SUN/101.2

Starlink Project
Starlink User Note 101.2

Jo Murray

24 January 1991

Introduction to ADAM Programming

Abstract

This document is a tutorial in the art of ADAM programming. ADAM is the Starlink Software Environment.

Contents

1 Introduction

This document is intended to provide a painless introduction to the art of ADAM programming. A series of programs which are increasingly ambitious in scope are presented and explained.

The next two sections briefly explain the concept of the Starlink standard data structure (the NDF), and how to compile and link ADAM programs. Subsequent sections deal with topics such as accessing NDFs, error and message reporting, data processing, I/O, character handling, graphics, bad pixel handling *etc.* Later sections illustrate the power of running ADAM programs under the special *Interactive Command Language* (ICL) rather than the familiar DCL (Digital Command Language). A selection of reference material is presented in the appendices.

All the files, including data files, referred to in this document can be found in the ADAM_EXAMPLES directory on Starlink machines.

Several points about the programs should be made.

- The example programs are each intended to illustrate only one or two aspects of ADAM programming, and are not presented as comprehensive applications. For example, the only program which can handle 'bad' pixels is that intended to illustrate that particular topic. Of course, many more of the programs ought to address this particular problem.
- VAX extensions to Fortran are freely used, but the reader is referred to SGP/16 for a discussion of the implications of using these.
- Prologues are omitted in the interests of brevity, but example ADAM prologues are shown in Section 20.
- The link command for each program is contained in ADAM_EXAMPLES:LINK.COMMANDS. However, the linking procedures described in this document are all liable to change in the near future. Both this document and the LINK.COMMANDS file will be updated as changes occur.

No previous experience of ADAM is assumed.

The reader is referred to SG/4 for an overview of ADAM and the various application packages which are available within it.

2 Starlink data structures

One of the sources of irritation when using applications software is the variety of data formats in existence. The typical package has lots of applications solely for the purpose of reading in different types of data.

A major preoccupation of Starlink since its inception has been to design a format which is both standard and yet which can accommodate most of the data objects which one might wish to store. The solution, the NDF, (Extensible *n*-Dimensional-Data format) uses the Hierarchical Data System (HDS, SUN/92), and is described in awesome detail in SGP/38.

However, the essence of the system is simple; data objects in an NDF are stored in a logical hierarchical structure which can be compared to the VMS directory structure. At each level there are objects which may be *primitives* or *structures*. Primitives actually contain data – like ordinary VMS files, whereas structures contain further levels of objects – like VMS directory files.

There are defined locations for standard items such as the main data array, axes, title, units *etc.* (see Appendix A). The only mandatory item is the main data array; all other items are optional. Non-standard items are stored in *extensions* as described in Section 16.

However, the huge advantage of this system is that the programmer doesn't need to know the details of the format at all! A set of routines has been provided to give access to all the standard components of an NDF. A full description of this NDF subroutine library is given in SUN/33; see also Appendix B.

For example, the title of an NDF can be read into the character string VALUE by the call below:

```
CALL NDF_CGET (NDF, 'TITLE', VALUE, STATUS)
```

The units and label can be accessed by replacing `'TITLE'` with `'UNITS'` and `'LABEL'` respectively.

Of course, you still want to know what the format actually is... You can examine the contents of a sample NDF using `TRACE`[1]. NDFs have the default file extension '.SDF' and a selection are contained in the ADAM_EXAMPLES directory. The example below uses SPECTRUM.SDF.

```
$ TRACE SPECTRUM

SPECTRUM  <NDF>

  DATA_ARRAY(852)  <_REAL>       56.47374,97.49321,68.82304,82.95155,
                                 ... 820.8976,570.0729,471.8835,449.574
  TITLE          <_CHAR*30>      'HR6259 - a Red Giant in w Cen'
  LABEL          <_CHAR*4>       'Flux'
  AXIS(1)        <AXIS>          {structure}

  Contents of AXIS(1)
     LABEL          <_CHAR*20>      'Wavelength'
     UNITS          <_CHAR*20>      'Angstroms'
     DATA_ARRAY(852)  <_REAL>       3849.26,3849.79,3850.32,3850.849,
                                    ... 4298.309,4298.838,4299.368,4299.897
End of Trace.
```

[1]Type ADAMSTART to set up the symbol TRACE.

The indentation reflects the hierarchy of the data objects. For example, the *first-level* objects in the structure above are `DATA_ARRAY, TITLE, LABEL` and `AXIS(1)`. The first three of these are primitives whereas `AXIS(1)` is a structure and contains the *second-level* objects `LABEL, UNITS & DATA_ARRAY`.

The output also indicates that the main data array is of type `_REAL`[2] and is a 1-d array with 852 elements; the first and last of these are shown, separated by an ellipsis. Similarly TITLE is an object of type `_CHAR` and length 30, and has a value of `'HR6259 - a Red Giant in w Cen'`.

It is important to note that the above is an example format which happens to show a primitive NDF, *i.e.* `DATA_ARRAY` is a primitive object. SGP/38 describes other possibilities or *variants* in which `DATA_ARRAY` is a structure which can be used to express data in a variety of ways.

[2]HDS data types `_REAL` and `_CHAR` correspond to Fortran types REAL and CHARACTER respectively (see Appendix C).

3 Compiling, linking and running a simple ADAM program

An ADAM application which simply writes the message 'Hello' is considered below. The first surprise for the new ADAM programmer is that the application consists merely of a subroutine, HELLO.FOR, and an associated *interface* file, HELLO.IFL. The application code, HELLO.FOR is as follows:

```
      SUBROUTINE HELLO (STATUS)
      INTEGER STATUS

*   Output Hello message.
      WRITE(*,*) 'Hello'
      END
```

And the interface file, HELLO.IFL:

```
interface hello
endinterface
```

All ADAM applications comprise a 'main' subroutine with the single integer `STATUS` argument. The program which calls the subroutine is automatically generated and compiled at the ADAM `LINK` stage. The files thus created, `APPMAIN.FOR` and `APPMAIN.OBJ` are automatically deleted when the executable file (named after the application subroutine) is created.

Each application also has an associated *interface file*. As the name suggests, the interface file is used to provide a flexible and powerful interface to ADAM programs. The primary purpose of interface files is to facilitate the passing of values between the user and the program. ADAM programs do not normally use Fortran READ statements (see Section 9 for an exception). Instead, values which are input to a program are accommodated by *parameters*. Usually an interface file comprises a list of such parameters together with information associated with each parameter. For example, the program in the next section has a single parameter called INPUT, and the interface file indicates that this parameter should be prompted for with the string '`Input NDF structure`'. The interface file is automatically processed when the associated program runs. In the case of the HELLO program, there are no parameters, so the interface file just contains the two lines shown above. This subject of interface files is considered in more detail in Section 7.

N.B. The program HELLO.FOR is presented only to illustrate the structure of an ADAM application and is deficient in many respects. For example, the WRITE statement would not be used in a proper ADAM program (see Section 5).

Before compiling and linking ADAM programs it is necessary to issue the following commands:

```
$ ADAMSTART
$ ADAMDEV
```

These commands set up many symbols, logical names *etc.* such as the special ADAM link command ALINK. It may be convenient to include them in your LOGIN.COM.

To produce an executable file, the source files `ADAM_EXAMPLES:HELLO.FOR, HELLO.IFL` can be copied and the procedure below followed:[3]

[3]To link with the debug option add " `/DEBUG`" at the end of the link command line.

```
$ FORTRAN HELLO                                    ! Produces HELLO.OBJ
$ ALINK HELLO                                      ! Produces HELLO.EXE
```

The HELLO program can now be tested:

```
$ RUN HELLO
Hello
```

4 A simple program

The program discussed in this section reports the dimensions of the main data array of an input NDF. The code for REPDIM.FOR is reproduced below and can be copied from the directory ADAM_EXAMPLES.

```
      SUBROUTINE REPDIM (STATUS)
      IMPLICIT NONE
      INCLUDE 'SAE_PAR'
      INTEGER DIM(10), I, NDF1, NDIM, STATUS

*   Check inherited global status.
      IF (STATUS.NE.SAI__OK) RETURN

*   Begin an NDF context.
      CALL NDF_BEGIN

*   Get the name of the input NDF file and associate an NDF identifier with it.
      CALL NDF_ASSOC ('INPUT', 'READ', NDF1, STATUS)

*   Enquire the dimension sizes of the NDF and write them out.
      CALL NDF_DIM (NDF1, 10, DIM, NDIM, STATUS)
      IF (STATUS.EQ.SAI__OK) THEN
         WRITE(*,*) NDIM, (DIM(I), I=1,NDIM)
      ENDIF

*   End the NDF context.
      CALL NDF_END (STATUS)
      END
```

The interface file REPDIM.IFL, associated with the above routine is as follows:

```
interface REPDIM
  parameter      INPUT
     prompt      'Input NDF structure'
  endparameter
endinterface
```

The application code explained.

But how does it work? To examine the code in detail:

```
      SUBROUTINE REPDIM (STATUS)
      IMPLICIT NONE
      INCLUDE 'SAE_PAR'
      INTEGER DIM(10), I, NDF1, NDIM, STATUS
```

The main subroutine is declared with the single STATUS argument as with all ADAM applications. The `IMPLICIT NONE` statement forces the explicit declaration of the type of all variables used within the program. The use of this extension is generally recommended for Starlink programming. The next statement includes the file with logical name SAE_PAR which contains

a set of symbolic constants used in ADAM. The only such constant used in this example is SAI__OK which corresponds to the 'OK' value of STATUS [4]. The fourth statement above simply declares the variables used.

```
*   Check inherited global status.
      IF (STATUS.NE.SAI__OK) RETURN
```

Most subroutines used in ADAM include the variable `STATUS` as the last argument. (In the case of 'main' subroutines like REPDIM.FOR, STATUS is the only argument.) The STATUS value is checked on entering a routine, and if it does not correspond to the 'OK' value (SAI__OK), control simply returns to the calling routine. This method of *inherited status checking* greatly simplifies the coding of applications as it is unnecessary to keep checking STATUS before proceeding with the next stage of the program – if STATUS has been set to a non-ok value, then a succession of calls to a series of routines will simply 'fall through' with each subroutine returning control as soon as it tests the STATUS value.

```
*   Begin an NDF context.
      CALL NDF_BEGIN
```

This statement begins an NDF *context* which ends with the NDF_END at the end. The significance of this *context* is that any clearing up made necessary by NDF routines called during the context will be done automatically by the NDF_END. The programmer familiar with HDS will realise that this means there is no need to worry about explicitly annulling identifiers or unmapping arrays *etc.*

```
*   Get the name of the input NDF file and associate an NDF identifier with it.
      CALL NDF_ASSOC ('INPUT', 'READ', NDF1, STATUS)
```

This is the statement which uses the interface file. The first argument of the NDF_ASSOC call is an ADAM *parameter* – in this case called `'INPUT'`. Parameters are used to refer to values which are input by the user of an ADAM program. The parameter `'INPUT'` is defined in the interface file, and referring to it causes the prompt specified in REPDIM.IFL to be issued. The value supplied by the user is returned to the program. The file named is opened, in this case with 'READ' access as specified, and an NDF identifier (NDF1) used to refer to the input NDF is returned.

```
*   Enquire the dimension sizes of the NDF and write them out.
      CALL NDF_DIM (NDF1, 10, DIM, NDIM, STATUS)
      IF (STATUS.EQ.SAI__OK) THEN
         WRITE(*,*) NDIM, (DIM(I), I=1,NDIM)
      ENDIF
```

These lines do most of the work of the program. The NDF_DIM call will return the dimensions of the main data array for the NDF associated with the NDF1 identifier. Ten is an arbitrary choice for the maximum number of dimensions which the program expects. DIM and NDIM accommodate the dimensions and the total number of dimensions respectively. The WRITE

[4] If you `$ TYPE SAE_PAR` you will probably see that the value of SAI__OK is zero, but different implementations of ADAM may redefine the value of this and other symbolic constants, so it is important that they are used in preference to the numbers they represent in programs.

statement used to report the answers would not be found in a proper ADAM program (see overleaf). Note that it is necessary to check STATUS before executing the WRITE statement, as it would not be appropriate to output the answers if STATUS were not OK. (Of course, if the output were done using a routine which used inherited status checking, the check would be unnecessary.)

```
*   End the NDF context.
      CALL NDF_END (STATUS)
      END
```

The NDF_END matches the NDF_BEGIN, and as described above, this routine does any tidying up incurred by calls made since the last NDF_BEGIN.

The program can be compiled and linked and tested with an NDF as shown below:

```
$ FORTRAN REPDIM
$ ALINK REPDIM
$ RUN REPDIM
INPUT - Input NDF structure > SPECTRUM
         1         852
```

5 Error and message reporting

As mentioned previously, ADAM programs should not normally use Fortran WRITE (nor indeed READ) statements[5]. This is because using these would subvert the sophisticated way in which ADAM performs I/O.

Two sets of subroutine libraries are available to output messages; these are the MSG_ and ERR_ routines, with the latter reserved for error reports (see SUN/104 for a full description). The ADAM ALINK command automatically links in these subroutine libraries.

Message Reporting.

The primary routine for reporting messages is MSG_OUT, which has the calling sequence below, where MSG is a string containing the message name, which is often blank[6], TEXT is the message text, and STATUS is the global status. (The routine will not execute if STATUS is not OK.)

```
CALL MSG_OUT (MSG, TEXT, STATUS)
```

Thus the example below would result in the message 'HELLO'.

```
CALL MSG_OUT (' ', 'HELLO', STATUS)
```

Should the programmer wish to embed any program variables in the message, this can be accomplished with the use of *tokens*. A token is set to the value of the variable concerned using one of the `MSG_SETx` routines which have the form:

```
CALL MSG_SETx ('TOKEN', VALUE)
```

where 'x' is one of R, I, D, L or C to deal with real, integer, double precision, logical and character variables respectively. The tokens are inserted in the message text at a point indicated by ^TOKEN.

Thus, in the example program in the previous section the integer NDIM could be reported as follows:

```
CALL MSG_SETI ('NDIM', NDIM)
CALL MSG_OUT (' ', 'No. of dimensions is ^NDIM', STATUS)
```

The dimensions are reported separately to avoid the problem of not knowing how many there are[7]:

```
CALL MSG_OUT (' ', 'Dimensions are:', STATUS)
DO I=1,NDIM
   CALL MSG_SETI ('DIM', DIM(I))
   CALL MSG_OUT ('  ', '             ^DIM', STATUS)
ENDDO
```

[5]However, Section 9 illustrates when this can be appropriate.

[6]If not blank, the message name should be unique within the application; this message name can be used to refer to alternative message text defined in the program interface file, but this procedure is not recommended.

[7]A more elegant method of formatting this message is shown in Section 11 which deals with the CHR_ character handling routines.

These changes have been made in ADAM_EXAMPLES:REPDIM1.FOR.

Values output in this way have the most concise format possible. If the programmer wishes to use a particular format this is done by setting the token with a MSG_FMTx routine instead of MSG_SETx. The former routines have the calling sequence:

```
CALL MSG_FMTx (TOKEN, FORMAT, VALUE)
```

For example if the variable Y=193.12, the two calls following result in the message on the last line:

```
CALL MSG_FMTR ('YVAL', '(1E12.3)', Y)
CALL MSG_OUT (' ', 'Y value is ^YVAL', STATUS)

 Y value is  0.193E+03
```

If an unsuitable format is used in these routines the token is not set. Generally if a token is undefined, the output message contains the token name in brackets, for example:

```
Y value is ^<YVAL>
```

N.B. ALL message tokens become undefined after a call to MSG_OUT.

Error Reporting.

Errors are reported using ERR_REP which is of similar form to MSG_OUT and uses tokens in the same way. The calling sequence is:

```
CALL ERR_REP (ERR, TEXT, STATUS)
```

Unlike MSG_OUT this routine will execute with bad status set, for obvious reasons. It is recommended that a meaningful non-blank error name is used.

In general, if a program reaches an error condition it should set STATUS to an error value, report the error and abort. For example the program IMADD can only deal with 2-D arrays:

```
IF (NDIM.NE.2) THEN
   STATUS=SAI__ERROR
   CALL MSG_SETI ('NDIM', NDIM)
   CALL ERR_REP ('IMADD_NOTIMAGE',
:        'Array is not an image, but is ^NDIM-dimensional', STATUS)
   GOTO 999
ENDIF
```

Generally the error value in an applications program should be set to SAI__ERROR, a symbolic constant defined in the SAE_PAR file which is used to represent a general error condition[8].

If STATUS is set by a subroutine, that subroutine should make an error report as described above. However the calling program may make an additional report to provide contextual information. For example:

[8]Many system routines set errors to values with associated meanings for which explicit tests can be made. An example is shown in Section 9

```
      CALL IMADD (ARRAY, CONST, STATUS)
      IF (STATUS.NE.SAI__OK)THEN
         CALL ERR_REP ('CADD_ADD', 'CADD: Error adding constant to array',
     :                 STATUS)
         GOTO 999
      ENDIF
```

N.B. ALL message tokens become undefined after a call to ERR_REP.

Message Synchronization.

When running under ICL (see Section 17) it is sometimes important to ensure that the output of messages to the terminal has been completed before subsequent screen output takes place. Such a situation occurs when messages and graphics output are interspersed in a program – messages may be reported on the graphics rather than the text plane of a VDU; an example is discussed in Section 17. The problem is avoided by making the call below immediately before any graphical output. (Unnecessary use is harmless.)

```
      CALL MSG_SYNC (STATUS)
```

6 Data manipulation

This next example adds 7.0 to all the values in the main data array of the input NDF. Although not the most useful of applications in itself, it does illustrate the method usually used in ADAM programs to manipulate data arrays and introduces the concept of *dynamic memory mapping*.

```
      SUBROUTINE ADD7 (STATUS)
      IMPLICIT NONE
      INCLUDE 'SAE_PAR'
      INTEGER STATUS, NELM, NDF1, PTR1
      REAL VALUE

*  Check inherited global status.
      IF (STATUS.NE.SAI__OK) RETURN

*  Begin an NDF context.
      CALL NDF_BEGIN

*  Obtain an identifier for the input NDF.
      CALL NDF_ASSOC ('INPUT', 'UPDATE', NDF1, STATUS)

*  Map the NDF data array.
      CALL NDF_MAP (NDF1, 'Data', '_REAL', 'UPDATE', PTR1, NELM, STATUS)

*  Assign a value of 7.0 to VALUE.
      VALUE = 7.0

*  Add the constant value to the data array.
      CALL ADDIT (NELM, %VAL (PTR1), VALUE, STATUS)

*  End the NDF context.
      CALL NDF_END (STATUS)
      END

*  Subroutine to perform the addition.
      SUBROUTINE ADDIT (NELM, A, VALUE, STATUS)
      IMPLICIT NONE
      INCLUDE 'SAE_PAR'
      INTEGER NELM, STATUS, I
      REAL A(NELM), VALUE

*   Perform the addition.
      IF (STATUS.NE.SAI__OK) RETURN
      DO I = 1, NELM
         A(I) = A(I) + VALUE
      ENDDO
      END
```

And the interface file:

```
interface ADD7
  parameter      INPUT
```

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

Dynamic memory mapping.

In order to get access to a data array, a Fortran program might declare an array of some fixed size, and read the data values into it. The problems with this approach are that such a program will contain an array larger than necessary for most purposes, and cannot deal with an array larger than that explicitly declared.

The solution is to exploit the method that most compilers use to pass values between subroutines. When a value is passed from one program unit to another via an argument list, *usually*[9] what is transferred is not the value itself, but simply the address of the storage location where the value is held. Thus a subroutine to which an array is passed does not have its own copy of that array, but just knows where to find it.

In the example program, ADD7, you will notice that the main subroutine does not declare an array at all. However it does have the INTEGER PTR1. The 'map' call below reads the data from the NDF into the computer's memory, and returns the pointer PTR1 whose value is the actual memory address of the first byte of the allocated memory. Also returned is NELM, the number of elements in the data array.

```
CALL NDF_MAP (NDF1, 'Data', '_REAL', 'UPDATE', PTR1, NELM, STATUS)
```

This address (PTR1) cannot be used to access the data array in the subroutine ADD7. *But* it can be used in the call to the subroutine ADDIT. Merely inserting PTR1 into the argument list of the call to ADDIT will not produce the desired result. This is because the subroutine will receive not the actual value of PTR1, but the address of PTR1 itself, and thus will simply have access to the integer variable PTR1 (as you would expect).

However, VAX Fortran supports a special extension called %VAL, which forces the actual *value* of a variable to be passed to the subroutine.[10] Thus in the call below, passing the argument %VAL(PTR1) is equivalent to actually passing the data array whose address is stored in PTR1.

```
CALL ADDIT (NELM, %VAL(PTR1), VALUE, STATUS)
```

An array of the correct size can then be declared in the subroutine ADDIT thus:

```
SUBROUTINE ADDIT (NELM, A, VALUE, STATUS)
INTEGER NELM
REAL A(NELM)
```

ADDIT now operates directly on the array in memory just as if it had received it in the normal way.

In this example the array is mapped for 'UPDATE' so when it is 'unmapped', the modified array is automatically written back into the NDF. (There is no explicit 'unmap' call in the example shown here, because the NDF_END will automatically annul the NDF1 identifier, and this unmaps any mapped arrays associated with that identifier.)

Several points should be noted:

[9] There is nothing in the Fortran standard to enforce this *passing by address*, so the method is not guaranteed to work on any computer.

[10] This *passing by value* is used when interfacing Fortran with C routines; the latter might need to receive not the address of a variable, but its actual value. The %VAL extension is supported by compilers on both SUN and Convex machines.

(1) In this example the program can modify the array as it was mapped for 'UPDATE'. An array mapped for 'READ' can be read but not modified; mapping with 'WRITE' access reserves an area in memory of appropriate size, but the array will be undefined until something is written to it.

(2) The program appears to assume the data array is 1-dimensional. In fact it need not be so – a mapped array of any dimensions is just a number of values stored in successive memory locations which can be considered as a 1-d array. In the example shown there is no need to consider the actual dimensions of the input data array.

If you are not convinced you can compile and link the program and try it on the 1-d SPECTRUM.SDF and 2-d IMAGE.SDF. (All the necessary files can be copied from ADAM_EXAMPLES.) Doing a TRACE on these files before and after program execution should confirm that the addition has been carried out.

7 Interface files and Parameters

It must be admitted that the previous example has limited value. The facility to specify `VALUE` at run-time would be an improvement. A simple READ statement might seem like the answer, but as mentioned before, this approach is likely to fall foul of the way in which the ADAM environment deals with I/O. In any case, using the parameter system gives enormous advantages – as will become clear. (Already one ADAM parameter has been used in the example programs – the 'INPUT' used to get the name of the input NDF for NDF_ASSOC.)

The program on the previous page could be modified by replacing the statement assigning 7.0 to VALUE with the call below which gets an ADAM parameter, in this case, named 'CONST'.

```
CALL PAR_GET0R ('CONST', VALUE, STATUS)
```

This change has been made in ADAM_EXAMPLES:ADDCONST.FOR. The 'PAR_GET' part is self explanatory, and the '0' indicates that the parameter to be retrieved is a scalar (rather than a vector, or an *n*-D object). The final 'R' in the subroutine name indicates that the routine retrieves a variable of type REAL. Similarly PAR_GET0I retrieves an integer, PAR_GET0C, a character string *etc.* A list of the PAR routines with their functions and calling sequences is given in Appendix D.

The ADAM parameter 'CONST' must now be declared in the interface file as shown:

```
interface ADDCONST
  parameter      INPUT
     prompt      'Input NDF data structure'
  endparameter
  parameter      CONST
     prompt      'Value to be added'
  endparameter
endinterface
```

When the PAR_GET0R routine is called, it looks to the interface file for instructions on retrieving the parameter in question. In this case, the only information found there is that the prompt 'Value to be added?' should be issued. If a suitable number is then entered by the user, it is assigned to VALUE and STATUS remains 'OK'. Any unsuitable response (*e.g.* a logical value) and the parameter system will issue an error message, and doggedly repeat the prompt five times or until a satisfactory value is entered.

The `prompt` is an example of a *fieldname*. Other fieldnames which are commonly defined in interface files are discussed briefly below, and a full discussion of the subject of interface files is contained in SUN/115.

- `position` – rather than simply typing `$ RUN ADDCONST`, you can set up a symbol:

  ```
  $ ADDCONST== "$mydir:ADDCONST"
  ```

  where *mydir* is the logical name of the directory which contains ADDCONST.EXE. Typing `ADDCONST` will now cause execution of the program. It is now possible to enter the parameters the program needs on the command line. Possible that is, if the program knows

the order in which to expect them. Including the lines, "`position 1`" and "`position 2`" within the parameter declarations of 'INPUT' and 'CONST' respectively provides this information. After modifying the interface file in this way, the example command below will work.

```
$ ADDCONST SPECTRUM 15
```

- `keyword` – parameters can also be specified on the command line by using *keywords*. If a keyword is not declared for a parameter in the interface file, the parameter name itself is used as the keyword. Thus the same result as in the previous item can be obtained by typing:

  ```
  $ ADDCONST CONST=15 INPUT=SPECTRUM
  ```

  It is not recommended that keywords be explicitly declared in interface files.

- `ppath` – when an ADAM program prompts for a parameter, a suggested value may be appended to the prompt string. For example:

  ```
  CONST - Value to be added / 17.4 / >
  ```

  This suggested value (17.4) is used if the user presses <CR> in response to the prompt. The fieldname `ppath` is used to specify where the parameter system gets this suggested value. For example, if ppath is declared as `current`, the value supplied for the parameter during the previous run of the program will be used. It is also possible to select a fixed default value which is supplied via the `default` fieldname. In the example interface file below, the parameter `INPUT` has been given a default value of `SPECTRUM`. However, ppath for `INPUT` has been set to `'current,default'`. This means that the prompt will contain the current value – that is, the last value assigned to the `INPUT` parameter – unless no value has been assigned, in which case the value specified by the `default` fieldname will be used. Other possibilities for defining ppath are described in SUN/115.

- `type` – this can be any of the primitive data types recognised by the ADAM system, *i.e.* '_INTEGER', '_REAL', '_DOUBLE', '_CHAR' or '_LOGICAL'. Alternatively a type such as 'NDF' can be used, but this is purely descriptive and is ignored by the parameter system. If a primitive type is specified, the parameter system will check if a value of the appropriate type has been offered, and will perform any necessary type conversion if possible. *N.B. If type is specified for a parameter, the declaration must appear before any of the `default`, `range` or `in` fieldnames are declared.*

- `help` – the help fieldname contains a single line of text which will be displayed to assist the user who types ? in response to a prompt for a parameter. However a single line of help is seldom useful so the method described in the following item may be preferred.

- `helpkey` – multi-line help from a help library module is available via this fieldname. See Section 19 for a description of creating and accessing such a help library.

- `access` – if specified this will ensure that a program does not attempt to access a parameter in a way not allowed by the value of the access fieldname. For example, the interface file for ADDCONST below specifies UPDATE access for the 'INPUT' parameter; if this were changed to READ, execution of the program ADDCONST would fail as the program can no longer update the input file and an error message would warn of 'Inconsistent access mode'.

A modified version of ADDCONST.IFL incorporating some of the above fieldnames is shown below.

```
interface ADDCONST
  parameter      INPUT          # Input NDF
     position    1
     type        NDF
     prompt      'Input NDF data structure'
     access      UPDATE
     ppath       'current,default'
     default     SPECTRUM
     help        'Enter the name of the NDF '
  endparameter
  parameter      CONST          # Scalar value to add
     position    2
     type        _REAL
     prompt      'Value to be added'
     ppath       'current'
  endparameter
endinterface
```

Some of the values assigned to fieldnames above are surrounded by quotes ' '. This is necessary if the values are character constants (such as the prompts), or contain more than one word (such as the ppath) but is optional otherwise. Comments in interface files are preceded by the '#' symbol.

8 Propagating NDFs

In the previous example the data array was modified *in situ* – no new NDF was created. However it is often preferable to create a new NDF rather than overwriting an input one. This can be done by *propagating* an existing NDF using NDF_PROP, which has the following calling sequence.

```
CALL NDF_PROP (NDF1, CLIST, PARAM, NDF2, STATUS)
```

NDF1 is the identifier for an existing NDF which is used as the model (or template) for the new NDF. CLIST contains a list of data objects separated by commas as described below. PARAM is an ADAM parameter used to retrieve the name of the new NDF. NDF2 returns the identifier allocated to the new NDF, and STATUS is the global status.

The data objects which are to be copied from the model NDF to the new one are defined in the string CLIST. By default, the HISTORY, LABEL and TITLE data objects plus all extensions are propagated to the new NDF. Others are propagated by specifying them in CLIST. The propagation of default standard items can be suppressed by specifying 'NOHISTORY' *etc.* The propagation of a particular extension is suppressed by specifying the extension name, for example, NOEXTENSION(FIGARO).

For example, the CLIST arguments below propagate the items which follow:

CLIST	**Items propagated**
`' '`	*title, label, history & extensions*
`'NOHISTORY'`	*title, label & extensions*
`'DATA,QUALITY,VARIANCE,AXIS'`	*title, label, history, extensions, data, quality, variance & axis*
`'NOEXTENSION(IRAS,FIGARO)'`	*title, label, history & extensions except IRAS & Figaro*

ADAM_EXAMPLES:ADDNEW.FOR is a modified version of the program ADDCONST considered in the last section. It uses NDF_PROP to produce a new output NDF rather than updating the input file. The associated interface file, ADDNEW.IFL, includes the parameter 'OUTPUT' to retrieve the name of the output NDF. The relevant portion of ADDNEW.FOR is reproduced below:

```
*  Obtain an identifier for the input NDF.
      CALL NDF_ASSOC ('INPUT', 'READ', NDF1, STATUS)

*  Propagate everything in the input NDF to the output.
      CALL NDF_PROP (NDF1, 'DATA,AXIS,QUALITY,VARIANCE,UNITS',
     :               'OUTPUT', NDF2, STATUS)

*  Map the output NDF data array for update.
      CALL NDF_MAP (NDF2, 'Data', '_REAL', 'UPDATE', PTR2, NELM, STATUS)

*  Get the value of the constant to be added.
      CALL PAR_GETOR ('CONST', VALUE, STATUS)

*  Add the constant value to the data array.
      CALL ADDIT (NELM, %VAL (PTR2), VALUE, STATUS)
```

Note that the input NDF is opened with `'READ'` access rather than `'UPDATE'` as in ADD-CONST.FOR.

In this example, everything including the main data array is propagated, and the copied data array in the output NDF is then mapped for update and modified *in situ*, thus avoiding the need to map the input data array. Of course this approach is not always suitable. Frequently the input data array is not propagated, but is mapped and used to generate the values which are written to the output data array. In such cases, the output data array is mapped for write access and is undefined until the program-generated data values are written to it.

What should be propagated?

A little consideration must be given to the choice of items to propagate. The basic rule is that an application program must not propagate items which may have become invalid.

For example, if the data processing is such that the original variance array is no longer valid then the output NDF must *not* contain this array. The program has the choice of producing an output with no variance array or creating a correctly evaluated variance array. (The subject of processing the variance array is considered in Section 13.) In the example program ADDNEW, all the data objects were propagated because adding a constant to the main data array does not invalidate the AXIS, VARIANCE, QUALITY *etc.*

For a given application, the standard objects in an NDF can be divided into three categories:

Those which remain valid – items in this first category are propagated unchanged. Title, history and label often fall into this category, hence their inclusion by default.

Those which are processed to retain their validity – these can be propagated and modified *in situ* or created afresh in the output NDF. The main data array often falls into this category as it is usually modified by an application. The choice of propagating the data array and modifying it *in situ*, or suppressing the propagation and creating a new array in the output NDF is application-dependent. Many applications require separate input and output arrays; for example a data array cannot be reversed *in situ*. In such cases the input data array should not be propagated as it is inefficient to copy large data arrays unnecessarily.

Those which have become invalid – these must not be propagated to the output NDF. Examples include variance arrays in cases where the program cannot or does not evaluate a new variance array.

As described in Section 16, an extension contains a set of related data objects which are not accommodated in the standard NDF. For example, the 'IRAS' extension might contain those data specific to the recording and processing of IRAS observations. The treatment of extensions obeys similar rules to that of standard NDF components and can be summarised as follows:

Extensions which the application doesn't recognise – such extensions should be propagated unchanged. (It is clearly inappropriate that the information in extensions be deleted by a general application.)

Extensions which the application recognises and is equipped to process correctly – such extensions should be propagated and processed. The application should ensure that no items become invalid.

Extensions the application recognises but realises it is not equipped to process – these must not be propagated.

The propagated NDF has the same dimensionality and data type as the template NDF – irrespective of whether the data component has been propagated or not. In the example NDF below, the data component was *not* propagated, and before values were assigned to the new data array, the output NDF had the following form:

```
IMAGE1  <NDF>

  DATA_ARRAY(256,256)  <_REAL>   {undefined}

End of Trace.
```

If the data shape and type are appropriate for the output NDF, the array can be mapped for 'WRITE' and appropriate values assigned. However, if the data shape or type are not as required in the output NDF, these can be modified using the NDF routines NDF_SBND and NDF_STYPE.

9 Reading from and writing to text files

One approach is simply to use the ADAM File Input/Output (FIO) routines to get a free logical unit number and then do normal Fortran I/O. This is the method adopted here. The alternative strategy of reading and writing character buffers (also using FIO routines) is described in APN/9.

A free logical unit number is obtained by associating a file with a *file descriptor* and then finding which logical unit number has been allocated to that file descriptor. The call to associate a file with a file descriptor (FD) has the form:

```
CALL FIO_ASSOC (FILE, ACCESS, FORM, RECSZ, FD, STATUS)
```

where the arguments are: FILE, the ADAM parameter used to retrieve the filename; ACCESS, the access mode which should be one of 'READ', 'WRITE', 'UPDATE' or 'APPEND'; FORM, the file format, which should be one of 'FORTRAN', 'LIST', 'NONE' or 'UNFORMATTED'; RECZ, maximum record size in bytes, or zero if the Fortran default is required; FD, the file descriptor, and the usual STATUS.

The logical unit number (UNIT) allocated to the file is then obtained with the call below and can be used to perform normal Fortran I/O.

```
CALL FIO_UNIT (FD, UNIT, STATUS)
```

ADAM_EXAMPLES:RDDATA.FOR reads various data from a text file and finds the mean of a set of numbers. The file format it expects is simple – a title on the first line, the number of data elements on the second, followed by the data elements in free format.

The portion of RDDATA.FOR which reads the text file is reproduced below.

```
*  Associate file with file identifier FD.
      CALL FIO_ASSOC ('INFILE', 'READ', 'FORTRAN', 0, FD, STATUS)

*  Get the logical unit number allocated for this file identifier.
      CALL FIO_UNIT (FD, UNIT, STATUS)
      IF (STATUS.NE.SAI__OK) GOTO 998

*  Read via the logical unit number. The I/O status is checked after
*  every READ and the program aborts if a non-zero I/O status occurs.

*  The first line of the input file is a character string containing
*  the title of the data array.
      READ (UNIT, '(A72)', IOSTAT=IOSTAT) TITLE
      IF (IOSTAT.NE.0) GOTO 998

*   The second line contains the number of data elements.
      READ (UNIT, *, IOSTAT=IOSTAT) NELM
      IF (IOSTAT.NE.0) GOTO 998

*   Check NELM is a suitable number (positive, non-zero, not too large etc.).
*   ...
```

```
*   Read the data array.
      READ (UNIT, *, IOSTAT=IOSTAT) (ARRAY(I),I=1,NELM)
      IF (IOSTAT.NE.0) GOTO 998
```

Note that after each read the variable IOSTAT is checked and if a non-zero value is found the program control moves to the statement labelled 998. Any ADAM program which uses READ or WRITE in this way must check for I/O errors, *i.e.* IOSTAT becoming non-zero. If an I/O error occurs, a program may be able to handle it – otherwise it should make an error report and abort. An error report is made using the routine ERR_FIOER which puts the message appropriate to the value of IOSTAT into a message token. The token is then reported using ERR_REP as shown in the further extract from RDDATA.FOR which follows.

```
998   CONTINUE
*   Report any I/O error.
      IF (IOSTAT.NE.0) THEN
         STATUS = SAI__ERROR

*      Translate IOSTAT into a message token and make an error report.
         CALL ERR_FIOER ('MSG', IOSTAT)
         CALL ERR_REP ('RDDATA_FIOER', '^MSG', STATUS)
      ENDIF
```

After opening a file with FIO_ASSOC it is necessary to cancel the ADAM parameter used for the filename, and deactivate the FIO package when interaction with the file has ceased, as shown below.

```
*   Cancel the ADAM parameter INFILE and deactivate FIO.
      CALL FIO_CANCL ('INFILE', STATUS)
      CALL FIO_DEACT (STATUS)
```

The FIO routines must be explicitly included during linking thus:

```
$ ALINK RDDATA,ADAM_LIB:FIOLINK/OPT
```

The program RDDATA can be tested with the data file DATA.DAT in ADAM_EXAMPLES.

Explicit error checking.

The program RDDATA.FOR gives up if it encounters any errors when trying to read the input file. However a program may wish to cope with various possibilities. This can be done by explicitly testing STATUS against the FIO error codes. These are made available to a program by including the file with logical name FIO_ERR. For example, if FIO_ASSOC tries to open a non-existent file for 'READ' an error will result, and STATUS will be set to the symbolic constant FIO_NOTFD. (A complete list of the FIO error codes is contained in APN/9 – or you can `$ TYPE FIO_ERR`.)

When a particular error condition is trapped in this way, the program should call the routine ERR_FLUSH as shown below. This has two effects; firstly it resets STATUS to SAI__OK, and secondly it forces the immediate output of any pending error messages. (If the error message is not flushed immediately, the user may be bewildered by error messages on program termination which refer to error conditions which have been corrected.)

```
      INCLUDE 'FIO_ERR'
*     ...
      CALL FIO_ASSOC ('INFILE', 'READ', 'FORTRAN', 0, FD, STATUS)
      IF (STATUS.EQ.FIO_NOTFD) THEN
*      File not found - take appropriate action, but first
*      reset STATUS to SAI__OK and flush the error message.
         CALL ERR_FLUSH (STATUS)
```

However the errors may happen not during FIO calls but during Fortran READ statements which set IOSTAT on failure. The IOSTAT returned by these statements can be converted to an FIO STATUS value with the routine FIO_SERR as shown in the example below.

```
      IF (IOSTAT.NE.0) THEN
*      Translate IOSTAT into an FIO STATUS.
         CALL FIO_SERR (IOSTAT,STATUS)
         IF (STATUS.EQ.FIO_EOF) THEN
*          End of file - take appropriate action.
```

10 Creating NDFs from scratch – a format conversion routine

This section presents a format conversion routine – an NDF is created from data in a text file. Only portions of the program are reproduced below; the full source code and interface file are contained in ADAM_EXAMPLES:OUTNDF.FOR and OUTNDF.IFL. (The part of the program which reads the input text file is very similar to the code used to illustrate the use of the FIO package in Section 9.)

A new NDF is created with the call:

```
CALL NDF_CREAT (PARAM, TYPE, NDIM, LBND, UBND, NDF, STATUS)
```

where, PARAM is the ADAM parameter used to get the name of the NDF; TYPE is the data type required for the main data array, NDIM is the number of dimensions for the main data array, LBND and UBND are integer arrays containing the lower and upper pixel bounds respectively as described below, NDF contains the NDF identifier allocated, and STATUS is the global status.

The arrays LBND and UBND require a little explanation. Just like a Fortran array, an NDF may have dimensions in which the array index begins with a number other than one. The shape of an NDF is completely specified by the number of dimensions and the lower and upper pixel bound of each dimension. For example (−2:6, 0:100) describes a 2-D array with pixel indices ranging from −2 to 6 in the first dimension and 0 to 100 in the second. An NDF with this shape could be created by calling NDF_CREAT with NDIM= 2, LBND$(1) = -2$, LBND$(2) = 0$, UBND$(1) = 6$ and UBND$(2) = 100$. In the example below, the lower pixel index bound is simply set to one.

The program OUTNDF reads a text file with the format shown below:

```
IUE spectrum of Saturn
534
   1191.200      1.6757190E-13
   1198.279      1.2435831E-13
   ...            ...
```

That is, a title on the first line, the number of data elements on the second, followed by successive pairs of axis and data values.

In OUTNDF.FOR the title of the NDF and the number of data elements are read into TITLE and NELM respectively. The program only deals with 1-d data so the number of dimensions, NDIM, is set to one. The lower bound of the main data array is set to one, and the upper bound is set to NELM. An NDF is then created with the call to NDF_CREAT as shown below.

The title read from the text file is used to set the NDF title. Note that the main data and axis arrays are not read in the main subroutine OUTNDF – this is to avoid the necessity of declaring arrays to accommodate them. Instead the arrays are mapped for 'WRITE' in the output NDF and the dynamically allocated space is used in the subroutine GTDATA to read in the arrays from the text file.

The code which creates the output NDF is reproduced below:

```
*  Use FIO to open the input text file and get the logical unit number.
*  Read TITLE and NELM, and check NELM is greater than zero.
```

```
*   ...

*   Begin an NDF context.
      CALL NDF_BEGIN

*  The lower pixel bound is set to unity, the upper to the number of
*  elements. The number of dimensions is set to one.
      LBND(1) = 1
      UBND(1) = NELM
      NDIM = 1

*   Create a new NDF file and associate an NDF identifier with it.
*   A data array of the correct size is specified via NDIM
*   and the LBND, UBND arrays.
      CALL NDF_CREAT ('OUTPUT', '_REAL', NDIM, LBND, UBND, NDF, STATUS)

*   Put the TITLE read from the input file into the NDF title.
      CALL NDF_CPUT (TITLE, NDF, 'TITLE', STATUS)

*   Map the NDF main data array for WRITE.
      CALL NDF_MAP (NDF, 'DATA', '_REAL', 'WRITE', DATPTR, NELM, STATUS)

*   Map the NDF AXIS(1) array for WRITE.
      CALL NDF_AMAP (NDF, 'CENTRE', 1, '_REAL', 'WRITE', AXPTR, NELM,
     :               STATUS)

*   Call a subroutine to read the input data into the mapped data arrays.
      CALL GTDATA (UNIT, NELM, %VAL(AXPTR), %VAL(DATPTR), STATUS)

*   End the NDF context.
      CALL NDF_END (STATUS)

999   CONTINUE

*   Report any I/O errors, shut down FIO and end.
*    ...
```

And the subroutine which reads the data…

```
*   Subroutine to read main data and axis arrays.
      SUBROUTINE GTDATA (UNIT, NELM, WAVE, FLUX, STATUS)
      IMPLICIT NONE
      INCLUDE 'SAE_PAR'
      INTEGER NELM, I, IOSTAT, STATUS, UNIT
      REAL WAVE(NELM), FLUX(NELM)

      IF (STATUS.NE.SAI__OK) RETURN

*   Read the data arrays.
      READ (UNIT, *, IOSTAT=IOSTAT) (WAVE(I), FLUX(I), I=1,NELM)

*   Report any I/O error.
*    ...
      END
```

The program OUTNDF can be tested with the data file SATURN.DAT in ADAM_EXAMPLES and can easily be adapted to deal with other data formats as required.

11 Character handling routines

A set of portable routines to perform tasks associated with character handling is available within ADAM. These routines have the prefix CHR, and a complete description of the specifications is given in SUN/40. Appendix E gives a list of all the routines together with their argument lists and a short description of their functions. The appendix also indicates whether each routine is implemented as a subroutine or a function. (Functions must be declared with the appropriate type in the programs using them.) The CHR library is automatically linked during the ALINK procedure. Several of the most useful CHR routines are considered below.

One set of routines can be characterised as having the form `CHR_xTOy` where `x` and `y` are each one of `C, D, I, L & R` corresponding to type `CHARACTER, DOUBLE, INTEGER, LOGICAL & REAL` respectively. For example the call below will encode the real number X as a string:

```
CALL CHR_RTOC (X, STRING, NCHAR)
```

NCHAR is the number of characters in the returned STRING. Thus if `X=1237.4`, `STRING` is returned as `'1237.4'` and `NCHAR` becomes `6`.

The example program REPDIM1 in Section 5 reported the dimensions of the input data array one after another. A tidier result can be achieved by building a string using a succession of CHR_PUTx calls, where as above, `x` can be any of C, D, I, L or R. These calls have the form:

```
CALL CHR_PUTx (VALUE, STRING, NCHAR)
```

where VALUE is a value of appropriate type, STRING is a character string which has the encoded value *appended*, and NCHAR on entry contains the position in the STRING at which the encoded VALUE is inserted and contains the length of the new string on return. The code to report the array dimensions can be changed to:

```
*   Report the dimensions.
      NCHAR = 0
      CALL CHR_PUTC ('Array dimensions are ', STRING, NCHAR)
      DO  I = 1, NDIM
*      Add a 'x' between the dimensions if there are more than one.
         IF (I.GT.1) CALL CHR_PUTC (' x ', STRING, NCHAR)
*      Add the next dimension to the string.
         CALL CHR_PUTI (DIM(I), STRING, NCHAR)
      ENDDO
      CALL MSG_OUT ('  ', STRING, STATUS)
```

ADAM_EXAMPLES:REPDIM2.FOR contains this modification. Running this program on the datafile IMAGE.SDF produces the output below:

```
No. of dimensions is 2
Array dimensions are 256 x 256
```

Another common use for the CHR routines is in the 'cleaning' and comparison of strings. The fragment of code below is used to remove blanks and determine if the units given in the two strings are the same.

```
      CHARACTER*(100) STRNG1, STRNG2
      LOGICAL YES
      LOGICAL CHR_SIMLR                  ! Logical external Function.
*     ...
      STRNG1 = '   ERGS / ( CM ** 2 * S )'
      STRNG2 = 'ergs/(cm**2*s)'

      CALL CHR_RMBLK (STRNG1)         ! Remove blanks from STRNG1
      CALL CHR_RMBLK (STRNG2)         ! Remove blanks from STRNG2
      YES =CHR_SIMLR (STRNG1, STRNG2) ! Determine if strings equal apart from case.
```

A more ambitious example taken from ADAM_EXAMPLES:GETEBV.FOR follows. The extract below compares an input object name, OBJECT, with star names in a text file (EBV.DAT) and if a match is found, the value (the E_{B-V}) associated with the star is reported. The text file format is:

```
* E(B-V) calculated from intrinsic photometry. (Accurate to within 0.03.)
*  HD#     E(B-V)
   HD886   0.02
  HD1337   0.18
   ..        ..
```

The code first tidies the input string OBJECT by removing blanks. The first character of OBJECT is then tested to see if it is a number. If only the number has been given, the string 'HD' is added at the beginning. Each line in the text file is now read; blank lines and lines beginning with '*' are ignored. Other lines are decoded into words. On each line the first word is the star name and the second is a string containing the E_{B-V} associated with that star. The star name on each line is compared with OBJECT. If a match is found the second word on that line is decoded into a real value and reported.

```
*   Remove all blanks.
      CALL CHR_RMBLK (OBJECT)

*   If only the number is given, prefix this with 'HD'.
      IF (CHR_ISDIG(OBJECT (1:1))) THEN
         NCHAR = CHR_LEN(OBJECT)
         OBJECT(1:NCHAR+2) = 'HD'//OBJECT(1:NCHAR)
      ENDIF

*   Now work through list of objects, trying to find a matching name
      FOUND = .FALSE.
      IOSTAT = 0
      DO WHILE (.NOT.FOUND)

*      Read a line from the file.
         READ (UNIT, ' (A)', IOSTAT=IOSTAT, END=998) LINE
         IF (IOSTAT.NE.0) GOTO 997

*      Ignore blank lines or lines beginning with '*'.
         IF (.NOT.((CHR_LEN(LINE).EQ.0) .OR. (LINE(1:1).EQ.'*'))) THEN

*         Decode line into words.
            CALL CHR_DCWRD (LINE, 2, NWRD, START, STOP, WORDS, LSTAT)

*         See if first word (containing star name) matches OBJECT.
            SAME = CHR_SIMLR (WORDS (1), OBJECT)

            IF (SAME) THEN
               FOUND = .TRUE.

*            Second word contains EBV. Encode this string as a REAL.
               CALL CHR_CTOR (WORDS (2), EBV, STATUS)

*            Output the star name and the associated EBV.
               CALL MSG_SETC ('OBJECT', OBJECT)
```

```
            CALL MSG_FMTR ('EBV', '(F4.2)', EBV)
            CALL MSG_OUT (' ','Star ^OBJECT has E(B-V)=^EBV', STATUS)
         END IF
      END IF
   END DO
```

12 Handling data quality

A data array may contain elements which are not of good quality. In the present context, this does not mean that perhaps data are noisier than the observer had hoped, but rather that there are data elements whose values are fundamentally flawed.

Such *bad* values can arise in a variety of ways. For example, a bad pixel in a data array may be due to a dead element in a CCD chip during an observing run. Bad values may also be the result of data processing. The example considered below deals with bad values due to attempting to take the square root of a negative number. Two methods of dealing with data quality in NDFs are available as follow:

Bad or 'magic' values – bad data values are replaced with special bad values. Each data type has an associated bad value. For example the bad value for real data on a VAX is defined as FFFFFFFF in hexadecimal, which is approximately −1.7014E38 (see SGP/38). However, this and other bad values are system dependent and programs *must* refer to them using symbolic constants. These symbolic constants are defined in the include file with logical name PRM_PAR, (see also Section 14) and are of the form `VAL__BADx` where `x` is one of `R`, `D`, `I`, `W`, `UW`, `B` or `UB` corresponding to the HDS data types `_REAL`, `_DOUBLE`, `_INTEGER`, `_WORD`, `_UWORD`, `_BYTE`, and `_UBYTE` respectively.

Quality arrays – data quality can also be indicated by using a quality array associated with a data array. Non-zero quality values generally indicate that the associated data element is bad. However a quality array is not normally used merely to differentiate between good and bad data as it requires an extra array – unlike the bad value method. The advantage of a quality array is that different indicators of quality may be set. For example, IUE data have associated flags which are used to indicate one of a range of conditions which may apply to a data element; a pixel may be subject to microphonic noise or be saturated or coincide with a reseau mark *etc.* An application may wish to differentiate between these conditions. This topic is not considered further, but the reader is referred to SUN/33, Section 10, for a description of implementing such a scheme.

An NDF may use either of the above methods – or both – or indeed have no indication of data quality at all. When a data array is mapped, the bad value for the data type is automatically inserted into the mapped array in place of any bad data elements.

ADAM_EXAMPLES:SQROOT.FOR takes the square root of each element in the input data array. Such an application must consider what to do in the event that any of the input data are negative. The correct behaviour is to check for this condition and insert the bad value for the data type as follows:

```
      INCLUDE 'PRM_PAR'                    ! Defines VAL__BADR etc
      REAL IN(NELM), OUT(NELM)
*   ........
      DO I=1,NELM
*      Test if input value is negative.
         IF(IN(I).LE.0)THEN
            OUT(I)=VAL__BADR
         ELSE
```

```
            OUT(I)=SQRT(IN(I))
         ENDIF
      ENDDO
```

However if an application is going to consider bad pixels it should also recognise the possibility of bad input pixel values. Such values should be propagated as bad (unless explicitly repaired in some fashion). So the test shown above should be amended as follows:

```
*    Test if input is negative or is itself a bad value.
         IF ( IN(I).LE.0 .OR. IN(I).EQ.VAL__BADR ) THEN
            OUT(I)=VAL__BADR
```

The above example illustrates the only two operations which should be conducted with bad values, *i.e.* assignment and comparison. It is inappropriate to perform any arithmetic function such as taking the square root of a bad value.

The bad-pixel flag.

Ideally all applications which process data should consider the possibility of bad data values. A simple application which divides each element in a data array by two may not itself give rise to new bad pixels, **but** it should trap the case where the input value is bad, as the output ought to contain the appropriate bad value, *not* the bad value divided by two!

However many data arrays contain no bad values and it is obviously undesirable that all applications be forced to check every data element as shown in the previous fragment of code. In order to address this problem, each array component (such as the main data or variance array) of an NDF has an associated logical *bad-pixel flag*. This is set to .FALSE. if there are *definitely* no bad pixels present, whereas .TRUE. indicates that bad pixels may or may not be present. (The uncertainty in the latter case arises because of the difficulty of keeping track of whether bad pixels have been set; certain operations may introduce bad pixels but the NDF system cannot be sure whether this has in fact happened without checking each data value explicitly – too time-consuming a procedure to perform by default.)

Two common situations where it is useful to know whether input data contain bad values are as follow:

(1) An application may choose not to handle data which contain bad pixels. Such an application should check whether an input data array contains such values in order that the user may be informed of the difficulty (and probably the application aborted). An example is shown in Section 13.

(2) It is more efficient for a program to deal separately with the case where no bad pixels are present.

In both the cases cited above, an application should find the value of the bad-pixel flag for any data arrays of interest. The call below will cause the value of the logical variable BAD to be set according to whether there are bad pixels in the main data array of the NDF.

```
CALL NDF_BAD (NDF, 'Data', CHECK, BAD, STATUS)
```

The input logical argument CHECK requires some explanation. If CHECK is set to .FALSE. the value of the bad-pixel flag returned is as described above, *i.e.*
`BAD=.FALSE.` $\Rightarrow$ definitely no bad pixels present, whereas `BAD=.TRUE.` $\Rightarrow$ bad pixels may be present.
However, if CHECK is set to .TRUE. this forces an explicit (and time consuming) check for the presence of bad data values and the bad-pixel flag becomes set as follows:
`BAD=.FALSE.` $\Rightarrow$ no bad pixels present, and `BAD=.TRUE.` $\Rightarrow$ bad pixels *are* present.

An application which cannot deal with bad values should use the explicit check (*i.e.* `CHECK=.TRUE.`) so that it never gives up unnecessarily. However, an application which is using the check for efficiency purposes might choose merely to look at the value of the bad-pixel flag (*i.e.* `CHECK=.FALSE.`) as the time taken to do the explicit check might negate any efficiency advantage which is gained.

An application which is aware that it has created an output data array which contains bad values should indicate this by setting the bad-pixel flag to .TRUE.; conversely if it can be confident that an output data array contains no bad values, the flag can be set to .FALSE..

The value of the bad-pixel flag for an NDF array component is set by calling NDF_SBAD as shown in this extract from ADAM_EXAMPLES:SQROOT.FOR. The program counts how many bad values it has assigned in the output array and sets the bad-pixel flag accordingly.

```
*   Set bad-pixel flag according to whether any bad pixels have been set.
*   (NBAD is the number of bad pixels which have been set.)
      IF (NBAD.EQ.0) THEN
         CALL NDF_SBAD (.FALSE., NDF2, 'DATA', STATUS)
      ELSE
         CALL NDF_SBAD (.TRUE., NDF2, 'DATA', STATUS)
      ENDIF
```

13 Processing the variance array

Errors associated with data can be accommodated in an NDF by means of a *Variance* array. Such an array has the same shape and size as the associated data array, and contains an estimate of the variance for each element in that data array. Note that it is the variance for each data element which is stored rather than the standard deviation, the former being simply the square of the latter. Variance is the chosen method of storage for errors as most error processing is considerably simpler when variance rather than standard deviation is used. For example if two numbers are added:

$$z = x + y$$

the standard deviation in the result, σ_z, is related to the standard deviations in the input numbers σ_x, σ_y, as follows:

$$\sigma_z = \sqrt{\sigma_x^2 + \sigma_y^2}$$

whereas if the variance is considered, the variance in the result V_z is simply the sum of the variances of the input numbers, *i.e.*

$$V_z = V_x + V_y$$

The latter is significantly quicker to compute.

Application programs should consider whether an input variance array remains valid after a data array has been processed. For example, the variance array remains valid when a constant with no associated error is added to each element of the main data array. In such a case the variance array should be propagated unchanged to the output NDF.

The program discussed in the previous section took the square root of each element in the main data array of the input NDF. Note that variance was *not* among the components included in the call to NDF_PROP, so any variance array in the input NDF would not be propagated to the output NDF. Obviously in this case, the input variance array is not appropriate to the output data, as taking the square root of a data set changes the variance.

However, in the case where Gaussian statistics apply and a data element is raised to a power, *i.e.*

$$y = x^n$$

the variance in y, V_y is related to the variance in x, V_x, as follows:

$$\frac{V_y}{y^2} = n^2 \frac{V_x}{x^2}$$

In the case of taking a square root ($n = 1/2$) the variance is given by:

$$V_y = \frac{V_x}{4x}$$

ADAM_EXAMPLES:SQROOTV.FOR processes the variance array according to this formula. In the interests of simplicity, the program does not try to cope with bad data values, but as described in the previous section it should ensure that it does not attempt the processing of data arrays which it cannot handle. (However in the form shown here, it will crash if the input data contain negative numbers! A more robust program is presented in the next section.)

The check for the presence of bad pixels in the input data is as follows:

```
*  Abort if main data array contains bad pixels.
      CALL NDF_BAD (NDF1, 'Data', .TRUE., BAD, STATUS)
      IF (BAD) THEN
         STATUS = SAI__ERROR
         CALL ERR_REP ('SQROOTV_BADPIX',
     :                 'Sorry, cannot cope with bad pixels', STATUS)
         GOTO 999
      ENDIF
```

The remainder of the program is summarised in the following steps. The input NDF is checked for the existence of a variance array; if such an array exists, then both the main data and variance arrays will be processed by the program, otherwise only the main data array will be processed.

NDF_MAP can be given a list of components rather than just one. In the example below the list will comprise either `'Data'` or `'Data,Variance'` as appropriate.[11] When a list is supplied to NDF_MAP, the pointer argument must be an array of at least the same size as the number of components in the list. A pointer to each mapped component will be returned via the array, in the order corresponding to that in the component list.

The output NDF is created by the propagation of the input NDF. The variance is not among the components propagated as it is more efficient in this case to create a new structure in the output NDF rather than copy the existing variance array and overwrite it. The NDF_MAP call which maps the variance for 'WRITE' in the output NDF creates a variance structure of the same size and type as the data array. If a different type is desired, the data and variance can be mapped separately with the chosen types, (of course the size of the variance array must match the data array).

The mapped arrays are passed to a subroutine which takes the square root of each element in the main data array and calculates the variance appropriate to the output NDF if a variance structure was found in the input NDF.

```
*   Check whether variance array exists
      CALL NDF_STATE (NDF1, 'Variance', VARNCE, STATUS)

*   Work out component list for NDF_MAP. If a variance array exists,
*   both data and variance are mapped - otherwise only the data.
      COMP = ' '
      IF (VARNCE) THEN
         COMP = 'Data, Variance'
      ELSE
          COMP = 'Data'
      ENDIF

*  Create a new output NDF based on the input NDF.
      CALL NDF_PROP (NDF1, 'Axis', 'OUTPUT', NDF2, STATUS )

*  Map the input and output data and variance arrays.
      CALL NDF_MAP (NDF1, COMP, '_REAL', 'READ', PNTR1, NELM, STATUS)
      CALL NDF_MAP (NDF2, COMP, '_REAL', 'WRITE', PNTR2, NELM, STATUS)

*   Take the square root of each element in the main data array.
```

[11] One advantage of mapping the data and variance arrays with a single call is that it is more efficient for NDF_MAP to access any quality array only once and insert bad pixels into the mapped data and variance arrays simultaneously.

```
*   The output variance is also generated if appropriate.
      CALL SQRTVR(NELM, VARNCE, %VAL(PNTR1(1)), %VAL(PNTR1(2)),
     :            %VAL(PNTR2(1)), %VAL(PNTR2(2)), STATUS)
```

And the code from the subroutine:

```
      SUBROUTINE SQRTVAR (NELM, VARNCE, IN, VARIN, OUT, VAROUT, STATUS)
*     ...
*   Take the square root of each input element and find the variance.
      DO I = 1, NELM
         OUT(I) = SQRT(IN(I))
         IF (VARNCE) VAROUT(I) = VARIN(I)*0.25/IN(I)
      ENDDO
      END
```

14 PRIMDAT – Primitive data processing

The PRIMDAT package (see SUN/39) provides a range of symbolic constants, functions and subroutines which aid in the processing of numeric data. Facilities to cope with all HDS numeric data types are included; manipulation involving the non-numeric types, `_LOGICAL` and `_CHAR` can be done using the CHR routines as described in Section 11.

The names of the routines and constants described below usually end with a one or two letter code appropriate to the data type to which they apply. These codes are `R, D, I, W, UW, B` or `UB` corresponding to the HDS data types `_REAL, _DOUBLE, _INTEGER, _WORD, _UWORD, _BYTE,` and `_UBYTE` respectively. For example, `VAL__BADR` is the symbolic constant which represents the bad data value for `_REAL` data, whereas `VAL__BADUB` represents the bad data value for `_UBYTE` data.

VAL, NUM and VEC routines.

The example program in Section 12 contained a subroutine which simply calculated the square root of the input data array and dealt correctly with bad data values. However the PRIMDAT package contains a set of general purpose routines which includes just such a subroutine. There are three sets of routines, which can be summarised as follow:

VAL_ functions – these perform arithmetic operations and type conversion on scalar values. Bad data handling is incorporated.

NUM_ functions – these are like the VAL routines except that bad data handling is not incorporated; numerical errors can cause these routines to crash and bad input values are interpreted literally.

VEC_ routines – these are subroutines which perform the same operations as the VAL functions but operate on arrays of numbers.

The name of a routine consists of one of the above prefixes, plus an indication of the function it performs and the type of data it expects. For example, the name of the subroutine which takes the square root of a double-precision input data array is `VEC_SQRTD`. There are also type conversion routines. For example, `VEC_RTOI` converts a real array to an integer one. A list of the formats of the routines is shown in the table on the following page.

The tables below indicate the range of arithmetic operations available. The functions in the left-hand table are implemented for all the HDS numeric types. The trigonometric functions in the right-hand table are implemented only for types `_REAL` and `_DOUBLE`; those trigonometric functions whose name ends with `D` operate in degrees; the others use radians.

Func	$\mathbf{N}_{\mathrm{arg}}$	**Operation performed**
ADD	2	addition: ARG1 + ARG2
SUB	2	subtraction: ARG1 − ARG2
MUL	2	multiplication: ARG1 ∗ ARG2
DIV	2	*(floating) division: ARG1 / ARG2
IDV	2	**(integer) division: ARG1 / ARG2
PWR	2	raise to power: ARG1 ∗∗ ARG2
NEG	1	negate (change sign): −ARG
SQRT	1	square root: $\sqrt{\mathrm{ARG}}$
LOG	1	natural logarithm: $\ln(\mathrm{ARG})$
LG10	1	common logarithm: $\log_{10}(\mathrm{ARG})$
EXP	1	exponential: $\exp(\mathrm{ARG})$
ABS	1	absolute value: $\lvert\mathrm{ARG}\rvert$
NINT	1	nearest integer value to ARG
INT	1	Fortran AINT (truncation to integer) fn.
MAX	2	maximum: $\max(\mathrm{ARG1}, \mathrm{ARG2})$
MIN	2	minimum: $\min(\mathrm{ARG1}, \mathrm{ARG2})$
DIM	2	Fortran DIM (positive difference) fn.
MOD	2	Fortran MOD (remainder) fn.
SIGN	2	Fortran SIGN (transfer of sign) fn.

Func	$\mathbf{N}_{\mathrm{arg}}$	**Operation performed**
SIN	1	$\sin(\mathrm{ARG})$
SIND	1	$\sin(\mathrm{ARG})$
COS	1	$\cos(\mathrm{ARG})$
COSD	1	$\cos(\mathrm{ARG})$
TAN	1	$\tan(\mathrm{ARG})$
TAND	1	$\tan(\mathrm{ARG})$
ASIN	1	$\sin^{-1}(\mathrm{ARG})$
ASND	1	$\sin^{-1}(\mathrm{ARG})$
ACOS	1	$\cos^{-1}(\mathrm{ARG})$
ACSD	1	$\cos^{-1}(\mathrm{ARG})$
ATAN	1	$\tan^{-1}(\mathrm{ARG})$
ATND	1	$\tan^{-1}(\mathrm{ARG})$
ATN2	2	Fortran ATAN2
		(inverse tangent) function
AT2D	2	VAX Fortran ATAN2D
		(inverse tangent) function
SINH	1	$\sinh(\mathrm{ARG})$
COSH	1	$\cosh(\mathrm{ARG})$
TANH	1	$\tanh(\mathrm{ARG})$

The following table gives the format of the routines as described above;

Format of routine	Example
`RESULT = VAL_funcx (BAD, ARG, STATUS)`	`PROOT = VAL_SQRTR (BAD, P, STATUS)`
`RESULT = VAL_funcx (BAD, ARG, ARG1, STATUS)`	`BSUM = VAL_ADDUB (BAD, B1, B2, STATUS)`
`RESULT = VAL_xTOy (BAD, ARG, STATUS)`	`IP = VAL_RTOI (BAD, P, STATUS)`
`RESULT = NUM_funcx (ARG)`	`PLOG = NUM_LOGR (P)`
`RESULT = NUM_funcx (ARG, ARG1)`	`ICUBE = NUM_PWRI (I, 3)`
`RESULT = NUM_xTOy (ARG)`	`P = NUM_DTOR (D)`
`CALL VEC_funcx (BAD, ARG, RESULT, IERR, NERR, STATUS)`	`CALL VEC_SINR (BAD, P, SINP, I, N, STATUS)`
`CALL VEC_funcx (BAD, ARG, ARG1, RESULT, IERR, NERR, STATUS)`	`CALL VEC_ADDD (BAD, A, B, C, I, N, STATUS)`
`CALL VEC_xTOy (BAD, ARG, RESULT, IERR, NERR, STATUS)`	`CALL VEC_RTOI (BAD, P, IP, I, N, STATUS)`

The arguments are summarised below. `BAD` is a logical value specifying whether bad input arguments are to be recognised; `N` is the number of elements in the case of VAL routines; `ARG`, `ARG1` and `ARG2` are the input arguments, and `RESULT` is the result. (In the case of the VEC routines the input arguments `ARG`, `ARG1`, `ARG2` and the `RESULT` are vectorised arrays, whereas they represent single values in the cases of the VAL and NUM routines.) `IERR` is an integer output argument which identifies the first array element to generate a numerical error, `NERR` is an integer output argument which returns a count of the number of numerical errors which occur, and finally `STATUS` is the usual integer status.

Thus the subroutine call in ADAM_EXAMPLES:SQROOT.FOR could be replaced with the call below:

```
CALL VEC_SQRTR (.TRUE., NELM, %VAL(PTR1), %VAL(PTR2), IERR, NBAD, STATUS)
```

The program SQROOT.FOR assumes that the input array is of REAL type – a safe assumption as the NDF_MAP routine used type `'_REAL'` which means that the array will be mapped as `_REAL` regardless of the actual type in the NDF. An obvious improvement would be to test the actual type of the data array, map with that type, and use an appropriate subroutine to take the square root[12]. ADAM_EXAMPLES:SQROOTGEN.FOR contains these modifications.

The PRIMDAT routines are linked using the options file PRM_LINK as shown below:

```
$ ALINK prog,PRM_LINK/OPT
```

Symbolic constants.

The set of symbolic constants provided within the PRIMDAT package is made available to a program by including the file with logical name `PRM_PAR`. These constants relate to machine-specific numeric quantities – for example, the range of values which can be represented for a particular data type or the number of bytes per value used for each data type. Programs *must* use such symbolic constants rather than the numbers which they represent. For example, the largest integer which can be represented on a VAX is 2147483647 (*i.e.* $2^{31} - 1$). A program which uses this number in arithmetic checks *etc.* will not be portable to a machine with different

[12]The general problem of producing and maintaining a set of subroutines which perform the same function for different data types is addressed by the GENERIC package described in SUN/7.

arithmetic capabilities. However, software which uses the appropriate symbolic constant (called VAL__MAXI) can be ported simply by providing an appropriate version of PRM_PAR.

The complete set of symbolic constants is represented in the table below, where the final x in the name is one of R, D, I, W, UW, B or UB as indicated above. The data type of each symbolic constant matches that of the data type to which it applies, except in the cases of VAL__NBx and VAL__SZx which are, of course, integers.

Constant	**Quantity**
VAL__BADx	Bad data value
VAL__EPSx	Machine precision – minimum ϵ such that 1 is distinguishable from $(1 + \epsilon)$
VAL__MAXx	Maximum (most positive) non-bad value
VAL__MINx	Minimum (most negative) non-bad value
VAL__NBx	Number of bytes used by a value
VAL__SMLx	Smallest positive non-zero value
VAL__SZx	Number of characters needed to format value as a decimal string

15 A graphics application

Before discussing the graphics application SNXPLOT, a comparable non-ADAM program (based on the simple XYPLOT program presented in SUN/90) is considered. This program uses the Simple Graphics System (SGS) together with the NCAR/AUTOGRAPH high-level facilities and the Starlink extensions to NCAR, (SNX)[13]. In the aforementioned program, the relevant code can be represented as follows:

```
*  Read  X,Y data arrays and get number of points, NELM.
      .......
*  Open SGS, then match the AUTOGRAPH co-ord system with the current zone.
*  (Actually these two calls are usually packaged as SNX_AGOP.)
      CALL SGS_OPEN (WKSTN, ZONE, STATUS)
      CALL SNX_AGWV

*   Plot.
      CALL SNX_EZRXY (X, Y, NELM, 'X-Label', 'Y-Label', 'Title')

*   Close down SGS.
      CALL SGS_CLOSE
```

The equivalent lines in an ADAM program would be:

```
*  Open SGS, then match the AUTOGRAPH co-ord system with the current zone.
      CALL SGS_ASSOC ('DEVICE', 'WRITE', ZONE, STATUS)
      CALL SNX_AGWV

*   Plot.
      CALL SNX_EZRXY (X, Y, NELM, 'X-Label', 'Y-Label', 'Title')

*   Close down SGS.
      CALL SGS_ANNUL (ZONE, STATUS)
      CALL SGS_DEACT (STATUS)
```

The differences are in the opening and closing of SGS; to summarize:

- the SGS_OPEN is replaced by an SGS_ASSOC. Whereas SGS_OPEN opens the device indicated by the character string WKSTN, SGS_ASSOC opens the device indicated by the ADAM parameter 'DEVICE'. Both routines return an SGS zone number and STATUS. The ADAM routine has an extra character argument which specifies the access to the graphics device – one of 'READ', 'WRITE', 'UPDATE'.
- the SGS_CLOSE is replaced by SGS_ANNUL followed by SGS_DEACT. In an application which opens several devices, more than one call to SGS_ASSOC followed by SGS_ANNUL might take place. SGS_DEACT should be called only once when all the plotting is finished.

[13]Sounds daunting, however the combination of these packages enables a 'default' graph to be drawn with only a few simple subroutine calls. With a little more effort the programmer can alter the style of the plot in virtually every desirable way, *e.g.* size, labelling, tick mark appearance, histogram, logarithmic axes, *etc.*)

A full example program is contained in ADAM_EXAMPLES:SNXPLOT.FOR. The actual data arrays are obtained by mapping the AXIS(1) data and the main data array of a spectrum and passing these via pointers, but that is incidental.

Several other considerations should be borne in mind when writing graphics applications under ADAM:

Linking – the standard ADAM link, ALINK, automatically links in the ADAM version of SGS. The non-ADAM SGS link command procedure activated by `SGS_DIR:SGSLINK` must *not* be included. This can happen accidentally as it is included in several packaged link commands. For example, the standard NCAR/SNX link command is `$ LINK prog,NCAR_DIR:SNXLINK` which is a packaged version of:

```
$ LINK prog,NCAR_DIR:AGPWRITX,AGCHNLZ,SNXLIB/L,NCARLIB/L,@SGS_DIR:SGSLINK
```

The link command for SNXPLOT is deduced by omitting `SGS_DIR:SGSLINK` from the NCAR link, *i.e.*

```
$ ALINK SNXPLOT,NCAR_DIR:AGPWRITX,AGCHNLZ,SNXLIB/L,NCARLIB/L
```

Error checking. – Unlike standard SGS, SGS under ADAM uses inherited STATUS checking. A list of symbolic constants and the errors they represent is contained in the file with logical name SGS_ERR. These values can be used to perform tests such as:

```
      INCLUDE 'SGS_ERR'
*      ...
      IF (STATUS.EQ.SGS__ZONTB) THEN
*       Zone too big - take appropriate action.
```

Bad pixels – the program SNXPLOT.FOR is likely to crash if it encounters values ~ -1.7E38 as will be the case if the input data contain bad pixels. However NCAR will ignore any data points which contain the current NCAR 'null' value. This value can be set to the bad value appropriate for the data type (see Section 12) causing NCAR to ignore bad pixels. The appropriate call is as follows, (see the NCAR users' manual, available as a Starlink MUD for details).

```
      INCLUDE 'BAD_PAR'                  ! Make VAL__BADR etc. available
*     ...
      CALL AGSETF ('NULL/1.', VAL__BADR)  ! After SGS_ASSOC & before plotting
```

This modification has been made in ADAM_EXAMPLES:SNXPLOT1.FOR.

PGPLOT

An example program using PGPLOT is contained in ADAM_EXAMPLES:PGPLOT.FOR. The basic adaptation required when using PGPLOT is that after opening a device via a call to SGS_ASSOC[14], it is necessary to inquire the workstation identifier, encode this as a character string and pass this string as the FILE (device) argument to PGBEGIN. The significant portion of the code is reproduced below:

```
      INTEGER ZONE, STATUS, IWKID, NCHAR
*     ....
*   Activate SGS.
      CALL SGS_ASSOC ('DEVICE', 'WRITE', ZONE, STATUS)
      IF (STATUS.NE.SAI__OK) GOTO 999

*   Enquire the workstation identifier.
      CALL SGS_ICURW (IWKID)

*   Encode IWKID into a character string as required by PGBEGIN.
      CALL CHR_PUTI (IWKID, WKID, NCHAR)

*   Call PGBEGIN to initiate PGPLOT and open the output device.
      CALL PGBEGIN (0, WKID(1:NCHAR), 1, 1)
```

[14]Several restrictions exist on the use of PGPLOT over SGS; these are discussed in SUN/15.

```
*   Plot with PGPLOT.
*     ...

*  Finally, call PGEND to terminate things properly.
      CALL PGEND

*   Deactivate SGS
      CALL SGS_ANNUL (ZONE, STATUS)
      CALL SGS_DEACT (STATUS)
```

The link command for this program is `$ ALINK PGPLOT,PGPLOT_DIR:GRPSHR/LIB`.

16 Dealing with Extensions – using HDS routines

As described in Section 2, *extensions* can be used to store non-standard items in an NDF. A typical use of an extension would be to store information associated with a particular instrument; this information would be processed by the data-reduction package specific to that instrument. Programmers who design extensions should register the extension names with Starlink to avoid duplication. Obvious generic names such as 'ASTROMETRY' should be avoided as these are likely to be defined by Starlink in the future.

It is, of course, possible to perform *all* access to HDS structures using HDS routines (see SUN/92). The NDF routines simply provide a convenient method of accessing standard items in NDFs, but the programmer must resort to HDS routines to deal with items in extensions. However, the NDF routines do include facilities for propagating extensions, checking for the presence of extensions, creating and deleting extensions and finding HDS *locators* to specific extensions.

This last statement requires some explanation. HDS refers to items in a structure via *locators*; each locator is a `CHARACTER*15` variable which points to a HDS object. Strictly, a locator is a string of length `DAT__SZLOC` – this latter item being a symbolic constant which is defined in the `SAE_PAR` include file. Locators must be declared as `CHARACTER*(DAT__SZLOC)` as there is no guarantee that the length of 15 will be used for future HDS implementations on machines other than VAXs. The use of locators is illustrated in the example which follows.

Recalling the program ADAM_EXAMPLES:ADD7.FOR discussed in Section 6, a constant value was added to each element in an NDF main data array. NDF_ASSOC was used to associate the input file with an NDF identifier and NDF_MAP was used to map the main data array. The program below performs a similar task, but uses HDS `DAT_` routines.

```
      SUBROUTINE ADD8 (STATUS)
      IMPLICIT NONE
      INCLUDE 'SAE_PAR'
      CHARACTER*(DAT__SZLOC) ILOC, DLOC
      INTEGER NELM, STATUS, PTR

*   Associate locator with input file.
      CALL DAT_ASSOC ('INPUT', 'UPDATE', ILOC, STATUS)

*   Get locator for input.DATA_ARRAY.
      CALL DAT_FIND (ILOC, 'DATA_ARRAY', DLOC, STATUS)

*   Map input.DATA_ARRAY.
      CALL DAT_MAPV (DLOC, '_REAL', 'UPDATE', PTR, NELM, STATUS)

*   Call subroutine to add 8.0 to each array element.
      CALL ADDIT (NELM, %VAL(PTR), 8.0, STATUS)

* Tidy up
      CALL DAT_ANNUL (DLOC, STATUS)
      CALL DAT_ANNUL (ILOC, STATUS)
      END
```

Two locators ILOC and DLOC are used in this program; ILOC is associated with the top-level of the input NDF, *i.e.* with `INPUT`, and DLOC with `INPUT.DATA_ARRAY`. Both are declared

as CHARACTER*DAT__SZLOC and both are *annulled* with a call to DAT_ANNUL at the end of the program. Annulling a locator cancels the association between the locator and the object and unmaps any arrays mapped via the locator. The first call to DAT_ASSOC associates a locator ILOC with the input file. This locator effectively points to the top-level of the file. To find a locator to a first-level object, DAT_FIND is used. For example, the call above finds a locator to INPUT.DATA_ARRAY. DAT_MAPV maps the object INPUT.DATA_ARRAY as a vectorised array, just as NDF_MAP did in ADD7.

However, this program will only work if the main data array is to be found in INPUT.DATA_ARRAY – *i.e.* the NDF is primitive. If INPUT.DATA_ARRAY is a structure with the actual data array contained in INPUT.DATA_ARRAY.DATA as is also consistent with the NDF definition, then ADD8 will crash. The program could perform checks to discern the format for itself, but it is obviously easier to use the NDF routines to access standard objects.

However this option is not available when wishing to process information in extensions. The next example considered here uses information from the FIGARO extension. This extension may contain the exposure time for an observation. If this exists it will be held in .OBS.TIME in the FIGARO extension. This extract from a TRACE on ADAM_EXAMPLES:FIGDATA.SDF shows the actual location is FIGDATA.MORE.FIGARO.OBS.TIME.

```
FIGDATA  <NDF>
  ...
  MORE           <EXT>           {structure}
     FIGARO         <EXT>           {structure}
        OBS            <STRUCT>        {structure}
           TIME           <_REAL>         7.5
```

The example program ADAM_EXAMPLES:DIVTIM.FOR reads the value of the exposure time and divides the main data array by this value. The steps involved in accessing the exposure time value are described below.

The input file is associated with an NDF identifier in the usual way. The program checks for the existence of a FIGARO extension and if such an extension does exist, a locator to it is found.

```
*   Check that FIGARO extension is there.
      CALL NDF_XSTAT (NDF, 'FIGARO', EXIST, STATUS)
      IF (EXIST) THEN
*      Get locator to FIGARO extension.
         CALL NDF_XLOC (NDF, 'FIGARO', 'UPDATE', FLOC, STATUS)
```

It is now necessary to use HDS routines. The program checks to see if there is an .OBS structure in the FIGARO extension; if there is, a locator to it is found.

```
*      See if FIGARO .OBS  structure is there.
         CALL DAT_THERE (FLOC, 'OBS', EXIST, STATUS)
         IF (EXIST) THEN
*         Get locator for FIGARO .OBS structure.
            CALL DAT_FIND (FLOC, 'OBS', OLOC, STATUS)
```

Now the program checks for the existence of a .TIME object in the FIGARO .OBS structure. If this exists, a locator to it is found.

```
*         See if .OBS.TIME is there.
            CALL DAT_THERE (OLOC, 'TIME', EXIST, STATUS)
            IF (EXIST) THEN
*             Get locator for FIGARO .OBS.TIME item.
              CALL DAT_FIND (OLOC, 'TIME', TLOC, STATUS)
```

The value of the `.OBS.TIME` object is now retrieved. The locator to it can be annulled.

```
*             Get value of .OBS.TIME
              CALL DAT_GET(TLOC, '_REAL', 0, 0, TIME, STATUS)
              CALL DAT_ANNUL (TLOC, STATUS)
```

17 Running under ICL

All the example programs discussed so far have been tested under the familiar DCL (Digital Command Language). However this approach does not allow ADAM to fulfil its rôle as a *multi-tasking software environment*. This means that a number of tasks (each task is a VMS process) can be active simultaneously and can communicate with each other. This functionality can be achieved using the Interactive Command Language (ICL). A full description of ICL is given in the *ICL Users' Guide* available as a Starlink MUD. Section 18 discusses how to write the ICL equivalent of DCL command procedures.

One advantage of using ICL is speed. The following simple experiment should demonstrate this.

Set up a symbol to run one of the ADAM programs. (The example below uses REPDIM2 which reports the dimensions of the input NDF – as described in Section 6.)

```
$ REPDIM2:=="$ADAM_EXAMPLES:REPDIM2"
```

Now run the program on an NDF:

```
$ REPDIM2 IMAGE
No. of dimensions is 2
Array dimensions are 256 x 256
```

Try this a few times and note how long it takes. A significant fraction of the run-time of this small program is occupied with loading the executable code into the computer's memory. This loading takes place every time the program runs.

The session below shows how to run the same program under ICL:

```
$ ICL
 (informational messages appear)
ICL> DEFINE REPDIM2 ADAM_EXAMPLES:REPDIM2
ICL> REPDIM2 IMAGE
Loading ADAM_EXAMPLES:REPDIM2 into 012DREPDIM2
No. of dimensions is 2
Array dimensions are 256 x 256
```

This takes just as long as running the program under DCL, *but* the executable image has been loaded into a subprocess, in this case named 012DREPDIM2. This process remains active after the program execution is completed. Consequently on second or subsequent runs, the program execution begins immediately – with a large increase in speed for the user.

The process will endure throughout the ICL session unless the maximum number of allowed subprocesses is reached at which point the least recently used process will be *killed*.

For example, after loading REPDIM2 you might try loading some other programs. In the continuation of the session above, commands for ADDNEW and ADDCONST are defined and the programs run. The ICL command TASKS shows which tasks are active.

```
ICL> TASKS
     ******  Cached Tasks  ******
   Task Names             Process Names
      ADDNEW                 012DADDNEW
     ADDCONST               012DADDCONST
      REPDIM2                012DREPDIM2
```

Three[15] is the maximum number of tasks which can be simultaneously active. When another program is loaded the user is warned that the REPDIM2 process is being killed; a subsequent invocation of REPDIM2 would require it to be loaded again.

You can explicitly kill a process by typing `KILL` *process name*. All processes are usually stopped when the user leaves ICL by typing `EXIT`.

[15]Three is the default value, but this may be adjusted on a system basis.

Monoliths.

The need to kill and reload tasks can be reduced by organising a group of programs into a *monolith* – such a monolith is loaded as a single task. KAPPA and the ICL version of Figaro are arranged in this way. The disadvantage is that the first time a program from the monolith is invoked, the whole monolith must be loaded. However once the monolith is loaded any programs it contains are ready to run. In the example below, once KAPPA is loaded, all the KAPPA commands are ready to run. (Section 21 explains how to build a monolith.)

```
$ ICL
ICL> KAPPA                              ! This defines all the KAPPA command names
ICL> CREFRAME                           ! The first command causes KAPPA to be loaded
Loading KAPPA_DIR:KAPPA into 012DKAPPA
```

Words of warning.

Programs which run under DCL should give the same results under ICL. Several possible problem areas should be noted:

Initialisation of variables – the programmer who relies on VMS initialising variables to zero will get away with this carelessness when running Fortran programs under DCL. However program variable values are 'remembered' between invocations of a program under ICL, so uninitialised variables may well have non-zero values. *So don't rely on initialisation to zero!*

Case of input parameters – parameters entered on a DCL command line are automatically converted to upper case, but this does not happen on an ICL command line. This may catch out the program which is case sensitive. *So don't write case-sensitive programs if you can help it!*

Message synchronization – as discussed in Section 5, it is possible for message reports to be made on the graphics screen of a VDU when running under ICL. An example of this undesirable behaviour occurs when a program reads and reports a series of cursor positions. The messages which report the co-ordinates may be output on the graphics screen – and may overwrite previous messages. This occurs because graphical output (such as the cursor) is sent directly to the terminal and causes a switch to graphics mode; text output is buffered and may arrive after the switch to graphics has occurred. The solution is to flush the textual output buffer immediately before any graphical output (such as a cursor call). This is done with the call:

```
CALL MSG_SYNC (STATUS)
```

The program ADAM_EXAMPLES:CURSOR.FOR has this call in the necessary places; without the MSG_SYNC calls, the program exhibits the problem described above.

DCL commands.

DCL commands can be executed from ICL by prefixing the DCL command with '$' (or 'DCL') thus:

```
ICL> $ SHO TIME
```

The first time a DCL command is entered in an ICL session, a subprocess for executing DCL commands will be created. Note that although abbreviated forms of commands can be used, it is necessary to enter the complete DCL command on a line as the user cannot be prompted for unspecified command parameters. (So you cannot type `$ SHOW` and be prompted with `_What?`) Another consideration is that DCL commands are run in a separate process from the main ICL 'command' process. This has several implications. For example, changing the default directory in the DCL subprocess does not affect that associated with the command process; this remains set to the directory current when the ICL process began. The command DEFAULT can be used to set the default directory for both the command process and DCL subprocess, thus:

```
ICL> DEFAULT DISK$USER1:[JM.ADAM]
```

Similar comments apply to the allocation of devices such as tape drives to processes.

18 Writing ICL command files and procedures

Like DCL, ICL has a set of commands, functions and control structures which can be used to write powerful *command files* and *procedures*, both of which have the default extension '.ICL'. An important difference between command files and procedures is that command files run when loaded, whereas loading a procedure simply makes it available to run.

ICL syntax.

ICL syntax will generally appear familiar to the Fortran user although there are several important distinctions. Perhaps the most striking difference involves variable type; this is not fixed but depends on the current value of a variable. The example session below illustrates some of ICL's capabilities. The open bracket '{' on an ICL command line begins a comment. Explanations of the ICL commands below are shown enclosed in brackets {} but the closing bracket is included only for appearance.

```
$ ICL
ICL> x=3              {Assign 3 to variable x}
ICL> =x               {Print the value of x}
       3
ICL> x='Hi there'     {Assign string to x - ICL variables have no intrinsic type}
ICL> =x               {Print the value of x}
Hi there
ICL> x=x&x            {String concatenation achieved using '&'}
ICL> =x               {Print the value of x}
Hi thereHi there
ICL> =sqrt(4)         {ICL has many Fortran-like intrinsic functions}
2
ICL> x=upcase(x)      {And other functions too.}
ICL> =x
HI THEREHI THERE
```

ICL commands.

A selection of ICL commands were introduced in Section 17, *i.e.* `LOAD`, `TASKS`, `DEFINE`, `KILL` and `DEFAULT`. A full list of the commands and their specifications can be examined by typing `HELP` in ICL.

ICL functions.

Several ICL functions (SQRT, UPCASE) are shown in the example session above. SNAME is a very useful ICL function which concatenates a string with an integer (it is described here as it is used in the procedure PLOTS later in this section). SNAME has two or three arguments, the first and second being a string and an integer. The optional third argument gives the number of characters which the integer part of the resultant string should occupy, for example: `SNAME('FIBRE',2)` returns `FIBRE2` whereas `SNAME('FIBRE',2,3)` returns `FIBRE002`. A complete up-to-date list of functions can be viewed by typing `HELP FUNCTIONS` in ICL.

ICL control structures.

Two types of control structures are available within ICL; the loop structure, which can be compared to the Fortran DO loop, and the IF or conditional structure, which also resembles

its Fortran equivalent. Three variants of the loop structure exist; an example of each is shown below. A BREAK statement can be used to pass control to the end of a LOOP; normally such a statement would be inside an IF structure. Also shown is an example IF structure.

```
LOOP          LOOP WHILE (I<5)     LOOP FOR I=1 TO 7      IF A=0
...           ...                  ...                    ...
END LOOP      END LOOP             END LOOP               ELSE IF NOT DONE
                                                          ...
                                                          END IF
```

ICL control structures can only be used in procedures. Both a LOOP and an IF control structure are used in the procedure PLOTS shown later in this section.

ICL command files.

ICL command files contain a set of ICL command lines and are activated by typing LOAD *filename* in ICL. For example, rather than typing the commands to define ADDNEW and REPDIM2 in each ICL session, the appropriate commands (as shown below) might be written to a file called MYCOM.ICL.

```
DEFINE ADDNEW  ADAM_EXAMPLES:ADDNEW
DEFINE REPDIM2 ADAM_EXAMPLES:REPDIM2
```

On entering ICL, loading this command file is analogous to activating a DCL command file.

```
ICL> LOAD MYCOM         {Defines ADDNEW and REPDIM2}
```

Just as you probably have a LOGIN.COM file which executes every time you begin a VAX session, an ICL login file can be set up which will execute each time you enter ICL. This is done by defining the appropriate file as `ICL_LOGIN`. For example, you might include the following line in your LOGIN.COM:

```
$ DEFINE ICL_LOGIN ADAM_EXAMPLES:LOGIN.ICL
```

ICL procedures.

ICL procedures are ideal for programming data-reduction sequences. An ICL procedure can use any ICL commands (including user-defined commands) control structures, functions *etc.* As mentioned above, loading a procedure does not cause it to run; it is simply made available for running whenever the procedure name is typed. A procedure begins with the declaration `PROC` *procedurename* and ends with `END PROC`. For example, the command file MYCOM.ICL above can be converted into a procedure named MYPROC.ICL by inserting an initial line `PROC MYPROC` and a final line `END PROC`. Running this procedure requires two steps: it is first loaded, after which it is run by typing the procedure name as shown below. Subsequent runs do not require the procedure to be loaded again.

```
ICL> LOAD MYPROC
ICL> MYPROC
```

Unlike command files, ICL procedures can have arguments as shown in the example below:

```
PROC MULT X Y
  Z=X*Y
  =Z
END PROC
```

This is loaded and run as shown below:

```
ICL> LOAD MULT
ICL> MULT 15 2
   30
```

The procedure below was written to produce CANON plots from seven data files each containing a spectrum. These spectra had been extracted from a CCD image and named `FIBRE01.SDF`, `...FIBRE08.SDF`. One file, `FIBRE06.SDF` is missing from the sequence, as this corresponded to a dud fibre. The procedure uses KAPPA's LINPLOT to produce and print the graphs on a CANON laser printer (see SUN/95); it is therefore necessary to define the command LINPLOT (by typing KAPPA) before running the procedure. Additional explanation is provided in the on-line file `ADAM_EXAMPLES:PLOTS.ICL`.

```
PROC PLOTS
LOOP FOR I = 1 TO 8
  IF NOT (I = 6)
     FILENAME = (SNAME('FIBRE',I,2))
     LINPLOT INPIC=('@'&FILENAME) DEVICE=CANON_L PLTITL=(FILENAME) \
     $ PRINT/PASSALL/QUE=SYS_LASER CANON.DAT
  ENDIF
END LOOP
END PROC
```

19 Creating a help library

As mentioned in Section 7, it is possible to store help information associated with a program in a HELP library. The whereabouts of this help can be indicated in the program interface file. The procedure to provide help information for the program ADDCONST is outlined below.

The first step is to create a help file appropriate for the program. This is a text file and has the default extension `.HLP`. ADDCONST is a very simple program with only two parameters: INPUT – which is used to get the name of an input NDF, and CONST, a scalar value which is added to the NDF main data array. An appropriate help file ADAM_EXAMPLES:ADDCONST.HLP is reproduced below:

```
1 ADDCONST
Add a scalar to an NDF data structure.
Description:
  The routine adds a scalar (i.e. constant) value to each pixel of
  an NDF's data array to produce a new NDF data structure.

2 Parameters
For information on individual parameters, select from the list below:

3 INPUT
INPUT = NDF (Update)
  Input NDF data structure, to which the value is to be added.

3 CONST
Enter a scalar value.
This will be added to each element in the main data array of the NDF.
```

The structure of the text file is hierarchical; the above file contains four items, `ADDCONST`, `Parameters`, `INPUT` and `CONST`. Each item has a position in the hierarchy within the help file as indicated by the number 1, 2 or 3 at the beginning of a line. The lines of text following each item contain the help information associated with it. For example, `ADDCONST` is a first-level object containing the application name and is followed by several lines of text containing general information associated with the application. `Parameters` is a second-level object, with a single line of associated text. The individual parameters are each at the third level. In this case there are two, `INPUT` and `CONST`, each of which is followed by a number of lines of associated help information.

This help file must be inserted into a help library. The commands to create such a library – in this case called MYHELP.HLB – and insert ADDCONST.HLP are as follow:

```
$ LIB/HELP/CREATE MYHELP          ! Creates MYHELP.HLB
$ LIB/HELP MYHELP ADDCONST        ! Inserts ADDCONST.HLP into MYHELP.HLP
```

The final step is to modify the ADDCONST interface file so that the program knows where to look for the help information. This is done using the `helpkey` field. The location of help information for a particular item is indicated by specifying the help library and the position in the library hierarchy where the information is stored. For example, the help information for the parameter CONST is located in `MYHELP ADDCONST PARAMETERS CONST`. A suitably modified ADDCONST.IFL is shown below:

```
interface ADDCONST
  parameter      INPUT          # Input NDF
     position    1
     ...
     helpkey     'ADAM_EXAMPLES:MYHELP ADDCONST PARAMETERS INPUT'
  endparameter
  parameter      CONST          # Scalar value to add
     position    2
     ...
     helpkey     'ADAM_EXAMPLES:MYHELP ADDCONST PARAMETERS CONST'
  endparameter
endinterface
```

An obvious refinement suggests itself. The location of the help library appropriate for a program can be specified once in the interface file, and the helpkey associated with each parameter can merely point to the location of the help within that library. Indeed not just the library name, but a location within the help library hierarchy can be specified using the `helplib` field. Only the part specific to each parameter need be given in the helpkey field for that parameter. Thus the interface file for ADDCONST could be amended as shown below:

```
interface ADDCONST
  helplib     'ADAM_EXAMPLES:MYHELP ADDCONST PARAMETERS'
  parameter      INPUT          # Input NDF
     position    1
     ...
     helpkey     'INPUT'
  endparameter
  parameter      CONST          # Scalar value to add
      position    2
      ...
     helpkey     'CONST'
  endparameter
endinterface
```

Having done this, the help can be accessed by typing `?` in response to a prompt as shown below:

```
$ RUN ADDCONST
INPUT - Input NDF structure > ?

ADDCONST

 PARAMETERS

   INPUT

      INPUT = NDF (Update)
      Input NDF data structure, to which the value is to be added.

INPUT - Input NDF structure >
```

Typing `??` rather than `?` leaves the user in the HELP system to browse through any other available information. Pressing <CR> one or more times (according to the current level in the help library) restores the program prompt.

It is also possible to examine the help information without running the program. This is usually done under ICL using the command DEFHELP as shown below:

```
$ ICL
ICL> DEFHELP ADDCONST ADAM_EXAMPLES:MYHELP
ICL> HELP ADDCONST
     ... Help information appears
```

The directory where the help library resides must be specified in the DEFHELP command; otherwise DEFHELP will search for MYHELP.HLB in system directories.

Usually help libraries contain information on a number of programs; other help files could be prepared and inserted into MYHELP.HLB. If a program has a standard ADAM prologue, an appropriate `.HLP` file can be generated automatically; see Section 20 for details.

20 Prologues

All of the example programs discussed so far have been fairly short and to-the-point. However *real* ADAM programs have lengthy prologues which describe the program's function, parameters, arguments, history, deficiencies, authors *etc.* The main purpose of such prologues is to document the program for prospective users and make the job of maintenance easier.

ADAM prologues are highly standardised. It is worthwhile to follow the standard – not least because utilities exist to automatically produce both LaTeX documentation and help libraries from standard prologues. These utilities form part of the *Simple Software Tools* package (SST) and are briefly described later in this section. The SST package is fully documented in SUN/110.

Rather than typing in a prologue from scratch, a programmer can edit an existing one or use the STARLSE editor described in SUN/105. An example ADAM prologue is reproduced on the opposite page and the prologue of the accompanying interface file is reproduced below. The complete files are contained in ADAM_EXAMPLES:CADD.FOR and CADD.IFL.

```
#+
#  Name:
#     CADD.IFL

#  Type of module:
#     ADAM A-task parameter interface.

#  Author:
#     RFWS: R.F. Warren-Smith (STARLINK)
#     {enter_new_authors_here}

#  History:
#     11-APR-1990 (RFWS):
#         Original version.
#     {enter_changes_here}

#-
```

The SST utilities used to produce documentation and help modules can be summarised as follow:

PROLAT – for producing LaTeX documentation. The PROLAT utility processes a file (or files) containing a prologue of the correct form, to produce a `.TEX` file (called `PROLAT.TEX` by default). This file can then be processed in the normal way (see SUN/12). The sequence of commands below illustrate the procedure for processing the file `CADD.FOR`.

```
$ SST                          ! Make SST available
$ PROLAT CADD.FOR              ! Process CADD.FOR to produce PROLAT.TEX
$ LATEX PROLAT                 ! Usual sequence to LaTeX and print
$ DVICAN PROLAT
$ PRCN PROLAT.DVI-CAN
```

PROHLP – for producing help libraries. Just as PROLAT produces a `.TEX` file, PROHLP produces a `.HLP` text file (called `PROHLP.HLP` by default) which can be inserted into a HELP library as shown below. (See Section 19 for more information on HELP libraries.)

```
$ SST                                   ! Make SST available
$ PROHLP CADD.FOR                       ! Process CADD.FOR to produce PROHLP.HLP
$ LIBRARY/HELP MYHELP.HLB PROHLP.HLP    ! Insert HLP file into MYHELP.HLB
```

```
      SUBROUTINE CADD( STATUS )
*+
*  Name:
*     CADD

*  Purpose:
*     Add a scalar to an NDF data structure.

*  Language:
*     Starlink Fortran 77

*  Type of Module:
*     ADAM A-task

*  Invocation:
*     CALL CADD( STATUS )

*  Description:
*     The routine adds a scalar (i.e. constant) value to each pixel of
*     an NDF's data array to produce a new NDF data structure.

*  ADAM Parameters:
*     IN = NDF (Read)
*        Input NDF data structure, to which the value is to be added.
*     OUT = NDF (Write)
*        Output NDF data structure.
*     SCALAR = _DOUBLE (Read)
*        The value to be added to the NDF's data array.
*     TITLE = LITERAL (Read)
*        Value for the title of the output NDF. A null value will cause
*        the title of the NDF supplied for parameter IN to be used
*        instead. ['KAPPA - Cadd']

*  Notes:
*     -  This routine correctly processes the AXIS, DATA, QUALITY,
*     LABEL, TITLE, UNITS and VARIANCE components of an NDF data
*     structure and propagates all extensions. Bad pixels and all
*     non-complex numeric data types can be handled. The HISTORY
*     component is simply propagated without change, if present.

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

*  Authors:
*     RFWS: R.F. Warren-Smith (STARLINK)
*     {enter_new_authors_here}

*  History:
*     11-APR-1990 (RFWS):
*        Original version.
*     {enter_changes_here}

*  Bugs:
```

```
*     {note_any_bugs_here}

*-
```

21 Building a monolith

The procedure used to create a monolith containing the ADDCONST and REPDIM2 programs is shown below. (Of course, monoliths usually comprise many more than two programs.) All the files described are in ADAM_EXAMPLES.

Firstly, a library is created to contain the object code of all the programs intended to comprise the monolith. In this example the library is called MIXLIB.OLB.[16]

```
$ LIB/CREATE MIXLIB           ! Creates MIXLIB.OLB
$ LIB MIXLIB ADDNEW,REPDIM2   ! Puts ADDNEW.OBJ & REPDIM2.OBJ into MIXLIB.OLB
```

A master program which calls the program corresponding to the command entered must now be written. MIXTURE.FOR below will call ADDNEW or REPDIM2 as appropriate.

```
      SUBROUTINE MIXTURE (NAME, STATUS)
      IMPLICIT NONE
      INCLUDE 'SAE_PAR'
      CHARACTER*(*) NAME
      INTEGER STATUS
      IF (STATUS.NE.SAI__OK) RETURN
      IF (NAME.EQ.'ADDNEW') THEN
         CALL ADDNEW (STATUS)
      ELSEIF (NAME.EQ.'REPDIM2') THEN
         CALL REPDIM2 (STATUS)
      ENDIF
      END
```

This program must now be compiled and linked. The special MLINK command is used to link monoliths. The object code library, MIXLIB, and anything else needed to link the constituent programs (*e.g.* graphics libraries) should also be included in the link.

```
$ FOR MIXTURE
$ MLINK MIXTURE,MIXLIB/LIB
```

A monolith interface file containing the interface files for each of the constituent programs must now be created. MIXTURE.IFL is shown below. Note that the file begins with the line "`monolith` *monolith-name*" and ends with "`endmonolith`". All the necessary interface files are simply included in between.

```
monolith mixture
  interface addconst
   ...
  interface repdim2
   ...
endmonolith
```

MIXTURE.ICL, an ICL command file to define the commands in the monolith should now be written. Each command points to the monolith which contains the program *i.e.* MIXTURE.EXE.

[16] Of course this step is not really necessary; you could simply list all the .OBJ files individually at the link stage.

```
define addnew  mixture
define repdim2 mixture
```

To try the monolith, simply enter ICL and load the procedure MIXTURE.ICL to define the commands. The first command which specifies a program in the monolith will cause the monolith to be loaded.

```
$ ICL
ICL> LOAD MIXTURE     ! Defines commands
ICL> REPDIM2          ! Loads MIXTURE monolith
```

22 Miscellaneous ADAM packages

The following packages may also be of interest:

AGI – Applications Graphics Interface. AGI is a graphics-database system which is used to retain information associated with a plot after the program creating the plot has finished. This information can be recalled by subsequent programs. The information stored includes: the plotting device used, the position and extent of the plot on the device, the co-ordinate system and a user-supplied picture name. A typical use of the system is as follows: a program draws a graph and the information described above is stored. A second program which invokes a cursor can then retrieve the coordinate system used by the first program and can thus be used to measure positions on the plot. This set of routines is linked as shown below. See SUN/48 for a full description.

```
$ ALINK prog,AGI_DIR:AGILINK/OPT
```

IDI – Image Display Interface. IDI provides a device-independent way of writing programs to perform image display. GKS provides limited image display facilities, *i.e.* an image can be displayed and its look-up table changed. Unlike GKS, IDI allows an image to be zoomed and panned, and a 'snapshot' can be taken of an image, enabling a hardcopy to be made (see KAPPA SNAPSHOT application). IDI routines are linked as shown below; see SUN/65 for details.

```
$ ALINK prog,IDI_DIR:IDILINK/OPT
```

SLALIB. This is a collection of subroutines and functions most of which are concerned with astronomical position and time. If you want a routine to find the approximate heliocentric position and velocity of the Earth on a particular date, SLALIB is the place to look. There are also more general mathematical routines which perform matrix operations, random number generation, trigonometrical functions *etc.* See SUN/67 for a full description. SLALIB routines are linked as shown below:

```
$ ALINK prog,SLALIB_DIR:SLALIB/LIB
```

Magnetic tape handling. The MAG_ package provides facilities for positioning, reading, and writing magnetic tapes. To link an ADAM program with the MAG library, it is necessary to include the options file ADAM_LIB:MAGLINK/OPT in the link command as shown below. See APN/1 for a full description.

```
$ ALINK task,ADAM_LIB:MAGLINK/OPT
```

A Standard components in an NDF

An NDF comprises a main data array plus a collection of objects drawn from a set of standard items and extensions (see SGP/38). Only the main data array must be present; all the other items are optional.

ADAM_EXAMPLES:EXAMPLE.SDF is an NDF which contains all the standard NDF components and also has a Figaro extension. The structure of the file (as revealed by `$ TRACE ADAM_EXAMPLES:EXAMPLE`) is shown below.

```
EXAMPLE  <NDF>

   DATA_ARRAY(856)  <_REAL>       -1.7014117E38,0.2284551,-2.040089,
                                  ... 820.8976,570.0729,-1.7014117E38,449.574
   TITLE          <_CHAR*30>      'HR6259 - AAT fibre data'
   LABEL          <_CHAR*20>      'Flux'
   UNITS          <_CHAR*20>      'Counts/s'
   QUALITY        <QUALITY>       {structure}
      BADBITS        <_UBYTE>        1
      QUALITY(856)   <_UBYTE>        1,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,
                                     ... 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0

   VARIANCE(856)  <_REAL>         2.1,0.1713413,1.5301,34.38378,42.35531,
                                  ... 615.6732,427.5547,353.9127,337.1805
   AXIS(1)        <AXIS>          {structure}

   Contents of AXIS(1)
      DATA_ARRAY(856)  <_REAL>       3847.142,3847.672,3848.201,3848.731,
                                     ... 4298.309,4298.838,4299.368,4299.897
      LABEL          <_CHAR*20>      'Wavelength'
      UNITS          <_CHAR*20>      'Angstroms'

   HISTORY        <HISTORY>       {structure}
      CREATED        <_CHAR*30>      '1990-DEC-12 08:21:02.324'
      CURRENT_RECORD  <_INTEGER>     3
      RECORDS(10)    <HIST_REC>      {array of structures}

      Contents of RECORDS(1)
         TEXT           <_CHAR*40>      'Extracted spectrum from fibre data.'
         DATE           <_CHAR*25>      '1990-DEC-19 08:43:03.08'
         COMMAND        <_CHAR*30>      'FIGARO V2.4 FINDSP command'


   MORE           <EXT>           {structure}
      FIGARO         <EXT>           {structure}
         TIME           <_REAL>         1275
         SECZ           <_REAL>         2.13

End of Trace.
```

Of course, this is only an example format. There are various ways of representing some of the components. These *variants* are described in SGP/38.

The components are considered in detail below. The names (in bold typeface) are significant as they are used by the NDF access routines to identify the components.

DATA – the main data array is the only component which must be present in an NDF. In the case of EXAMPLE.SDF, the data component is a 1-d array of real type with 856 elements.

TITLE – the character string `'HR6259 - AAT fibre data'` describes the contents of the NDF. The NDF's TITLE might be used as the title of a graph *etc.*

LABEL – the character string `'Flux'` describes the quantity represented in the NDF's main data array. The LABEL is intended for use on the axis of graphs *etc.*

UNITS – this character string describes the physical units of the quantity stored in the main data array, in this case, `'Counts/s'`.

QUALITY – this component is used to indicate the quality of each element in the main data array. The quality structure contains a quality array and a BADBITS value, both of which *must* be of type _UBYTE. The quality array has the same shape and size as the main data array and is used in conjunction with the BADBITS value to decide the quality of a pixel in the main data array. In EXAMPLE.SDF the BADBITS component has value 1. This means that a value of 1 in the quality array indicates a bad pixel in the main data array, whereas any other value indicates that the associated pixel is good.

VARIANCE – the variance array is the same shape and size as the main data array and contains the errors associated with the individual data values. These are stored as *variance* estimates for each pixel.

AXIS – the AXIS structure may contain axis information for any dimension of the NDF's main array. In this case, the main data array is only 1-d, therefore only the AXIS(1) structure is present. This structure contains the actual axis data array, and also label and units information.

HISTORY – the history component provides a record of the processing history of the NDF. Only the first of three records is shown for EXAMPLE.SDF. This indicates that the spectrum was extracted from fibre data using the Figaro FINDSP command on 19th December 1990. Support for the history component is not yet provided by the NDF access routines.

EXTENSIONs – the purpose of extensions is to store non-standard items. EXAMPLE.SDF began life as a Figaro file[17] which contained values for the airmass and exposure time associated with the observations. These are stored in the Figaro extension, and the intention is that the Figaro applications which use these values will know where to find them.

[17]The Figaro file was converted to an NDF using the command DST2NDF, see SUN/55.

B NDF routine summary

NDF_ACGET (INDF, COMP, IAXIS, VALUE, STATUS) – Obtain the value of an NDF axis character component

NDF_ACLEN (INDF, COMP, IAXIS, LENGTH, STATUS) – Determine the length of an NDF axis character component

NDF_ACMSG (TOKEN, INDF, COMP, IAXIS, STATUS) – Assign the value of an NDF axis character component to a message token

NDF_ACPUT (VALUE, INDF, COMP, IAXIS, STATUS) – Assign a value to an NDF axis character component

NDF_ACRE (INDF, STATUS) – Ensure that an axis coordinate system exists for an NDF

NDF_AFORM (INDF, COMP, IAXIS, FORM, STATUS) – Obtain the storage form of an NDF axis array

NDF_AMAP (INDF, COMP, IAXIS, TYPE, MMOD, PNTR, EL, STATUS) – Obtain mapped access to an NDF axis array

NDF_ANNUL (INDF, STATUS) – Annul an NDF identifier

NDF_ANORM (INDF, IAXIS, NORM, STATUS) – Obtain the logical value of an NDF axis normalisation flag

NDF_AREST (INDF, COMP, IAXIS, STATUS) – Reset an NDF axis component to an undefined state

NDF_ASNRM (NORM, INDF, IAXIS, STATUS) – Set a new value for an NDF axis normalisation flag

NDF_ASSOC (PARAM, MODE, INDF, STATUS) – Associate an existing NDF with an ADAM parameter

NDF_ASTAT (INDF, COMP, IAXIS, STATE, STATUS) – Determine the state of an NDF axis component (defined or undefined)

NDF_ASTYP (TYPE, INDF, COMP, IAXIS, STATUS) – Set a new numeric type for an NDF axis array

NDF_ATYPE (INDF, COMP, IAXIS, TYPE, STATUS) – Obtain the numeric type of an NDF axis array

NDF_AUNMP (INDF, COMP, IAXIS, STATUS) – Unmap an NDF axis array component

NDF_BAD (INDF, COMP, CHECK, BAD, STATUS) – Determine if an NDF array component may contain bad pixels

NDF_BASE (INDF1, INDF2, STATUS) – Obtain an identifier for a base NDF

NDF_BB (INDF, BADBIT, STATUS) – Obtain the bad-bits mask value for the quality component of an NDF

NDF_BEGIN – Begin a new NDF context

NDF_BOUND (INDF, NDIMX, LBND, UBND, NDIM, STATUS) – Enquire the pixel-index bounds of an NDF

NDF_CGET (INDF, COMP, VALUE, STATUS) – Obtain the value of an NDF character component

NDF_CINP (PARAM, INDF, COMP, STATUS) – Obtain an NDF character component value via the ADAM parameter system

NDF_CLEN (INDF, COMP, LENGTH, STATUS) – Determine the length of an NDF character component

NDF_CLONE (INDF1, INDF2, STATUS) – Clone an NDF identifier

NDF_CMPLX (INDF, COMP, CMPLX, STATUS) – Determine whether an NDF array component holds complex values

NDF_CMSG (TOKEN, INDF, COMP, STATUS) – Assign the value of an NDF character component to a message token

NDF_COPY (INDF1, PLACE, INDF2, STATUS) – Copy an NDF to a new location

NDF_CPUT (VALUE, INDF, COMP, STATUS) – Assign a value to an NDF character component

NDF_CREAT (PARAM, FTYPE, NDIM, LBND, UBND, INDF, STATUS) – Create a new simple NDF via the ADAM parameter system

NDF_CREP (PARAM, FTYPE, NDIM, UBND, INDF, STATUS) – Create a new primitive NDF via the ADAM parameter system

NDF_DELET (INDF, STATUS) – Delete an NDF

NDF_DIM (INDF, NDIMX, DIM, NDIM, STATUS) – Enquire the dimension sizes of an NDF

NDF_END (STATUS) – End the current NDF context

NDF_EXIST (PARAM, MODE, INDF, STATUS) – See if an existing NDF is associated with an ADAM parameter.

NDF_FIND (LOC, NAME, INDF, STATUS) – Find an NDF in an HDS structure and import it into the NDF_ system

NDF_FORM (INDF, COMP, FORM, STATUS) – Obtain the storage form of an NDF array component

NDF_FTYPE (INDF, COMP, FTYPE, STATUS) – Obtain the full data type of an NDF array component

NDF_IMPRT (LOC, INDF, STATUS) – Import an NDF into the NDF_ system from HDS

NDF_ISACC (INDF, ACCESS, ISACC, STATUS) – Determine whether a specified type of NDF access is available

NDF_ISBAS (INDF, ISBAS, STATUS) – Enquire if an NDF is a base NDF

NDF_ISTMP (INDF, ISTMP, STATUS) – Determine if an NDF is temporary

NDF_MAP (INDF, COMP, TYPE, MMOD, PNTR, EL, STATUS) – Obtain mapped access to an array component of an NDF

NDF_MAPQL (INDF, PNTR, EL, BAD, STATUS) – Map the quality component of an NDF as an array of logical values

NDF_MAPZ (INDF, COMP, TYPE, MMOD, RPNTR, IPNTR, EL, STATUS) – Obtain complex mapped access to an array component of an NDF

NDF_MBAD (BADOK, INDF1, INDF2, COMP, CHECK, BAD, STATUS) – Merge the bad-pixel flags of the array components of a pair of NDFs

NDF_MBADN (BADOK, N, NDFS, COMP, CHECK, BAD, STATUS) – Merge the bad-pixel flags of the array components of a number of NDFs

NDF_MBND (OPTION, INDF1, INDF2, STATUS) – Match the pixel-index bounds of a pair of NDFs

NDF_MBNDN (OPTION, N, NDFS, STATUS) – Match the pixel-index bounds of a number of NDFs

NDF_MSG (TOKEN, INDF) – Assign the name of an NDF to a message token

NDF_MTYPE (TYPLST, INDF1, INDF2, COMP, ITYPE, DTYPE, STATUS) – Match the types of the array components of a pair of NDFs

NDF_MTYPN (TYPLST, N, NDFS, COMP, ITYPE, DTYPE, STATUS) – Match the types of the array components of a number of NDFs

NDF_NEW (FTYPE, NDIM, LBND, UBND, PLACE, INDF, STATUS) – Create a new simple NDF

NDF_NEWP (FTYPE, NDIM, UBND, PLACE, INDF, STATUS) – Create a new primitive NDF

NDF_NOACC (ACCESS, INDF, STATUS) – Disable a specified type of access to an NDF

NDF_PLACE (LOC, NAME, PLACE, STATUS) – Obtain an NDF placeholder

NDF_PROP (INDF1, CLIST, PARAM, INDF2, STATUS) – Propagate NDF information to create a new NDF via the ADAM parameter system

NDF_QMASK (QUAL, BADBIT) – Combine an NDF quality value with a bad-bits mask to give a logical result

NDF_QMF (INDF, QMF, STATUS) – Obtain the value of an NDF's quality masking flag

NDF_RESET (INDF, COMP, STATUS) – Reset an NDF component to an undefined state

NDF_SAME (INDF1, INDF2, SAME, ISECT, STATUS) – Enquire if two NDFs are part of the same base NDF

NDF_SBAD (BAD, INDF, COMP, STATUS) – Set the bad-pixel flag for an NDF array component

NDF_SBB (BADBIT, INDF, STATUS) – Set a bad-bits mask value for the quality component of an NDF

NDF_SBND (NDIM, LBND, UBND, INDF, STATUS) – Set new pixel-index bounds for an NDF

NDF_SECT (INDF1, NDIM, LBND, UBND, INDF2, STATUS) – Create an NDF section

NDF_SHIFT (NSHIFT, SHIFT, INDF, STATUS) – Apply pixel-index shifts to an NDF

NDF_SIZE (INDF, NPIX, STATUS) – Determine the size of an NDF

NDF_SQMF (QMF, INDF, STATUS) – Set a new logical value for an NDF's quality masking flag

NDF_SSARY (IARY1, INDF, IARY2, STATUS) – Create an array section, using an NDF section as a template

NDF_STATE (INDF, COMP, STATE, STATUS) – Determine the state of an NDF component (defined or undefined)

NDF_STYPE (FTYPE, INDF, COMP, STATUS) – Set a new type for an NDF array component

NDF_TEMP (PLACE, STATUS) – Obtain a placeholder for a temporary NDF

NDF_TRACE (NEWFLG, OLDFLG) – Set the internal NDF_ system error-tracing flag

NDF_TYPE (INDF, COMP, TYPE, STATUS) – Obtain the numeric data type of an NDF array component

NDF_UNMAP (INDF, COMP, STATUS) – Unmap an NDF or a mapped NDF array

NDF_VALID (INDF, VALID, STATUS) – Determine whether an NDF identifier is valid

NDF_XDEL (INDF, XNAME, STATUS) – Delete a specified NDF extension

NDF_XGT0x (INDF, XNAME, CMPT, VALUE, STATUS) – Read a scalar value from a component within a named NDF extension

NDF_XLOC (INDF, XNAME, MODE, LOC, STATUS) – Obtain access to a named NDF extension via an HDS locator

NDF_XNAME (INDF, N, XNAME, STATUS) – Obtain the name of the N'th extension in an NDF

NDF_XNEW (INDF, XNAME, TYPE, NDIM, DIM, LOC, STATUS) – Create a new extension in an NDF

NDF_XNUMB (INDF, NEXTN, STATUS) – Determine the number of extensions in an NDF

NDF_XPT0x (VALUE, INDF, XNAME, CMPT, STATUS) – Write a scalar value to a component within a named NDF extension

NDF_XSTAT (INDF, XNAME, THERE, STATUS) – Determine if a named NDF extension exists

C HDS data types

HDS recognises a selection of *primitive data types* which correspond to Fortran data types but have names prefixed by an underscore. The correspondence between Fortran types and HDS data types is as follows:

HDS Type	**VAX FORTRAN Type**
_INTEGER	INTEGER
_REAL	REAL
_DOUBLE	DOUBLE PRECISION
_LOGICAL	LOGICAL
_CHAR[*n]	CHARACTER*n
_UBYTE	BYTE
_BYTE	BYTE
_UWORD	INTEGER*2
_WORD	INTEGER*2

For example, a variable declared as REAL in a program has HDS type `'_REAL'`. It is necessary to appreciate that if the data type is an argument in a routine then that argument should be '_REAL' rather than 'REAL'. For example:

```
CALL NDF_MAP (NDF, 'Data', '_REAL', 'UPDATE', PTR, NELM, STATUS)
```

N.B. HDS structures also have a type, although this is purely descriptive. For example, the type of an axis structure in an NDF is `AXIS`*. The only restriction on the names of structure types is that they must not begin with an underscore (to distinguish them from primitive data types).*

D PAR routines

The calling sequences for the ADAM parameter system routines are reproduced below. SG/4, Section 8 contains an introduction to the parameter system. A full description will shortly be available, that is, SUN/114, (in preparation).

`PAR_CANCL (PARAM, STATUS)` – cancel a parameter. The named parameter is cancelled and any storage associated with it is released. A subsequent attempt to get a value for the parameter will result in a new value being obtained by the underlying parameter system.

`PAR_DEF0x(PAR, VALUE, STATUS)` – set scalar dynamic default parameter value. This routine sets a scalar as the dynamic default value for a parameter. The dynamic default may be used as the parameter value by means of appropriate specifications in the interface file.

`PAR_DEF1x (PARAM, NVAL VALUES, STATUS)` – set a 1-D array of values as the dynamic default for a parameter. This routine sets a 1-D array of values as the dynamic default for a parameter of primitive type. The dynamic default may be used as the parameter value by means of appropriate specifications in the interface file.

`PAR_DEFNx (PARAM, NDIM, MAXD, VALUES, ACTD, STATUS)` – set an array of values as the dynamic default for a parameter. This routine sets an array of values as the dynamic default for a parameter of primitive type. The dynamic default may be used as the parameter value by means of appropriate specifications in the interface file.

`PAR_GET0x (PARAM, VALUE, STATUS)` – obtain a scalar parameter value. This routine obtains a primitive scalar parameter value.

`PAR_GET1x (PARAM, MAXVAL, VALUES, ACTVAL, STATUS)` – read vector parameter values. This routine obtains a primitive vector parameter value.

`PAR_GETNx (PARAM, NDIM, MAXD, VALUES, ACTD, STATUS)` – obtain an array parameter value. This routine obtains a primitive array parameter value.

`PAR_GETVx (PARAM, MAXVAL, VALUES, ACTVAL, STATUS)` – read parameter values as if object were a vector. This routine reads the values from a primitive parameter storage object as if it were vectorized (*i.e.* regardless of its dimensionality).

`PAR_PROMT (PARAM, PROMPT, STATUS)` – set a new prompt string for a parameter. Replace the prompt string for the indicated parameter by the given string.

`PAR_PUT0x (PARAM, VALUE, STATUS)` – write a scalar parameter value. This routine puts a primitive scalar value into the storage object for the named parameter.

`PAR_PUT1x (PAR, NVAL, VALUES, STATUS)` – write vector parameter values. This routine puts a 1-D array of primitive values into the storage object for the named parameter.

`PAR_PUTNx (PARAM, NDIM, MAXD, VALUES, ACTD, STATUS)` – write array parameter values. This routine puts an *n*-dimensional array of primitive values into the storage object for the named parameter.

E Character handling routines

The following are subroutines unless specifically indicated as functions.

Decoding Routines

`CHR_CTOD(`*string, dvalue, status*`)`	Read a double precision number from a character string.
`CHR_CTOI(`*string, ivalue, status*`)`	Read an integer number from a character string.
`CHR_CTOL(`*string, lvalue, status*`)`	Read a logical value from a character string.
`CHR_CTOR(`*string, rvalue, status*`)`	Read a real number from a character string.
`CHR_DCWRD(`*string, mxw, nwrd, start, stop,words, lstat*`)`	Returns all the words in a string.
`CHR_HTOI(`*string, ivalue, status*`)`	Read an integer from a hex string.
`CHR_OTOI(`*string, ivalue, status*`)`	Read an integer from an octal string.

Encoding and Formatting Routines

`CHR_CTOC(`*value, cvalue, nchar*`)`	Write a character value into a string.
`CHR_DTOC(`*dvalue, cvalue, nchar*`)`	Encode a double precision value as a string.
`CHR_ITOC(`*ivalue, cvalue, nchar*`)`	Encode an integer value as a string.
`CHR_LTOC(`*lvalue, cvalue, nchar*`)`	Encode a logical value as a string.
`CHR_RTOC(`*rvalue, cvalue, nchar*`)`	Encode a real value as a string.
`CHR_PUTC(`*cvalue, string, length*`)`	Copy one string into another at given position.
`CHR_PUTD(`*dvalue, string, length*`)`	Put double precision value into string at given position.
`CHR_PUTI(`*ivalue, string, length*`)`	Put integer value into string at given position.
`CHR_PUTL(`*lvalue, string, length*`)`	Put logical value into string at given position.
`CHR_PUTR(`*rvalue, string, length*`)`	Put real value into string at given position.
`CHR_RTOAN(`*rvalue, units, string, length*`)`	Write a real into character string as hr/deg:min:sec.

Enquiry Routines

`CHR_DELIM(`*string, delim, index1, index2*`)`	Locate substring with given delimiter character.
`CHR_EQUAL(`*str1, str2*`)`	Determine whether two strings are equal. (Logical function)
`CHR_FANDL(`*string, index1, index2*`)`	Find the indices of the first and last non-blank characters.
`CHR_FIWE(`*string, index, status*`)`	Find next end of word.
`CHR_FIWS(`*string, index, status*`)`	Find start of next word.
`CHR_INDEX(`*string, substr*`)`	Find the index of a substring in a string. (Integer function)
`CHR_INSET(`*set, string*`)`	Determine whether a string is a member of a set. (Logical function)
`CHR_ISALF(`*char*`)`	Determine whether a character is alphabetic. (Logical function)
`CHR_ISALM(`*char*`)`	Determine whether a character is alphanumeric. (Logical function)
`CHR_ISDIG(`*char*`)`	Determine whether a character is a digit. (Logical function)
`CHR_ISNAM(`*string*`)`	Determine whether a string is a valid name. (Logical function)
`CHR_LEN(`*string*`)`	Find used length of string. (Integer function)
`CHR_SIMLR(`*str1, str2*`)`	Determine if two strings are equal apart from case. (Logical function)
`CHR_SIZE(`*string*`)`	Find the declared size of string. (Integer function)

String Manipulation Routines

`CHR_APPND(`*str1, str2, len2*`)`	Copy one string into another – ignoring trailing blanks.
`CHR_CLEAN(`*string*`)`	Remove all non-printable `ASCII` characters from a string.
`CHR_COPY(`*instr, flag, outstr, lstat*`)`	Copy one string to another, checking for truncation.
`CHR_FILL(`*char, string*`)`	Fill a string with a given character.

CHR_LCASE(*string*)	Convert a string to lower case.
CHR_LDBLK(*string*)	Remove leading blanks from a string.
CHR_LOWER(*achar*)	Give lower case equivalent of a character. (Character function)
CHR_MOVE(*str1*, *str2*)	Move one string into another – ignoring trailing blanks.
CHR_RMBLK(*string*)	Remove all blanks from a string in situ.
CHR_SWAP(*c1*, *c2*)	Swap two single-character variables.
CHR_TERM(*length*, *string*)	Terminate string by padding out with blanks.
CHR_TRUNC(*delim*, *string*)	Truncate string rightwards from a given delimiter.
CHR_UCASE(*string*)	Convert a string to upper case.
CHR_UPPER(*achar*)	Give upper case equivalent of a character. (Character function)