means that unwanted recursion is possible if a conversion utility accesses a foreign dataset that in turn causes a further conversion utility to be invoked, and so on…

In practice, this is unlikely to be a problem if care is taken to ensure that NDF datasets are never stored in objects whose names might be mistaken for foreign format data files. If it does prove necessary to suppress unwanted format conversion, however, this can be achieved by setting the NDF_ library's DOCVT tuning parameter to zero. This will have the effect of disabling recognition of foreign data files by the conversion utility.

One way of doing this is by setting the environment variable NDF_DOCVT to 0 as part of the format conversion command, immediately before the conversion utility itself is invoked. Alternatively, the conversion utility may call the NDF_TUNE routine itself in order to control recognition of foreign data formats. The latter approach allows individual control over each dataset accessed by the utility if necessary.

## 3.8 Debugging Conversion Commands

Normally, all foreign data access commands invoked by the NDF library execute silently, unless an error occurs or a command writes information to standard output (this should normally be avoided). To assist in debugging, however, the NDF library provides a tuning parameter 'SHCVT'. This can be used to make it display all commands before they are executed but after message token substitution has taken place.

To enable this feature, the 'SHCVT' tuning parameter should be set to 1. This can be done from within an application by calling the NDF_TUNE routine (see SUN/33), or from outside the application by setting the NDF_SHCVT environment variable, as follows:

```
setenv NDF_SHCVT 1
```

# 4 DATA EXTENSIBILITY

## 4.1 General Principles

A notable feature of the NDF data format is its extensibility, which is achieved by means of independent *extensions*[7] to the format, which can be defined and added to suit the needs of individual software authors. A key distinction between these extensions and the other contents of an NDF dataset is that the meaning and processing rules for data held in extensions are generally unknown to writers of format conversion utilities, whereas the standard components of an NDF have well-defined and universal meanings (see SUN/33).

This has important implications. It means, for instance, that it is relatively straightforward to write a general purpose utility to change (say) IRAF format into NDF format, so long as only standard NDF components need to be considered. However, if the receiving NDF application is equipped to handle data in its own NDF extension, then converting that additional data (*i.e.* extracting it from the IRAF file and putting it into the NDF extension) will require specialist knowledge, and so cannot be expected of a general purpose utility.

What is required is for conversion utilities to be extensible in the same way as the NDF datasets themselves. A standard utility could then be used to convert the bulk of the data, and a more specialised utility could simply add the extension information to the converted dataset.

As will be explained below, the NDF library supports this concept, but there still remains one problem. If, for example, you had written a software package and an associated utility that extracted specialist extension information from IRAF datasets, you would probably not want to repeat this work for every other possible data format that might come along in future – you would surely prefer to use the same specialist utility to access a whole range of foreign formats. This is where the NDF's *FITS extension* comes in.

## 4.2 The FITS Extension

An important feature of the well-known FITS [8] data format (which was originally designed as a convenient container for the interchange of astronomical images between sites) is its "FITS header". This, in essence, is a sequence of character strings each of which contains the name of a keyword, an associated value and (optionally) a comment.

Although rather few of the keywords that appear in a FITS header have standardised meanings, the freedom that this gives makes it a convenient place to store information about which the reader or writer may have little knowledge. A special NDF extension mirroring the properties of a FITS header can therefore provide a useful "airlock" or "staging post" for interchanging specialist information between general purpose conversion utilities (for which the information is meaningless) and specialist utilities (for which it has meaning).

To satisfy this requirement a FITS extension, equipped to hold FITS header information, may be added to an NDF. By convention, it consists of a 1-dimensional (HDS) array of _CHAR∗80 character strings which holds a sequence of header records according to FITS formatting rules (including the final 'END' record).

[7] Be careful to distinguish an "extension" to an NDF data structure (which is an addition of extra data to the file) from the "file extension" (which is the end part of the file name, such as '.sdf', used to identify the file's format).

[8] FITS stands for Flexible Image Transport System.

## 4.3 Extension Import and Export Operations

To illustrate the function that the FITS extension performs, consider the following sequence of events in which an IRAF format file is read by an application that expects to find a CCDPACK extension present:

(1) Having detected that it needs to convert the data format, the NDF library first invokes a general purpose conversion command, as defined in the NDF_FROM_IRAF environment variable (see §2.3). This, in turn, invokes a conversion utility which creates the NDF data structure and fills in all the relevant standard components using information obtained from the IRAF dataset.

(2) The same utility then assembles all ancillary information that it doesn't recognise (essentially the contents of the IRAF header file) and writes it to a FITS extension, which it creates in the new NDF. It then terminates.

(3) The NDF library then invokes a specialist utility, written by the designer of the CCDPACK extension. This inspects the FITS header (and other standard components of the NDF if necessary) and transfers the information it recognises into the CCDPACK extension, which it creates. It too, then terminates.

(4) Other specialist utilities may then be invoked, if required, to create further extensions using information in the FITS header.

(5) Finally, the original application regains control and accesses the NDF dataset that has been built.

When writing to a foreign dataset, the sequence of events is broadly similar, except that the specialist utilities are invoked first (before the general purpose one) and transfer information from their relevant extensions **into** the FITS extension. The general purpose conversion utility then transfers the contents of the FITS extension to the foreign dataset as part of its conversion task.

The processes of (a) creating a specialist extension from information stored in the FITS extension and (b) writing specialist extension information back into the FITS extension are referred to as *importing* and *exporting* the extension information.

Using this scheme, utilities that import and export extension information will, in many circumstances, be able to rely entirely on the contents of the FITS extension and need not access the foreign data file at all. This relieves their authors of the need to understand the foreign format, beyond knowing what FITS keywords will be used to store the information of interest. Import and export utilities are therefore easily re-used when new formats are encountered. Indeed, since FITS keywords are so widely used, there will often be conventions in place that make even a change of keywords unnecessary when adding a new format.

The following sections now describe the stages involved in setting up import and export utilities to make use of this scheme.

### 4.4 Defining the Extension List

It is first necessary to define the set of specialist NDF extensions that should be recognised. This is normally done via the environment variable NDF_XTN, as follows:

```
setenv NDF_XTN CCDPACK,IRAS90
```

This is simply a comma-separated list of extension names conforming to the naming conventions described in SUN/33. It applies to all foreign format datasets, unless overridden. The order in which extensions occur in this list determines the order in which they will be imported. They will be exported in the reverse order.

On occasion, it may be necessary to use a different list of NDF extensions for a particular foreign format. Most commonly, this involves simply using an empty list for formats that do not require any extension handling (data compression of ordinary NDF data files would be an example – see §3.5). To specify a separate extension list for a particular foreign format, an environment variable is used whose name is constructed by prefixing 'NDF_XTN_' to the format name (in upper case). For example:

```
setenv NDF_XTN_COMPRESSED ''
```

would over-ride the normal extension list with an empty one so that no extension handling would occur when COMPRESSED format data is accessed.

### 4.5 Extension Import and Export Commands

The commands that perform import and export of extension data are defined in the usual way via environment variables whose names are formed by prefixing 'NDF_IMP_' or 'NDF_EXP_' to the extension name (in upper case), for example:

```
setenv NDF_IMP_CCDPACK 'impccd ndf=^ndf'
setenv NDF_EXP_CCDPACK 'expccd ndf=^ndf'
```

Here, the extension name is CCDPACK (and should appear in the NDF_XTN list) and the "impccd" and "expccd" utilities are assumed to have been written to import and export information for this extension.

The commands are invoked after message token substitution has taken place, as described in §2.3. In this case, the set of tokens defined for use is as follows:

| Token | Value |
|---|---|
| **dir** | Directory in which the foreign file resides |
| **name** | Foreign file name (without directory or extension) |
| **type** | Foreign file extension (with leading '.') |
| **vers** | Foreign file version number (blank if not supported) |
| **fxs** | Foreign extension specifier (see §2.4 ) |
| **fxscl** | Clean version of **fxs** (all non-alphanumeric characters replaced by underscores) |
| **fmt** | Foreign format name (upper case) |
| **ndf** | Full name of the native NDF format copy of the dataset |
| **xtn** | Name of the NDF extension (upper case) |

As explained earlier, the foreign format file should not normally be accessed by import and export utilities unless that is unavoidable, so one set of import and export commands will normally suffice for accessing a whole range of foreign formats.

In special cases, however, where techniques specific to a particular format are needed, an alternative set of commands may be defined to apply to that format alone. This is done via environment variables whose names are constructed by appending an underscore and the foreign format name to the usual names shown above. For instance, when importing and exporting CCDPACK extension information to FIGARO files, one might want to use:

```
setenv NDF_IMP_CCDPACK_FIGARO 'impccd ndf=^ndf file=^dir^name^type'
setenv NDF_EXP_CCDPACK_FIGARO 'expccd ndf=^ndf file=^dir^name^type'
```

If these variables were defined, they would over-ride any defined without the '_FIGARO' suffix when accessing that particular format. Any special techniques can therefore be restricted to those formats that require them.

## 4.6 Writing Import and Export Utilities

Before writing your own import and export utilities, you should consider using standard ones that already exist. For example, the KAPPA package (SUN/95) contains a general purpose "fitsimp" command that can be used to build a specialist NDF extension by importing information from a FITS extension. It is driven by a keyword translation table stored in a text file, so can easily be adapted for different needs. For example, it might be used in an NDF import command as follows:

```
setenv NDF_IMP_MINE 'fitsimp ndf=^ndf xname=MINE table=$HOME/mine.imp'
```

Here, `mine.imp` is the table that drives the importation process. This could be different for each format if necessary. An equivalent extension export utility "fitsexp" is also available.

If you find that you must write your own software for this purpose, then the IMG library (SUN/160) provides a convenient programming interface for accessing items of NDF extension information (including individual items within the FITS extension) and should make most import and export utilities straightforward to write. With a little more effort, you can, of course, also use the NDF and HDS libraries, which allow you to construct any form of extension you want.

## 4.7 Example: Setting Up an Extension

The following example shows typical C shell commands that might be used to allow NDF applications to handle specialist extension information derived from foreign format datasets. Normally such commands would form part of the startup sequence for the package that utilised the extension.

```
#  Append the CCDPACK extension to the list of extensions to be
#  handled.
      if ($?NDF_XTN) then
         setenv NDF_XTN $NDF_XTN,CCDPACK
      else
         setenv NDF_XTN CCDPACK
      endif

#  Define commands for importing and exporting CCDPACK extension
#  information.
      setenv NDF_IMP_CCDPACK 'ccdimp ndf=^ndf table=$CCDPACK_DIR/^fmt.imp'
      setenv NDF_EXP_CCDPACK 'ccdexp ndf=^ndf table=$CCDPACK_DIR/^fmt.exp'
```

Note that we have specified keyword translation tables here (for use in the import and export commands) which depend on the foreign data format being accessed. This would be necessary if, for instance, data in different formats were to follow different conventions about how its header information is stored, so that different FITS keywords were used in the FITS extension as a result. By concentrating this information in a table, it becomes easy to change and users can even have their own versions if necessary.

Applications which process the converted data need only deal with the validated information stored within their own private extension. They are therefore insulated from details of the conversion process and any need to change in order to access new data formats in future. The use of a private extension also protects them from the possibility of other software inadvertently corrupting their private data.

# A ENVIRONMENT VARIABLES AND TOKENS

## A.1 Summary of Environment Variables Used

The following list summarises the environment variables used by the NDF library to control access to foreign format data. The symbols <FMT> and <XTN> are used to represent the names of foreign data formats and NDF extensions respectively. All environment variable names should be entirely in upper case.

**NDF_FORMATS_IN**
Comma-separated list of format names and associated file extensions, used when accessing pre-existing datasets. (Mandatory if pre-existing foreign datasets are to be accessed.)

**NDF_FORMATS_OUT**
Comma-separated list of format names and associated file extensions, used when accessing new datasets. (Mandatory if new foreign datasets are to be accessed other than by propagation of format information from a pre-existing dataset.)

**NDF_FROM_<FMT>**
Command to convert a foreign format dataset into an NDF. (Mandatory if datasets in that format are to be read.)

**NDF_TO_<FMT>**
Command to convert an NDF into a foreign format dataset. (Mandatory if datasets in that format are to be written or modified.)

**NDF_DEL_<FMT>**
Command to delete a foreign format dataset. (Optional.)

**NDF_KEEP_<FMT>**
Name of the NDF to be used to hold a native format copy of a foreign dataset in cases where the NDF is to be retained for future use. (Optional.)

**NDF_TEMP_<FMT>**
Name of the NDF to be used to hold a native format copy of a foreign dataset in cases where the NDF is temporary and will be deleted when the dataset is released. (Optional.)

**NDF_XTN**
Comma-separated list of NDF extensions to be recognised when accessing foreign datasets. (Optionally required if extension information is to be handled.)

**NDF_XTN_<XTN>**
Comma-separated list of NDF extensions to be recognised when accessing datasets in a particular foreign format. (Optionally overrides the value of NDF_XTN for that format.)

**NDF_IMP_<XTN>**
Command to import information into an NDF extension. (Optionally required to allow reading of extension information from foreign datasets.)

**NDF_IMP_<XTN>_<FMT>**
Command to import information into an NDF extension when accessing datasets in a particular foreign format. (Optionally overrides the value of NDF_IMP_<XTN> for that format.)

**NDF_EXP_<XTN>**
Command to export information from an NDF extension. (Optionally required to allow writing or modification of extension information in foreign datasets.)

**NDF_EXP_<XTN>_<FMT>**
Command to export information from an NDF extension when accessing datasets in a particular foreign format. (Optionally overrides the value of NDF_EXP_<XTN> for that format.)

## A.2 Summary of Message Tokens Used

The following table summarises all the message tokens used by the NDF library in format conversion commands, *etc:*

| **Token** | **Value** |
|---|---|
| **dir** | Directory in which the foreign file resides |
| **name** | Foreign file name (without directory or extension) |
| **namecl** | Cleaned version of **name** (all non-alphanumeric characters replaced by underscores) |
| **type** | Foreign file extension (with leading '.') |
| **vers** | Foreign file version number (blank if not supported) |
| **fxs** | Foreign extension specifier (see §2.4 ) |
| **fxscl** | Clean version of **fxs** (all non-alphanumeric characters replaced by underscores) |
| **fmt** | Foreign format name (upper case) |
| **ndf** | Full name of the native NDF format copy of the dataset |
| **xtn** | Name of the NDF extension (upper case) |

Each environment variable which is used to control access to foreign format data, and which permits substitution of the above message tokens, is of the form:

NDF_<CTX>_...

where <CTX> is the name of a "context" within which message token substitution takes place. The following table shows which tokens are defined for use within each context:

| | | *Token* | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | **dir** | **name** | **namecl** | **type** | **vers** | **fxs** | **fxscl** | **fmt** | **ndf** | **xtn** |
| *Context* | **FROM** | √ | √ | √ | √ | √ | √ | √ | √ | √ | × |
| | **TO** | √ | √ | √ | √ | √ | √ | √ | √ | √ | × |
| | **DEL** | √ | √ | √ | √ | √ | √ | √ | √ | × | × |
| | **KEEP** | √ | √ | √ | √ | √ | √ | √ | √ | × | × |
| | **TEMP** | √ | √ | √ | √ | √ | √ | √ | √ | × | × |
| | **IMP** | √ | √ | √ | √ | √ | √ | √ | √ | √ | √ |
| | **EXP** | √ | √ | √ | √ | √ | √ | √ | √ | √ | √ |