SC/12.1

Starlink Project
Starlink Cookbook 12.1

Jon Lockley

28 November 1997

Writing your own Data Reduction Software

Abstract

This Cookbook shows you how to write your own software for astronomical data reduction. There are two parts. The first part introduces you to simple data reduction using Starlink's IMG subroutine library. This hides a lot of the underlying complications. The second part introduces you to the more powerful (but more complex) NDF subroutine library for handling data.

Contents

1 Introduction

1.1 What is this book about?

Put simply, this book is about helping you to write your own data reduction software. This is achieved by showing you how to access and manipulate astronomical data files.

1.2 Why write my own software?

This book is partly for people who haven't got the tool to do the job. For example, existing software may not work on a particularly non-standard data set. They are in the position of having to write their own data reduction software to get around the problem. This isn't always an easy task, especially for the inexperienced programmer.

Alternatively, this book can be used by people who have an idea for a new data reduction algorithm. They might even want to pass it on to a larger audience and want to be sure that it will work on a variety of machines with a wide variety of data formats.

Before you go any further:

Make absolutely sure you *need*, to write new software.

Use the findme command (SUN/188) to see if the job has already been done. Ask your site manager. Ask a Starlink programmer (you'll find their email addresses on the back of the Starlink Bulletin or on the Starlink home page). Above all, don't waste *your* valuable time on writing software that already exists.

1.3 How this book works

This cookbook comes in two parts.

The first part leads you through a series of examples of very simple programs which handle data files. It tries to expose you to the absolute minimum of "inside trickery", Much non-essential information is sacrificed to help you get basic jobs done quickly.

The second part of this book delves a little deeper. It takes an approach which involves writing more complex code than that seen in part one, but it's code that can do more stuff. We'll also start looking at how to write software *packages* i.e. a toolbox to keep your tools in.

This book is *not* for people who want to know the in-depth workings of data mapping, format conversion and the like (although you will certainly find the references to the appropriate documents here). It is therefore not recommended that you use this guide as a manual.

1.4 What will I be able to do after reading this?

Hopefully, this book will show you how to do stuff you didn't know how to do before. For example, after part one you'll be able to write simple software which can:

- Read, manipulate, and create your own NDF (e.g. .sdf) format files.
- Gain access to other data formats.
- Read FITS header information.

After reading part two of this book, you'll be able to:

- Access quality and variance information.
- Access and create extensions.
- Make your own data reduction packages (monoliths).

Part I

SIMPLE ACCESS TO DATA

This first section of the cookbook follows a tutorial style. Its aim is to get you creating very simple applications. Because of its tutorial style, more confident programmers might find it more useful to skip the explanatory text and go straight to the sample source codes.

However, I recommend that even the experienced programmer should read the sections on interface files and the alink command.

2 Getting started

2.1 Introduction

The programs used in this book, and those you will write yourself in the same style, make use of the *ADAM software environment*. In a nutshell, this is a mechanism for bringing lots of different facilities and giving the result a common look and feel.

ADAM does a lot more than make things look pretty, but you'll be glad to know that there is no need to understand ADAM inside out to use this book. In fact, we'll hardly discuss ADAM at all. Suffice it to say that ADAM is there, providing the infrastructure you need to manipulate data files.

A program using the ADAM environment is usually referred to as an ADAM *task*. In practice, there are two types of ADAM task. *I-tasks* are used to perform jobs such as instrumentation control. They will not be referred to again in this book. *A-tasks* are concerned with data reduction and analysis. It is these we will be concentrating on.

A-tasks are often also called *applications*. This is the convention we will use for the rest of this book. After all we are in the business of writing software which does *stuff*.

It is possible to group a collection of applications together in a single package known as a *monolith*. Starlink packages are often made in this way (e.g. Figaro). This helps to keep related applications together. In the above example, one could package both the log and alog applications together into a simple package called logarithms. Both log and alog would be used separately in the same way that each application in Figaro (or KAPPA etc.) is run separately. We will return to monoliths in part two of this book.

2.2 Interface files

Most of the applications you write will require information to be given them by you or another user. Each piece of information required by the code is called a *parameter*.

Every ADAM application requires an *interface file* to provide information about the parameters required to perform the job requested. This file also specifies the prompts the user receives when asked to input parameter values, and the type of value required.

The name of the interface file for an application ends with the extension .ifl. In other words, an application called prog has a source code file prog.f and a corresponding interface file prog.ifl.

2.3 Compiling your applications – The alink command

ADAM A-tasks are easily compiled with the facility alink. A typical use of alink might be:

```
% alink prog.f -L/star/lib ‘img_link_adam‘
```

This command compiles and links the application source code. It also makes use of a Starlink library (in this case the IMG library). It's not too important that you understand how all this works just now.

2.4 Section summary

We've covered quite a bit of terminology in this section, so rather than allow confusion to set in, let's review what's essential to know in what we've seen so far:

- You will be writing applications.
- Applications can be individual or part of larger packages called monoliths.
- All applications require a source code file and an interface file.
- Applications are compiled (and linked) using alink.

3 The basics of the IMG library

Note – The more experienced programmer might find the first couple of examples in this section worth skipping. In this section, you'll begin to write your own applications. We'll start very slowly by looking at a traditional "do nothing" code.

3.1 Example 1 – A "do nothing" code

Let's get started on your first application. Enter the following code via your choice of text editor. Save it as nowt.f.

```
      SUBROUTINE NOWT(STATUS)
C This application does nothing. It is an inapplicable application.
      CONTINUE
      END
```

You'll need an interface file as well, despite the lack of code, as it is required by alink. Again, enter this with your text editor, but this time save it as nowt.ifl

```
interface nowt
endinterface
```

Now we can compile and link the code in the following way. From the UNIX prompt type:

```
% alink nowt.f
```

This should create an executable file called nowt. Try running nowt. If you've done everything correctly, nowt happens. While this is a good sign, it's hardly gripping stuff. You could always edit the code so it reads:

```
      SUBROUTINE NOWT(STATUS)
C This application does nothing. But at least you know it's friendly.
      WRITE (*,*) 'Hello World!'
      CONTINUE
      END
```

Note that you can run alink to recompile nowt.f without changing the interface file. This is because the nowt application still doesn't require any parameters to work.

3.2 Example 2 – Opening and closing files

Let's now progress to a code which opens a Starlink (.sdf) data file. Enter the following code and save it as sesame.f

```
       SUBROUTINE SESAME(STATUS)
C This application opens a file and promptly closes it again.
       INTEGER NX, NY, IP, STATUS
       CALL IMG_IN('IN',NX,NY,IP,STATUS)
       CALL IMG_FREE('IN',STATUS)
       END
```

And of course we'll need the obligatory interface file sesame.ifl

```
interface SESAME
  parameter IN
    prompt 'Input Image'
  endparameter
endinterface
```

Now, to compile sesame use:

```
% alink sesame.f -L/star/lib `img_link_adam`
```

If you get error messages, first make sure you have started the Starlink software on your session. If you still have trouble, try the command star_dev. (This sets up a link to the Starlink libraries – don't worry too much about how it works). Note the change in the alink command from the previous ones.

Now try running your code. You should get:

```
% sesame
IN - Input image >
```

Assuming your code works(!), you can give the name of an NDF file which it will read and promptly do nothing. This code appears to be another "do nothing" code. It isn't, so let's take a closer look at what's going on.

Firstly, we start the application with the usual

```
SUBROUTINE SESAME(STATUS)
```

It is a convention when writing ADAM applications to always have a subroutine (of the same name as the application) with an integer argument instead of the more familiar

```
PROGRAM SESAME
```

statement. Don't worry too much about why this should be the case. The next line merely declares some variables. Now we get to:

```
      CALL IMG_IN('IN',NX,NY,IP,STATUS)
```

This line opens the file you told it to. It associates the file with a *parameter* called IN. In order to get information about the parameter IN, the application looked in the interface file. It was told that for this parameter it would need to prompt the user into providing a file name by using a message.

IMG_IN also found out some information about the image. Its size is returned via the arguments NX and NY. The next variable (IP) is a *pointer*. We will return to this concept in the next section. STATUS is used to check all is well.

The next call,

```
      CALL IMG_FREE('IN',STATUS)
```

simply closes the image associated with the parameter IN and tidies up. Applications which call more than one image must apply this call to all the images used.

3.3 About the IMG library

In the last section, we used two subroutine calls of the form CALL IMG_ These subroutines are part of the *IMG library*. You can find out all about the facilities offered by IMG by reading SUN/160.

IMG is a collection of software designed to make the task of reading astronomical data easy. It allows access to both the data itself and the header information contained within the files. It can be called from both Fortran and C codes (we will stick to the former) and forms part of the NDF library. The NDF library is somewhat more complex (and somewhat more powerful) than the IMG library, so let's save any more discussion on it until later sections.

When the alink command was used in the last section,

```
      % alink sesame.f -L/star/lib `img_link_adam`
```

you explicitly told alink that you needed to link sesame.f to the IMG library. **You must make absolutely sure that every time you use the IMG library, you inform alink in this way!** This also goes for any other Starlink libraries you use (as we will do later on in this book).

3.4 Really getting to your data (or data mapping)

Opening and closing files is fine, but what you really want to be able to do is get to your data. ADAM libraries have a particular way of handling arrays of data. Remember how in sesame.f you used:

```
      CALL IMG_IN('IN',NX,NY,IP,STATUS)
```

but we didn't discuss the integer variable IP? Well, this is related to data access. Specifically, IP acts as a *pointer* to the array of data in the file. Users of the C programming language will be familiar with the concept of pointers, but Fortran programmers will generally have little or no experience of them. No problem, you don't need to know the ins and outs of how they work. Put simply, a pointer tells a piece of code where some information is kept in the memory of the machine. In the case of the applications we're going to be looking at in a minute, we'll use pointers to "map" how the data contained in a file is held in the memory of your machine.

3.5 Putting it all together – Complete applications

In this section, we'll go through two applications. The first demonstrates how data in a file is mapped into an array of values. The second covers the methods of creating new files and updating old files.

3.6 Example 3 – Processing data values

Type in the following code and save it as stats.f

```
      SUBROUTINE STATS(STATUS)
C This program works out the mean, the minimum and the maximum
C values in the data array of a file.
      INTEGER NX, NY, IP, STATUS, I, J
      REAL MIN, MAX, MEAN, SUM
C
      CALL IMG_IN('IN',NX,NY,IP,STATUS)
      CALL DOSTAT(%VAL(IP),NX,NY,STATUS)
      CALL IMG_FREE('IN',STATUS)
      END

      SUBROUTINE DOSTAT(IMAGE,NX,NY,STATUS)
C Make sure SAE_PAR is available
      INCLUDE 'SAE_PAR'
      REAL IMAGE(NX,NY), SUM, MIN, MAX
C
      IF (STATUS .NE. SAI__OK) RETURN
C Initialise the variables we'll need.
      SUM = 0.0
      MAX = IMAGE(1,1)
      MIN = IMAGE(1,1)
C Now loop around all the points in the data
      DO I = 1, NX
        DO J = 1, NY
          SUM = SUM + IMAGE (I,J)
          IF (IMAGE(I,J) .GT. MAX) MAX = IMAGE(I,J)
          IF (IMAGE(I,J) .LT. MIN) MIN = IMAGE(I,J)
        END DO
      END DO
C Print the results
      WRITE (*,*) 'The size of the image is ',NX,' by ',NY,' pixels'
      WRITE (*,*) ' '
      WRITE (*,*) 'The mean of the image is ', SUM/REAL(NX*NY)
      WRITE (*,*) ''
      WRITE (*,*) 'The minimum and maximum are ',MIN,' and ',MAX
      END
```

Have a go at writing your own interface file for stats and compile it with alink. Finally, try running it on an NDF (i.e. .sdf) file of your choice.

Let's have a careful look at how this code is constructed. The main body of the code consists of opening an image, calling a subroutine which does the actual work, followed by a call to close the image and clean up. You'll find this structure in a lot of Starlink software. It helps to keep the code uncluttered and is easy to maintain, so we recommend that you try to stick to it.

The call to the DOSTAT subroutine shows how pointers are used to access the data in the file. The %VAL statement is a VAX extension to Fortran. Normally, when a Fortran code uses arrays, it needs to have some information about the size of the array right from the outset. There is no method of doing memory allocation "on the fly". (This is not true of Fortran 90, but let's restrict ourselves to using Fortran 77 for this guide). This just isn't practical for applications which have to deal with many different sizes of images.

The %VAL method makes the pointer (in this case IP) look just like an array. What's more, it makes sure that the array is of the right size, in this case NX by NY pixels. This is not a feature of "standard" Fortran 77, but unless you hear otherwise, it should be OK on your machine.

You'll see that when the subroutine DOSTAT inherits the %VAL, it does so with a ready-made array of real values. This way, there is no need to pre-define your array sizes.

3.7 Output from applications

When you want to write out a data file you want to do one of two things. Either you want to write out the same data file you read in, presumably with some modification, or you want to create an entirely new file. The IMG library quite happily lets you do both. The next two codes describe each of these processes in turn.

3.8 Example 4 – Updating a file

Enter the code for clip.f:

```
      SUBROUTINE CLIP(STATUS)
C This application sets all the data values above and
C below two thresholds to zero.
      CALL IMG_IN(IMAGE,NX,NY,IP,ISTAT)
      WRITE (*,*) 'Enter the maximum data value permitted >'
      READ (*,*) MAX
      WRITE (*,*) 'Enter the minimum data value permitted >'
      READ (*,*) MIN
C
C Make sure we haven't done anything really dumb
C
      IF (MAX .LE. MIN) THEN
C
C Oh dear...
C
        WRITE (*,*) 'Doh!!!!'
        RETURN
      END IF
C
C Open the image, but do so in such a way that it can be
C modified rather than kept the same.
C
      CALL IMG_MOD('IN',NX,NY,IP,STATUS)
C
```

```
C Now clip off the values which are higher and lower than
C the requested values.
C
       CALL CLIPIT(%VAL(IP),NX,NY,MAX,MIN,NCLIP,STATUS)
C
C Let the user know how many pixels were reset.
C
       WRITE (*,*) NCLIP,' values were reset.'
C
C Tidy up.
C
       CALL IMG_FREE('IN',STATUS)
       END

       SUBROUTINE CLIPIT(IMAGE,NX,NY,MIN,MAX,NCLIP,STATUS)
C
       REAL IMAGE (NX,NY)
       REAL MIN, MAX
       INTEGER NX, NY, STATUS, NCLIP
C
C Make sure SAE_PAR is available
C
       INCLUDE 'SAE_PAR'
C
C Is everything OK?
C
       IF (STATUS .NE. SAI__OK) RETURN
C
C Get clipping!
C
       NCLIP = 0
       DO I = 1, NX
         DO J = 1, NY
           IF (IMAGE(I,J).LT.MIN .OR. IMAGE(I,J).GT.MAX) THEN
           IMAGE (I,J) = 0.0
           NCLIP = NCLIP + 1
         END DO
       RETURN
       END
```

and the corresponding interface file clip.ifl:

```
interface CLIP
  parameter IN
    prompt 'Image to clip'
  endparameter
endinterface
```

and compile it using alink, remembering to link to the IMG library.

You can check that your code has worked by using the stats application from the last section. You should see how the reported maximum and minimum data values have changed if you chose your clipping limits appropriately.

This code *modifies* your input file, i.e. no new file is created. This is often not the most useful form of output as, especially in data reduction, you might want to repeat a step with the original data many times. In such a case, it is more useful to preserve the input data and create a new file from scratch. This next code demonstrates how this is done using the IMG library.

3.9 Example 5 – Creating a new file

Enter the following code for bigger.f.

```
      SUBROUTINE BIGGER(STATUS)
C
C This program goes through two images pixel by pixel. It writes
C the largest value of two pixels in the same parts of the two
C images to a third image. If the first two images are not the
C same size then the program quits.
C
C Read in the two input images.
C
      CALL IMG_IN('IN1,IN2',NX1,NY1,IP1,STATUS)
C
C Make an output image, modelled on the first input image
C
      CALL IMG_OUT('IN1','OUT',IPOUT,ISTAT)
C
C Do the comparison
C
      CALL BIG(%VAL(IP1),%VAL(IP2),%VAL(IPOUT),NX,NY,STATUS)
C
C Tidy up and close
C
      CALL IMG_FREE('*',STATUS)
      END

      SUBROUTINE BIG(IMAGE1,IMAGE2,IMAGE3,NX,NY,STATUS)
C
      REAL IMAGE1(NX,NY)
      REAL IMAGE2(NX,NY)
      REAL IMAGE3(NX,NY)
      INTEGER NX, NY, STATUS, I, J
C
      DO I = 1, NX
        DO J = 1, NY
          IF (IMAGE1(I,J) .GT. IMAGE2(I,J)) THEN
            IMAGE3(I,J) = IMAGE1(I,J)
          ELSE
            IMAGE3(I,J) = IMAGE2(I,J)
          ENDIF
        END DO
      END DO
      RETURN
      END
```

Notice how the code reads two distinct images in with a single statement.

The interface file, bigger.ifl, looks a little different to that which you have already seen:

```
interface BIGGER

  parameter IN1
    prompt 'Enter first file name'
  end parameter

  parameter IN2
    prompt 'Enter second file name'
  end parameter

  parameter OUT
    prompt 'Enter output file name'
  end parameter

endinterface
```

4 Accessing header information

It is often very useful for an application to be able to access the header information within an NDF, e.g. to get the airmass, time, name of the source etc. The IMG library provides a number of simple routines which allow you to gain access to the header. The following example shows how an integer header item called ALT_OBS is read from the header.

4.1 Example 6 – Reading a header item

Code:

```
      SUBROUTINE HEADER(STATUS)
C This simple code reads a header item.
      INCLUDE 'SAE_PAR'
      INTEGER STATUS     ! Global Status
      INTEGER NX         ! Number of X pixels
      INTEGER NY         ! Number of Y pixels
      INTEGER IP         ! Image pointer
      INTEGER ALTOBS     ! Item to get
C
      CALL IMG_IN('IN',NX,NY,IP,STATUS)
      CALL HDR_INI('IN','FITS','ALT_OBS',1,ALTOBS,STATUS)
      IF (STATUS .NE. SAI__OK) RETURN
      WRITE (*,*) ALTOBS
      CALL IMG_FREE('IN',ISTAT)
      END
```

Interface file:

```
interface HEADER
  parameter IN
```

```
      prompt 'Input NDF'
      endparameter
    endinterface
```

Note that this assumes that there really is an item called ALT_OBS in the header. To check this (and to find if you need to substitute something else in as a test) use the fitskeys command in Figaro to find out which items exist within the FITS header of your test files.

The above example read in an integer value. HDR_INR, HDR_INC, HDR_IND and HDR_INL read in real, character, double precision and logical header items respectively.

Other HDRroutines include:

- HDR_DELET *Delete a header item.*
- HDR_MOD *Read and/or modify a header item.*
- HDR_NAME *Return the name of a header item.*
- HDR_NUMB *Return number of items in header or number of occurrences of a named item.*
- HDR_OUT[I,R,C,D,L] *Write a header item.*

A full description of how to use these items can be found in the IMG manual (SUN/160).

5 Accessing different file formats

5.1 The Convert package

One major difficulty encountered by all astronomers at one time or another is the number of different formats data gets stored in. So far you have been using NDF files. It would be very useful to have code which reads not only NDF files, but IRAF format files, GIFS, TIFFS, ASCII, FITS etc.

Starlink has a conversion package (called Convert surprisingly) which allows you to move either directly or indirectly from one format to another. See SUN/55 for more information on the Convert package.

5.2 Getting your code to read different file formats

Wouldn't it be great if you could write a piece of code like Convert which could handle all these different formats? Guess what – you already have!

By using the IMG library, you have unknowingly given yourself the ability to access a whole range of other data formats.

All you need to do now to get to these other formats is to type

```
    % convert
```

and then you can immediately apply all the codes you've written with this book to the following format of files:

- ASCII
- DST
- FITS
- GASP
- GIF
- IRAF
- NDF
- TIFF

Try one of your codes on a file in a different format (after starting Convert of course). If you don't have any other format files, pick up some random junk from the World Wide Web. Alternatively, use Convert to change the format of a test NDF file into, for example, a GIF format file.

If you have your own special format of data, SUN/55 shows you how to add it to those which Convert can handle.

6 Summary of part one

We've now finished part one of this book. You are now in a position to do the following with a whole range of data formats:

- Open a data file.
- Manipulate the data within the file.
- Modify existing data files.
- Create new files.
- Read FITS header items.

This might be sufficient for your needs. On the other hand, you will probably be aware that there is a lot more information contained within an NDF than "just" data. For example, we have not dealt with access to statistical error and quality information stored within the file.

The IMG library is part of a much larger collection of data access routines: the NDF library. In part two of this book, we switch to the latter method of data access so that we can deal with the above issues. If you already have enough information to complete your job, there is no strict requirement to go further – go get your stuff done.

Part II

ADVANCED ACCESS TO DATA

In the second part of this book, we switch from the IMG library to the more advanced NDF data access library. We explore how the NDF library is able to access many more articles of information which can be stored within an NDF data file, such as error information and objects known as *extensions*.

Later in this half of the book, a list of helpful resources for programmers is provided, along with some helpful hints for writing applications.

7 Using the NDF library calls

To illustrate the NDF library calls, let's change the code clip.f so that the IMG calls are replaced by NDF calls. Enter the following code and save it as clip2.f.

7.1 Example 7 – Using NDF library calls to update an NDF

Code:

```
      SUBROUTINE CLIP2(STATUS)
C
      IMPLICIT NONE
      INCLUDE 'SAE_PAR'
      INTEGER STATUS, NDF1, NELM, PTR1
      REAL MIN, MAX
C
C Enable the NDF calls
C
      CALL NDF_BEGIN
C
C Associate the input NDF with some convenient label.
C In this case, let's call it NDF1. We're also going to
C access the file in such a way that it is UPDATEd.
C
      CALL NDF_ASSOC('INPUT','UPDATE',NDF1,STATUS)
C
C Map the NDF data array
C
      CALL NDF_MAP(NDF1,'Data','_REAL','UPDATE',PTR1,NELM,STATUS)
C
C Now get the threshold values
C
      WRITE (*,*) 'Enter min and max values >'
      READ (*,*), MIN, MAX
C
C Do the clipping.
C
      CALL CLIPIT(NELM,%VAL(PTR1),MIN,MAX,STATUS)
C
C Close the NDF
C
      CALL NDF_END(STATUS)
      END

      SUBROUTINE CLIPIT(NELM,VALUE,MIN,MAX,STATUS)
C
      IMPLICIT NONE
      INCLUDE 'SAE_PAR'
      INTEGER NELM, STATUS, COUNTER, NCHANGE
      REAL VALUE(NELM), MIN, MAX
C
```

```
C Check everything is OK
C
      IF (STATUS .NE. SAI__OK) RETURN
      DO COUNTER = 1, NELM
        IF (VALUE(COUNTER).GT.MAX .OR. VALUE(COUNTER).LT.MIN) THEN
        VALUE (COUNTER) = 0.0
        NCHANGE = NCHANGE + 1
        ENDIF
      END DO
      WRITE (*,*) NCHANGE, ' points were changed.'
      END
```

Interface file clip2.ifl:

```
interface clip2
   parameter INPUT
      prompt 'Input NDF'
   endparameter
endinterface
```

and compile it using

```
% alink clip2.f -L/star/lib `ndf_link_adam`
```

noting the change in the library you're linking your code to. Try running the application and compare its performance with the one that used the IMG library. You shouldn't notice any difference.

Note that you can still access GIF, IRAF, ASCII etc. formats just as you could with the IMG library. All you have to remember is to start "Convert" first. The NDF library is more versatile in terms of available file formats than its name would suggest.

You'll probably notice that a lot of the methodology behind the NDF library is similar to that which we used for the IMG library in part one. We still label NDFs. We still map them with pointers. One change you might have noticed is that when we mapped the NDF, we explicitly stated it was the *'Data'* structure we wanted to map. As we'll see later, we are not limited to this.

7.2 Creating a new NDF

In the last example, an NDF was made by updating an old one. In this section, there are three examples of creating a new NDF. The third example is the most complete, so the reader who is in a hurry could skip the first two examples.

7.3 Example 8a – Creating a new NDF (the bare essentials)

Code:

```
       SUBROUTINE CREATE(STATUS)
*
* This code just makes an empty, square NDF
*
```

```
* Parameters
*
* LBND = _INTEGER (Read)
*  Lower Bounds
*
* UBND =  _INTEGER (Read)
*  Upper Bounds
*
* OUT = NDF (Read)
*  Output NDF
*
      IMPLICIT NONE     ! No implicit typing
      INCLUDE 'SAE_PAR' ! SAE constants
      INTEGER STATUS    ! Global status
      INTEGER NDIM      ! Number of dimensions of NDF
      INTEGER LBND      ! Lower bound
      INTEGER UBND      ! Upper bound
      INTEGER INDF      ! NDF Identifier
      PARAMETER(NDIM=2) ! Set Number of Dimensions = 2
*
* Start NDF context
*
      CALL NDF_BEGIN
*
* Check status
*
      IF (STATUS .NE. SAI__OK) RETURN
*
* Read in the bounds of the NDF
*
      CALL PAR_GET0I('LBND',LBND,STATUS)
      CALL PAR_GET0I('UBND',UBND,STATUS)
*
* Create the new NDF
*
      CALL NDF_CREAT('OUT','_REAL',NDIM,LBND,UBND,INDF,STATUS)
*
* Finish up
*
      CALL NDF_END(STATUS)
      END
```

Interface file:

```
interface CREATE

  parameter LBND
    position 1
    type _INTEGER
    prompt 'Lower bound'
  endparameter

  parameter UBND
    position 2
    type _INTEGER
    prompt 'Upper bound'
  endparameter

  parameter OUT
    position 3
    prompt 'Output NDF'
  endparameter

endinterface
```

Running this code gives:

```
% create
LBND - Lower bound > 0
UBND - Upper bound > 100
OUT - Output NDF > jon
!! The NDF structure /home/TEST/jon has been
!     released from the NDF_ system with its data component in an undefined
!     state (possible programming error).
!  NDF_END: Error ending an NDF context.
!  Application exit status NDF__DUDEF, data component undefined
```

This rather ghastly error merely means that the file is empty. Using hdstrace to list the contents of the NDF gives:

```
JON  <NDF>

   DATA_ARRAY     <ARRAY>         {structure}
      DATA(101,1)    <_REAL>         {undefined}
      ORIGIN(2)      <_INTEGER>      0,0

End of Trace.
```

Now let's see how we can begin to put information into the NDF.

7.4 Example 8b – Putting a title into the NDF

Code:

```
       SUBROUTINE CREATE2(STATUS)
*
* This code just makes a square NDF with a Title Component
*
* Parameters
*
* LBND = _INTEGER (Read)
*  Lower Bounds
*
* UBND =  _INTEGER (Read)
*  Upper Bounds
*
```

```
* OUT = NDF (Read)
*  Output NDF
*
* TITLE = LITERAL (Read)
*  The NDF title
*
      IMPLICIT NONE     ! No implicit typing
      INCLUDE 'SAE_PAR' ! SAE constants
      INTEGER STATUS    ! Global status
      INTEGER NDIM      ! Number of dimensions of NDF
      INTEGER LBND      ! Lower bound
      INTEGER UBND      ! Upper bound
      INTEGER INDF      ! NDF Identifier
      PARAMETER(NDIM=2) ! Set Number of Dimensions = 2
*
* Start NDF context
*
      CALL NDF_BEGIN
*
* Check status
*
      IF (STATUS .NE. SAI__OK) RETURN
*
* Read in the bounds of the NDF
*
      CALL PAR_GET0I('LBND',LBND,STATUS)
      CALL PAR_GET0I('UBND',UBND,STATUS)
*
* Create the new NDF
*
      CALL NDF_CREAT('OUT','_REAL',NDIM,LBND,UBND,INDF,STATUS)
*
* Reset the old title if there is one
*
      CALL NDF_RESET(INDF,'Title',STATUS)
*
* Get the new title
*
      CALL NDF_CINP('TITLE',INDF,'Title',STATUS)
*
* Tidy up
*
      CALL NDF_END(STATUS)
      END
```

Interface file:

```
interface CREATE2

  parameter LBND
    position 1
    type _INTEGER
    prompt 'Lower bound'
  endparameter
```

```
    parameter UBND
      position 2
      type _INTEGER
      prompt 'Upper bound'
    endparameter

    parameter OUT
      position 3
      prompt 'Output NDF'
    endparameter

    parameter TITLE
      position 4
      type 'Literal'
      prompt 'Title'
    endparameter

  endinterface
```

Again, on running this code, a warning is reported. However, hdstrace reports:

```
JON2  <NDF>

   DATA_ARRAY     <ARRAY>         {structure}
      DATA(101,1)    <_REAL>         {undefined}
      ORIGIN(2)      <_INTEGER>      0,0

   TITLE          <_CHAR*4>       'test'

End of Trace.
```

7.5 Example 8c – Creating a data and variance array in an NDF

In this example, we put both a data array and variance array (we'll mention these in more more detail in the next section) into the file. This is done purely by mapping them:

```
       SUBROUTINE CREATE3(STATUS)
*
* This code just makes a square NDF with a Title Component
*
* Parameters
*
* LBND = _INTEGER (Read)
*  Lower Bounds
*
* UBND =  _INTEGER (Read)
*  Upper Bounds
*
* OUT = NDF (Read)
*  Output NDF
*
```

```
* TITLE = LITERAL (Read)
*  The NDF title
*
       IMPLICIT NONE     ! No implicit typing
       INCLUDE 'SAE_PAR' ! Global SAE constants
       INTEGER STATUS    ! Global status
       INTEGER NDIM      ! Number of dimensions of NDF
       INTEGER LBND      ! Lower bound
       INTEGER UBND      ! Upper bound
       INTEGER INDF      ! NDF identifier
       INTEGER IPNTR     ! Data pointer
       INTEGER VPNTR     ! Variance pointer
       INTEGER NPIX      ! Number of pixels
       PARAMETER(NDIM=2) ! Set Number of Dimensions = 2
*
* Start NDF context
*
       CALL NDF_BEGIN
*
* Check status
*
       IF (STATUS .NE. SAI__OK) RETURN
*
* Read in the bounds of the NDF
*
       CALL PAR_GET0I('LBND',LBND,STATUS)
       CALL PAR_GET0I('UBND',UBND,STATUS)
*
* Create the new NDF
*
       CALL NDF_CREAT('OUT','_REAL',NDIM,LBND,UBND,INDF,STATUS)
*
* Reset the old title if there is one
*
       CALL NDF_RESET(INDF,'Title',STATUS)
*
* Get the new title
*
       CALL NDF_CINP('TITLE',INDF,'Title',STATUS)
*
* Map data and variance arrays to force them into existence
*
       CALL NDF_MAP(INDF,'Data','_REAL','WRITE',IPNTR,NPIX,STATUS)
       CALL NDF_MAP(INDF,'Variance','_REAL','WRITE',VPNTR,NPIX,STATUS)
*
* Tidy up
*
       CALL NDF_END(STATUS)
       END
```

Interface file:

```
interface CREATE3
```

```
    parameter LBND
      position 1
      type _INTEGER
      prompt 'Lower bound'
    endparameter

    parameter UBND
      position 2
      type _INTEGER
      prompt 'Upper bound'
    endparameter

    parameter OUT
      position 3
      prompt 'Output NDF'
    endparameter

    parameter TITLE
      position 4
      type 'Literal'
      prompt 'Title'
    endparameter

  endinterface
```

Obviously an additional subroutine is needed to put useful values into the arrays.

8 Variances, bad pixels, and quality – The art of error propagation

NDFs contain a lot more than just the data values. They contain so called *headers*, along with values such as Quality and Variance associated with each pixel. For some NDFs there is also axes information.

The easiest way to take a look at what structures exist within an NDF is to use hdstrace. A sample trace might produce an output such as:

```
F2  <NDF>

   DATA_ARRAY     <ARRAY>         {structure}
      DATA(1021,200)  <_REAL>        41.46541,69.59,51.00168,48.32741,
                                     ... 169.1622,151.1605,150.1822,166.3774
      ORIGIN(2)      <_INTEGER>      1,1
      BAD_PIXEL      <_LOGICAL>      FALSE

   TITLE          <_CHAR*8>       'FEIGE 34'
   UNITS          <_CHAR*9>       'ELECTRONS'
   VARIANCE       <ARRAY>         {structure}
      DATA(1021,200)  <_REAL>        521.5062,571.9399,534.9544,532.3478,
                                     ... 587.9011,556.7531,627.0024,590.9243
      ORIGIN(2)      <_INTEGER>      1,1
```

```
         MORE            <EXT>           {structure}
            FITS(174)       <_CHAR*80>      'SIMPLE  =                      T','BI...'
                                            ... '15TELE  =
...','PACKEND','END'
            CCDPACK         <CCDPACK_EXT>   {structure}
               DEBIAS          <_CHAR*24>      'Tue Apr 29 21:11:30 1997'End of
Trace.
```

We can see from this that there are arrays for both the data and the variances. There is also some FITS information, and the file has at some point been processed by CCDPACK. Let's write a new code that processes the variances in a similar way to how clip processed the data array.

8.1 Processing a variance array

Let's illustrate the way in which the NDF library can handle variance information by developing a simple application. In the following example, an image is searched for "spikes" which are potential cosmic ray events.

When spikes are detected, the pixels are set to what is known as a *BAD* value. BAD pixels arise in situations where the values of the pixels are either unknown, useless, or meaningless. In this simple routine, we assign BAD values to pixels whose information is corrupted by cosmic rays.

In this example, any variance array present in the NDF is left untouched. This means that the variance array *is no longer valid*. It is, therefore, not passed on to the output image. Guidelines for when components of an NDF should and should not be propagated are listed in SUN/33. We will deal with a case where the variance information *is* propagated later in this section.

8.2 Example 9 – A code which does not propagate the variance array

Code:

```
      SUBROUTINE ZAPPER(STATUS)
*
* This routine zaps all pixels whose value goes above a certain
* threshold defined by the user. A new NDF is propagated from the
* old one. In this example the variance array (if present) is
* not modified.
*
* Parameters
*
* IN = NDF (Read)
*  Input NDF
*
* OUT = NDF (Write)
*  Output NDF
*
* THRESH = _REAL (Read)
*  Threshold for zapping
*
      IMPLICIT NONE     ! No implicit typing
      INCLUDE 'SAE_PAR' ! SAE constants
      INTEGER STATUS    ! Global Status
```

```
*
* Local Variables
*
      INTEGER DIM(2)    ! Dimensions of image
      INTEGER NPIX      ! Number of pixels
      INTEGER INDF      ! Input NDF identifier
      INTEGER ONDF      ! Output NDF identifier
      INTEGER IPNTR     ! Input NDF pointer
      INTEGER OPNTR     ! Output NDF pointer
      INTEGER NDIM      ! Number of dimensions
      REAL THRESH       ! Threshold value
*
* Check the "inherited status" before starting the code proper
*
      IF (STATUS .NE. SAI__OK) RETURN
*
* Obtain the input NDF. Find out how big it is in both dimensions
*
      CALL NDF_ASSOC('IN','READ',INDF,STATUS)
      CALL NDF_DIM(INDF,2,DIM,NDIM,STATUS)
*
* Read in the threshold value
*
      CALL PAR_GETOR('THRESH',THRESH,STATUS)
*
* Create a new output NDF. Model it on the old one. Propagate and
* DATA information. The variances will NOT be propagated.
*
      CALL NDF_PROP(INDF,'Data,NoVariance','OUT',ONDF,STATUS)
*
* Map the data arrays in both the input and the output files
*
      CALL NDF_MAP(INDF,'Data','_REAL','READ',IPNTR,NPIX,STATUS)
      CALL NDF_MAP(ONDF,'Data','_REAL','WRITE',OPNTR,NPIX,STATUS)
*
* Call the main working subroutine. Write new values to output
*
      CALL ZAP(THRESH,NPIX,%VAL(IPNTR),%VAL(OPNTR),STATUS)
*
* Close down
*
      CALL NDF_END(STATUS)
      END

      SUBROUTINE ZAP(THRESH,NPIX,IMAGE,OUT,STATUS)
*
* Zap pixels with more counts than THRESH by assigning BAD values to them
*
      IMPLICIT NONE       ! No implicit typing
      INCLUDE 'SAE_PAR'   ! Standard SAE constants
      INCLUDE 'PRM_PAR'   ! Define BAD constants
*
* (>) - Given,  (<) - Output
*
```

```
      REAL THRESH          ! (>) Threshold value
      INTEGER NPIX         ! (>) Number of pixels in images
      REAL IMAGE(NPIX)     ! (>) Array of input pixel values
      REAL OUT(NPIX)       ! (<) Array of output pixel values
      INTEGER STATUS       ! (<>) Global Status
*
* Local variables
*
      INTEGER N
*
* Go through image and find excess values
*
      DO 1, N = 1, NPIX
        IF (IMAGE(N).NE.VAL__BADR .AND. IMAGE(N).GT.THRESH) THEN
          OUT(N) = VAL__BADR
        ELSE
          OUT(N) = IMAGE(N)
        ENDIF
 1    CONTINUE
      END
```

Interface file:

```
interface ZAPPER

 parameter IN
  position 1
  prompt 'Input file'
 endparameter

 parameter OUT
  position 2
  prompt 'Output file'
 endparameter

 parameter THRESH
  position 3
  type _REAL
  prompt 'Threshold value'
 end parameter

endinterface
```

Note how the interface file now includes a parameter which is a *REAL* number as well as filenames.

The hdstrace application reveals that if the input NDF contained a variance array, it does not exist in the output NDF. This is because the NDF_PROP call did not explicitly list it. In order to get the variance array propagated through to the output, let's adapt the code and call it zapper2.

8.3 Example 10 – A code which DOES propagate the variance array

This code propagates the variance array. Note you will need to copy the zapper.ifl file to a zapper2.ifl file.

```
      SUBROUTINE ZAPPER2(STATUS)
*
* This routine zaps all pixels whose value goes above a certain
* threshold defined by the user. A new NDF is propagated from the
* old one.
*
* Parameters
*
* IN = NDF (Read)
*  Input NDF
*
* OUT = NDF (Write)
*  Output NDF
*
* THRESH = _REAL (Read)
*  Threshold for zapping
*
      IMPLICIT NONE     ! No implicit typing
      INCLUDE 'SAE_PAR' ! SAE constants
      INTEGER STATUS    ! Global Status
*
* Local Variables
*
      INTEGER DIM(2)    ! Dimensions of image
      INTEGER NPIX      ! Number of pixels
      INTEGER INDF      ! Input NDF identifier
      INTEGER ONDF      ! Output NDF identifier
      INTEGER IPNTR     ! Input NDF data pointer
      INTEGER OPNTR     ! Output NDF data pointer
      INTEGER VIPNTR    ! Input NDF variance pointer
      INTEGER VOPNTR    ! Output NDF variance pointer
      INTEGER NDIM      ! Number of dimensions
      REAL THRESH       ! Threshold value
*
* Check the "inherited status" before starting the code proper
*
      IF (STATUS .NE. SAI__OK) RETURN
*
* Begin the NDF context
*
      CALL NDF_BEGIN
*
* Obtain the input NDF. Find out how big it is in both dimensions
*
      CALL NDF_ASSOC('IN','READ',INDF,STATUS)
      CALL NDF_DIM(INDF,2,DIM,NDIM,STATUS)
*
* Read in the threshold value
*
      CALL PAR_GETOR('THRESH',THRESH,STATUS)
*
* Create a new output NDF. Model it on the old one. Propagate
* DATA and VARIANCES.
*
```

```
      CALL NDF_PROP(INDF,'Data,Variance','OUT',ONDF,STATUS)
*
* Map the data arrays in both the input and the output files
*
      CALL NDF_MAP(INDF,'Data','_REAL','READ',IPNTR,NPIX,STATUS)
      CALL NDF_MAP(INDF,'Variance','_REAL','READ',VIPNTR,NPIX,STATUS)
      CALL NDF_MAP(ONDF,'Data','_REAL','WRITE',OPNTR,NPIX,STATUS)
      CALL NDF_MAP(ONDF,'Variance','_REAL','WRITE',VOPNTR,NPIX,STATUS)
*
* Call the main working subroutine. Write new values to output
*
      CALL ZAP(THRESH,NPIX,%VAL(IPNTR),%VAL(OPNTR),STATUS)
*
* Close down
*
      CALL NDF_END(STATUS)
      END

      SUBROUTINE ZAP(THRESH,NPIX,IMAGE,OUT,STATUS)
*
* Zap pixels with more counts than THRESH by assigning BAD values to them
*
      IMPLICIT NONE       ! No implicit typing
      INCLUDE 'SAE_PAR'   ! Standard SAE constants
      INCLUDE 'PRM_PAR'   ! Define BAD constants
*
* (>) - Given,  (<) - Output
*
      REAL THRESH         ! (>) Threshold value
      INTEGER NPIX        ! (>) Number of pixels in images
      REAL IMAGE(NPIX)    ! (>) Array of input pixel values
      REAL OUT(NPIX)      ! (<) Array of output pixel values
      INTEGER STATUS      ! (<>) Global Status
*
* Local variables
*
      INTEGER N
*
* Go through image and find excess values
*
      DO 1, N = 1, NPIX
        IF (IMAGE(N).NE.VAL__BADR .AND. IMAGE(N).GT.THRESH) THEN
          OUT(N) = VAL__BADR
        ELSE
          OUT(N) = IMAGE(N)
        ENDIF
 1    CONTINUE
      END
```

Note that although the variance array was mapped, none of its values were changed in this simple application. The variance array was mapped and therefore allocated pointers (although this wasn't necessary to force the variance propagation). The interested reader might wish to try to adapt the call to ZAP to fix the variance values using these pointers.

8.4 The Quality component

In addition to the Data and Variance components, some NDFs also contain a *Quality* component. The quality array is not usually accessed directly by the user. Instead, the processing of the Data and Variance arrays *automatically* takes into account the values in the Quality array unless the application explicitly accesses and maps the Quality array.

Why bother with a Quality array when there is a variance array? The Quality value of a pixel can be used to describe its suitability for performing a particular task. For example, imagine a situation where photometry is performed on an object. The quality array could be set to one value for the object and another for the sky pixels.

Manipulation of the Quality array is not commonly done in most applications and so there will be no further discussion of it here. SUN/33 contains a full discussion of how to access the Quality array.

8.5 Rules for propagation of variance arrays

It is not appropriate to propagate the values in the variance array in all circumstances. For example, adding two NDFs together will result in new variances for the result. If these new variances are not calculated, the output NDF must not be allowed to contain a variance array. On the other hand, if a constant is added to an NDF, the variances can be passed on to the output with no calculations necessary.

The default used by the routine NDF_PROP is *not* to propagate the data, quality, or variance arrays. This explains why is was necessary to explicitly list what we wanted to propagate in the last example.

9 NDF extensions

The NDF format is an example of the HDS (Hierarchical Data Structure) format. Essentially, this means that lots of different items of information can be put into the same file. For example, we have already seen how the Data and Variance arrays come as distinct pieces of information.

All NDFs have a number of standard components, many of which (but not all) we have dealt with in this book. HDS (and therefore NDF) files may also contain *extensions*. These may be so commonly used they are virtually standard (e.g. a FITS header), or be specific to a particular instrument or software package (e.g. the IRAS extension).

In this section, we deal with how to access and create extensions to NDF files.

9.1 Example 11 – Reading an extension

The following example reads in an IRAS extension:

```
      SUBROUTINE EXTEND(STATUS)
*
* This application looks for an IRAS extension to an NDF.
* If it is present, it reads and prints the OFFSET value
```

```
*
      INTEGER STATUS               ! Global Status
      INTEGER INDF                 ! NDF identifier
      INTEGER OFFSET               ! Item from the IRAS extension
      INCLUDE 'DAT_PAR'            ! Define DAT__SZLOC
      CHARACTER * (DAT__SZLOC) LOC ! Locator for the IRAS extension
      LOGICAL EXIST                ! TRUE if the IRAS extension exists
      INCLUDE 'SAE_PAR'            ! Define the SAE constants
*
* Begin the NDF context
*
      CALL NDF_BEGIN
*
* Get the name of the NDF
*
      CALL NDF_ASSOC('IN','READ',INDF,STATUS)
*
* Find out if the IRAS extension exists or not
*
      CALL NDF_XSTAT(INDF,'IRAS',EXIST,STATUS)
*
* If it does, print it to the screen. If not, finish up.
*
      IF (EXIST) THEN
        CALL NDF_XLOC(INDF,'IRAS','READ',LOC,STATUS)
        CALL CMP_GET0I(LOC,'OFFSET',OFFSET,STATUS)
        WRITE (*,*) 'Offset = ', OFFSET
*
* Annul the locator
*
        CALL DAT_ANNUL(LOC,STATUS)
      ELSE
        WRITE (*,*) 'No IRAS extension exists in this file'
      ENDIF
*
* Tidy up
*
      CALL NDF_END(STATUS)
      END
```

Interface file:

```
interface EXTEND
  parameter IN
  prompt 'Input NDF'
  endparameter
endinterface
```

Note that the call to CMP_GET0I should be replaced by CMP_GET0R for real numbers, CMP_GET0D for double precision numbers, CMP_GET0L for logical values, CMP_GET0C for character values.

9.2 Example 12 – Creating a new extension

In the following example, a code is used to place the temperature of the CCD chip used during the observations into a new extension. In practice, many more items could also be put there such as pixel scale, size, gain etc. These items can then be read and used by later applications as part of the data reduction process.

```
      SUBROUTINE EXTEND2(STATUS)
*
* This application puts a value for the CCD temperature into
* an extension designed to hold information about the CCD's
* run time properties.
*
      INTEGER INDF                 ! NDF identifier
      INTEGER STATUS               ! Global Status
      REAL TEMP                    ! CCD Temperature
      LOGICAL EXIST                ! TRUE if the extension exists
      INCLUDE 'DAT_PAR'            ! DAT constants
      CHARACTER * (DAT__SZLOC) LOC ! Extension Locator
      INCLUDE 'SAE_PAR'            ! SAE constants
*
* Begin the NDF context
*
      CALL NDF_BEGIN
*
* Get the name of the NDF
*
      CALL NDF_ASSOC('IN','UPDATE',INDF,STATUS)
*
* Find out if the CCD extension exists or not
*
      CALL NDF_XSTAT(INDF,'CCD',EXIST,STATUS)
      IF (.NOT. EXIST) THEN
*
* Create the new extension
*
        CALL NDF_XNEW(INDF,'CCD','CCD_EXTENSION',0,0,LOC,STATUS)
*
* If the temperature component doesn't exist then make it
*
        CALL CMP_MOD(LOC,'TEMPERATURE','_REAL',0,0,STATUS)
*
* Get the _REAL value for the temperature
*
        CALL PAR_GETOR('TEMP',TEMP,STATUS)
*
* Put the _REAL value into the extension
*
        CALL CMP_PUTOR(LOC,'TEMPERATURE',TEMP,STATUS)
*
* Annul the locator
*
        CALL DAT_ANNUL(LOC,STATUS)
      ELSE
```

```
          WRITE (*,*) 'CCD extension already present'
        ENDIF
*
* Tidy up
*
        CALL NDF_END(STATUS)
        END
```

Interface file:

```
interface EXTEND2

  parameter IN
  prompt 'Name of NDF'
  endparameter

  parameter TEMP
  type _REAL
  prompt 'Temperature'
  endparameter

endinterface
```

10 Making your own software package

10.1 Monoliths

Sometimes it can be useful to group several applications together into one package. The result is called a *monolith*. A monolith consists of a “top level” code which then calls the various applications as needed. The monolith also has its own interface file (whose structure is slightly different from those of applications). The following monolith brings together examples 8a, 8b, and 8c into a single package called newndf.

10.2 Example 13 – A monolith

Code:

```
        SUBROUTINE NEWNDF(STATUS)
*
* This monolith puts together all three create applications
* into a single package.
*
        INCLUDE 'SAE_PAR'
        INCLUDE 'PAR_PAR'
        INTEGER STATUS
        CHARACTER * (PAR__SZNAM) ACTION
*
        IF (STATUS .NE. SAI__OK) RETURN
```

```
*
* Get the action name
*
       CALL TASK_GET_NAME(ACTION,STATUS)
*
* Go through the commands
*
       IF (ACTION .EQ. 'CREATE') THEN
         CALL CREATE(STATUS)
       ELSEIF (ACTION .EQ. 'CREATE2') THEN
         CALL CREATE2(STATUS)
       ELSEIF (ACTION .EQ. 'CREATE3') THEN
         CALL CREATE3(STATUS)
       ELSE
         WRITE (*,*) 'Dunno!'
       ENDIF
       END
```

Note how the names of the three separate tasks are in CAPITAL LETTERS. Lower case will not work, even though the filenames of your applications usually are lower case. The corresponding interface file is:

```
monolith NEWNDF

interface CREATE
  parameter LBND
    position 1
    type _INTEGER
    prompt 'Lower bound'
  endparameter
  parameter UBND
    position 2
    type _INTEGER
    prompt 'Upper bound'
  endparameter
  parameter OUT
    position 3
    prompt 'Output NDF'
  endparameter
endinterface

interface CREATE2
  parameter LBND
    position 1
    type _INTEGER
    prompt 'Lower bound'
  endparameter
  parameter UBND
    position 2
    type _INTEGER
    prompt 'Upper bound'
  endparameter
  parameter OUT
    position 3
    prompt 'Output NDF'
  endparameter
  parameter TITLE
    position 4
    type 'Literal'
    prompt 'Title'
  endparameter
endinterface

interface CREATE3
  parameter LBND
    position 1
    type _INTEGER
    prompt 'Lower bound'
  endparameter
  parameter UBND
    position 2
    type _INTEGER
    prompt 'Upper bound'
  endparameter
  parameter OUT
    position 3
    prompt 'Output NDF'
  endparameter
  parameter TITLE
    position 4
    type 'Literal'
    prompt 'Title'
  endparameter
endinterface

endmonolith
```

To compile the code, alink is used thus:

```
alink newndf.f create.f create2.f create3.f -L/star/lib `ndf_link_adam`
```

but one further stage is still required to get the monolith to work. Soft links must be made from the monolithic executable to the names of the tasks it contains. In other words, to get the above example to work you must type:

```
ln -s newndf create
ln -s newndf create2
ln -s newndf create3
```

Most Starlink packages also use a script to set up aliases to these soft links. For example, when Figaro is started, the user is really running a script called:

```
/star/bin/figaro/figaro.csh
```

11 Compiling code without ADAM

Throughout this book, the codes have been compiled using the ADAM software environment. While useful, it does not necessarily *have* to be the case. So called Standalone applications are independent of ADAM and have no interface files. They are compiled with the standard f77 command and flags rather than the alink command. For example:

```
% alink myprog.f -L/star/lib ‘ndf_link_adam‘
```

would be replaced by:

```
% f77 myprog.f -L/star/lib ‘ndf_link‘ -o myprog
```

Note, however, that there are some NDF routines which rely on the parameter system. For more details, see Appendix 3 of SUN/33.

12 Miscellaneous

12.1 A brief word about axes

Some NDFs contain information about axes other than just the pixel coordinate numbers of each pixel in the image, e.g. the wavelength scale of a spectrum. The issue of manipulating axes is complex and somewhat beyond the scope of this cookbook. There are already applications in existence which modify axis scales for tasks such as wavelength and flux calibration (such as those in Figaro).

It is suggested that the reader who really *needs* to manipulate axes consults the NDF programming manual directly (SUN/33).

12.2 Starlink libraries

One of the reasons Starlink came into existence was to try to reduce the duplication of effort amongst astronomers. Why should there be an individual flat fielding routine for every astronomer? One of the best ways to reduce the amount of duplication of effort is by using software *libraries*. These contain large amounts of code already written for you. For example, if you need to write a code which works out the Julian Date of an object, you can use the SLALIB Library.

You can think of these libraries as a bunch of subroutines scrunched up together in one big file. All you have to do to use them is to write a “top layer” of Fortran (or sometimes C) code to call them. In fact, this is what you’ve been doing throughout this book. The IMG and NDF libraries are called by the top layer of code which you write.

Just like you had to link your application to the NDF and IMG libraries, you also have to link to any other libraries you call. Check the reference manual of the libraries you need to use to find out how to use alink appropriately.

There are a number of libraries available for you to use when writing your applications or packages. Starlink provides access to a number of them not covered in this book and they are listed in Appendix A of this book.

Note that Starlink no longer uses NAG libraries. The reason for this is that NAG is a commercial package and as such, cannot be freely distributed. The PDA library *can* be distributed freely. If you intend to pass your software on to other users, you might need to take the same approach.

12.3 Hints for writing good applications

There are two groups of people you need to think about in writing a "public" application. The first group is the most obvious: the people who are going to use it. A code containing the slickest algorithms ever seen is no use to anyone if they can't understand the instructions! This can be avoided by paying attention to both your run time prompts and user documentation.

The second, less obvious, group of people you need to cater for are those who support the code. If that person is you the author, then it is all too tempting not to comment your code adequately. On the other hand the code might need to be passed on to a colleague. There is nothing worse for the useful lifetime of a package than giving the support team thousands of lines of uncommented, poorly structured code. Write your code to be as friendly to the programmer as it is to the user.

12.4 Documenting your code

There are three ways to document a code. Firstly, write a manual or manuals. You certainly need a "User Guide" telling the user how to use the package, what it does, and to some small extent how the inner workings function. If the latter is getting too technical i.e. full of "this algorithm incorporates bivariate Chebyshev polynomial solutions", perhaps it's time to write the "Reference Guide". There's no hard and fast rule as to how to do this. It's partly a matter of taste but more a case of knowing your audience. Some advice is given in SGP/28.

Secondly, comment the source code. Firstly and most importantly, the programmer must be shown how the code works, i.e. "what goes where", "this line does" and "this piece of code is commented out because it was both too slow and unaesthetic". Don't be verbose (like some of the examples in this book have been), but make sure it's clear.

Thirdly, there is online help. There are a number of Starlink facilities to help implement this. The HLP package helps develop your online support facilities.

12.5 Portability

At the time of writing, Fortran 90 is not yet easily portable from one platform to the next. It is recommended that the established Fortran 77 is used for writing applications for the time being.

12.6 Submitting code to Starlink

Starlink welcomes software contributions from the astronomical community. If you feel you may have software to contribute to the Starlink Project, please contact the Starlink Librarian (starlink@jiscmail.ac.uk) who will be glad to hear from you.

A Useful resources

A.1 Useful libraries and commands

Convert – *(Automatic) format conversion* - (SUN/55)

Findme – *Documentation search via a Web browser* - (SUN/188)

FIO – *Fortran file handling* - (SUN/143)

HLP – *On-line documentation support* - (SUN/124).

IMG – *Simple data access library* - (SUN/160)

MEMSYS – *Maximum entropy image reconstruction* - (SUN/117)

NDF – *More advanced data access library* - (SUN/33)

PDA – *Public domain mathematical algorithms (as used by Starlink Software)* - (SUN/194)

Showme – *Web-based document retriever* - (SUN/188)

SLALIB – *Positional and time information* - (SUN/67)

A.2 Further reading

ADAM – *The Starlink software environment* - (SG/4)

ADAM – *Introduction to ADAM programming* - (SUN/101)

ADAM – *Unix version* - (SUN/144)

ADAM – *Programmer's facilities and documentation guide* - (SG/6)

ADAM – *Graphics programmer's guide* - (SUN/113)

ADAM – *Interface module, reference module* - (SUN/115)

ADAM – *Guide to writing instrumentation tasks* - (SUN/134)

NDF – *Adding format conversion facilities to NDF data* - (SSN/20)

CNF – *C and Fortran mixed programming* - (SUN/209)

FITSIO – *Disk FITS input/output subroutines* - (SUN/136)

FIO/RIO – *Fortran file I/O routines* - (SUN/143)

FTNCHEK – *A Fortran77 source-code checker* - (SUN/172)

FORCHECK – *Fortran verifier and programming aid* - (SUN/73)

HDSTRACE – *HDS data file listing* - (SUN/102)

How to write good documents for Starlink - (SGP/28)

IRAFSTAR – *The IRAF/Starlink inter-operability infrastructure* - (SSN/35)

MERS (MSG and ERR) – *Message and error reporting systems* - (SUN/104)

Starlink applications programming standard – (SGP/16)

Starlink C programming standard – (SGP/4)

B Glossary

Application – A piece of software which operates on data to produce some desired result.

C – A programming language used in some applications and libraries.

Extensions (or HDS extensions) – Components of an HDS (e.g. an NDF) file used to store additional information about data.

f77 – the Fortran 77 compiler command

f90 – the Fortran 90 compiler command

Fortran – The Starlink preferred language for applications programming.

Header – Information at the start of a file which describes the origins, components, and sometimes the history of the file.

HDS – The Hierarchical Data Structure file format. This allows many pieces of distinct information to be stored in one file. NDFs are an example of the HDS format.

IMG library – A simple library used to gain access to NDF files.

Interface file – A file which the ADAM system uses to find out information about the various parameters needed to use a code.

Monolith – A code which groups many applications together into a single package.

NDF – N-dimensional data format. This is a way of storing many different types of information in the same file. All Starlink Data Format files (.sdf files) are NDFs.

NDF library – A collection of routines used to access and manipulate all the various components of an NDF file.

Package – A larger piece of software consisting of many small applications.

Parameters – Items of information which the code needs to run. These are usually provided by the user.

Mapping – A way of describing the layout of an array within an NDF to the codes that manipulate them.

Quality array – An array of values associated with all the pixels in a file which report on the status of the pixel e.g. good/bad.

Starlink – The RAL-based organisation responsible for coordinating the UK astronomical computing effort.

Variance array – An array of values associated with all the pixels in a file which correspond to the squared standard deviation of the data value of those pixels.