DAF REQUIRED READING

Navigation Ancillary Information Facility

Prepared by
H. A. Neilan

NAIF Document No. 167.03
Last Modified: 13 JUL 1994

Table of Contents

Double Precision Array Files (DAF)

Purpose

This document describes the Double Precision Array File, DAF, an architecture for files that store arrays of double precision numbers. The SPICE SP-kernel and C-kernel files use the DAF architecture and associated software. The Required Reading files for these two SPICE kernels refer to this document for details.

Revisions

July 12, 1994

The document differs from the previous version of September 1991 in that it includes the addition of new routines that should be used instead of existing SPICELIB routines. DAFONW replaces DAFOPN, DAFTB replaces DAFA2B and DAFT2B, and DAFBT replaces DAFB2A and DAFB2T. Also, the term to describe the non-binary DAF file, text, has been replaced with a more accurate term, transfer, indicating that the files are written in a format suitable for transfer from one platform to another.

Intended Audience

This document is intended for users that require detailed knowledge of DAF-based file formats, such as the SP- and C-kernel formats already mentioned. It is also intended for users of the SPICELIB library who wish to create their own DAF-based file formats.

Most users of DAF-based formats will not generally need to understand the material presented in this document. For example, users of NAIF SP- and C-kernel files who wish to read state vectors and pointing angles from those files will normally do so using only subroutines and programs designed specifically for those formats. These subroutines are documented in the NAIF SPK and CK Required Reading files.

Related Documents

The following NAIF documents contain material that is closely related to the subject of this document.

1. SPK Required Reading

   This document describes how the DAF structure is used in making the SPICE SP-kernel (SPK) file containing trajectory and ephemeris data.

2. CK Required Reading

   This document describes how the DAF structure is used in making the SPICE C- kernel (CK) file containing spacecraft or instrument orientation (pointing).

3. COMMNT User's Guide

   This document describes how labels, descriptive text and other comments may be added to or removed from a DAF-based SPK or CK file.

Introduction

DAF—which stands for 'Double precision Array File'—is a file architecture that provides the advantages of arrays and direct access files without incurring the disadvantages of either one.

This architecture is supported by a set of Fortran-77 subroutines, part of the NAIF Toolkit software library, SPICELIB.

DAF is called an architecture instead of a format because it includes an extensible family of file formats, each of which is characterized by a pair of parameters. A file that conforms to any one of these formats is called an 'array file'.

An array file can contain any number of double precision arrays. Each of these arrays can contain an arbitrary number of elements. Because DAF files are intended to be portable, the DAF design requires that the array elements must be 'pure' double precision numbers. That is, they may not contain equivalenced or encoded integer or character values.

The DAF subroutines in SPICELIB support the following operations:

1. Create, open, or close an array file.

2. Add a new array to a file.

3. Locate an array within a file, either by index or by using descriptive information about the array.

4. Access—that is, retrieve or update—any contiguous set of elements in an array.
5. Convert a binary (direct access) array file to an equivalent SPICE transfer file, suitable for transfer to an environment using a form of binary representation different from that of the source CPU.
6. Convert an array file in SPICE transfer format into an equivalent binary file.

The last two functions make array files portable to any environment that supports ANSI Standard Fortran-77.

DAF-based Formats

The DAF architevture has been designed with the intention that each array contained in a particular DAF possess a 'descriptive summary' of itself. The information making up the summary and the organization of the summary should be the same for each array in the DAF. The descriptive summary is composed of double precision and integer components. The number of double precision components, ND, and the number of integer components, NI, making up the the array summaries, determine a particular format within the DAF architecture.

Values for ND and NI are fixed at the time an array file is created. Any two array files that have the same values for ND and NI can be thought of as having the same 'format'. (This does not guarantee that the arrays in the files contain the same kinds of information, only that they could be stored in the same file.) The values selected for ND and NI must satisfy the following inequalities:

```
     (NI + 1)                    (Note that this is
ND + -------- <= 125              integer division.
        2                         That is, (NI + 1)/2
                                  is rounded down to
                                  the nearest integer.)

0 <= ND <= 124

2 <= NI <= 250
```

Each array stored in an array file is 'described', in part, by ND double precision numbers and NI integer numbers, which are stored separately from the array. Most of the details of this 'description'—how many numbers are needed, and what they contain—are left to the designer of a specific DAF format.

The double precision numbers could include limits (the smallest and largest values in an array), a range of epochs throughout which the elements may be used, or statistics (the mean, median, and standard deviation of the elements).

The integer numbers could include contextual information (case number, identification codes for related objects or arrays) and conditional information (flags to indicate whether the array is unsorted, sorted by increasing or decreasing magnitude, or marked for deletion). Some integer numbers are used to keep track of the location of the array within the file.

Each array in an array file is is further described by NC characters of alphanumeric information. Examples of this alphanumeric information are producer names, archive codes, historical information, or anything else that is not easily encoded as double precision or integer numbers.

NC is a function of ND and NI. The relationship between NC and user-specified ND and NI was chosen to allowing a reasonable amount of space for storing the alphanumeric information for an array. NC is defined below:

```
                        (NI + 1)
NC  =   8 * ( ND + -------- )     (Note that this is
                           2         integer division.)
```

The double precision and integer numbers that describe each array are 'packed', or equivalenced, into an auxiliary double precision array before they are stored in the file. This auxiliary array is called the 'summary' of the associated array. The individual (unpacked) numbers are called the 'components' of the summary.

(The first ND elements of the summary contain the double precision components of the summary. Each of the remaining elements contains a pair of integer components. If NI is odd, the final element of the summary contains a single integer component.)

The NC alphanumeric characters that further describe each array are stored in a single character string, called the 'name' of the array.

Array Addresses

The location of each array in an array file is defined by a pair of numbers, called the 'initial address' and 'final address' of the array.

The term 'address' refers to a particular way of looking at an array file. Every array file is actually a standard Fortran-77 direct access file, with a particular record length. (Every array file has the same record length.) Each record is capable of storing up to 128 double precision numbers.

It is convenient, however, to think of an array file as a numbered collection of slots named 'words'. Each word is large enough to hold one double precision number. Words 1 through 128 are located in the first record of the file; words 129 through 256 are located in the second record; and so on. The number of each word is called the 'address' of the word within the file.

Any pair of addresses defines a contiguous set of words, which may fall within a single physical record or span a number of records. The elements of each array in an array file are stored in just such a set. The address of the first array element is the 'initial address' of the array. The address of the final array element is the 'final address' of the array.

The initial and final addresses of an array are always the values of the final two integer components of the summary for the array.

Array Files and Linked Lists

It is a simplification, but a useful one, to say that the arrays in an array file form a doubly-linked list. Each new array added to a file is placed at the tail of this list.

Because the list is doubly-linked, the head and tail of the list can be located immediately. The arrays can be located by moving a pointer through the list, in either direction, one array at a time.

At any time, the summary and name of the array at which the pointer is currently pointing can be retrieved and examined to determine whether the array is of interest. If it is, the initial and final addresses (the final two integer components of the summary) may be used to access—retrieve or update—the entire array, or any contiguous set of elements therein.

For example, if BEGIN and END are the initial and final addresses of an array, the first ten elements of the array can be retrieved by asking for the elements stored in addresses BEGIN through BEGIN+9. If the array contains an odd number of elements, the middle element can be retrieved by asking for the element stored in address (BEGIN+END)/2.

Read and Write Access

Array files may be opened for two kinds of access: read and write. A file opened for read access cannot be changed, either by adding a new array or by updating an existing one. Unless one of these operations must be performed, files should be opened for read access. The protection provided to files opened for read access is independent of any particular operating system.

A program may open only one array file at a time for write access. When a program attempts to open a file for write access, an error is signalled if another file is already open for write access, or if the file is already open for read access. An error is also signalled if a program attempts to open a file for read access if the file is already open for write access. (Errors are signalled through the standard SPICELIB error handling mechanism.)

File Handles

When a file is opened for either kind of access, it is assigned an integer 'handle'. A mapping between handles and Fortran logical units is maintained internally by the DAF subroutines.

As a means of accessing files, handles have two advantages over logical unit numbers.

1. They reduce the possibility that two or more program units of the same program will interfere with each other when both need to access the same array file. When a program opens an array file for the first time, the file is connected to a logical unit, and the unit is mapped to a handle. If the program attempts to open the same file again, the handle is returned immediately, and a counter is incremented. The file is not disconnected from the logical unit until it has been closed as many times as it has been opened. (This is analogous to the creation of multiple links to a single file under the UNIX operating system.) Any one program unit is prevented from releasing a file that is still being used by other program units.

2. They allow the SPICELIB subroutines to prevent files opened for read access from being modified. (Positive handles are assigned to files opened for read access, negative handles to files opened for write access. Any attempt to modify a file with a positive handle signals an error.) Note that this scheme is independent of any file protection provided by the host operating system.

DAF Family of Subroutines

SPICELIB contains a family of subroutines that can be used to create, populate, and manipulate array files. The name of each routine begins with the letters 'DAF', followed by a two- or three-character mnemonic. For example, the routine that begins a forward search of an array file is named DAFBFS, pronounced 'DAF-B-F-S'. A complete list of mnemonics, translations, and calling sequences can be found at the end of this document.

Each subroutine is prefaced by a complete SPICELIB module header that describes inputs, outputs, restrictions, and exceptions, discusses the context in which the subroutine should be used, and shows typical examples of its use. Any discussion of the subroutines in this article is intended as an introduction: the final documentation for any subroutine is its module header.

In this document, whenever a subroutine appears in an example, the translation of the mnemonic part of its name will appear to the right of the reference, in braces. For example,

```
CALL DAFBFS ( HANDLE )                 { Begin forward search }
```

Examples will make use of the structured DO ... END DO and DO WHILE ... END DO statements supported by the VAX/VMS Fortran compiler. These statements are easily converted to the standard equivalents

```
        DO label var = e1, e2, e3
           stmt
label   CONTINUE
```

and

```
label   IF       ( expr )
       .THEN
           stmt
           GO TO label
        END IF
```

Opening and Closing Array Files

An existing array file can be opened by supplying the name of the file to DAFOPR (for read access) or DAFOPW (for write access). Each routine returns a file handle, which must be used for all subsequent access to the file.

```
CALL DAFOPR ( FNAME, HANDLE )          { Open for read }
CALL DAFOPW ( FNAME, HANDLE )          { Open for write }
```

Once opened, an array file can be closed by supplying its handle to DAFCLS.

```
CALL DAFCLS ( HANDLE )                 { Close }
```

Creating Array Files

A new array file can be created by supplying the name of the file, the type of data in the file, values for ND and NI, an internal file name, and the number of records to be reserved. NI must be greater than or equal to 2. (For more information about the bounds on NI and ND see the subsection 'Summary records'.)

```
CALL DAFONW ( FNAME,                    { Open new }
              FTYPE,
              ND,
              NI,
              IFNAME,
              RESV,
              HANDLE  )
```

The internal name of an array file is simply a string of up to 60 characters, which may be used to characterize the contents of the file. Its primary value is that, being internal to the file, it remains unchanged when the file is transferred between environments.

Any number of records may be reserved at the front of an array file. By definition, the contents of these records are invisible to DAF subroutines, and may contain any information that the user wishes to store in them.

Once created, a new array file remains open for write access until explicitly closed.

Modifying Reserved Records

When a DAF is created, the number of reserved records must be specified in the call to DAFONW. Because it is not always possible to know at the time the file is created exactly how many reserved records are needed, the reserved record area may need to be modified later. The reserved record area can be modified in two ways: reserved records can be added or they can be removed. DAFARR adds RESV number of reserved records to the end of the reserved record area. DAFRRR removes RESV number of reserved records from the end of the reserved record area.

```
CALL DAFARR ( HANDLE, RESV )             { Add reserved records }
CALL DAFRRR ( HANDLE, RESV )             { Remove reserved records }
```

Adding Arrays

A new array can be added to an existing array file by calling four routines: DAFPS, DAFBNA, DAFADA, and DAFENA.

First, the summary is packed by DAFPS, which requires two arrays containing the double precision and integer components of the summary. It also requires the values of ND and NI for the file.

```
CALL DAFPS ( ND, NI, DC, IC, SUM )     { Pack summary }
```

The final two integer components of the summary are always used to store the initial and final addresses of the array imposing that NI be greater than or equal to two. These components are filled in after the array has been stored: any values for these components supplied by the user are ignored.

Next, the new array must be initialized by calling DAFBNA. DAFBNA requires the handle of the file (which must be open for write access), the array name, and the array summary.

```
CALL DAFBNA ( HANDLE, SUM, NAME )      { Begin new array }
```

The elements of the array are added by DAFADA. The elements may be supplied in one shot,

```
CALL DAFADA ( DATA, N )                { Add data to array }
```

or in any number of installments,

```
DO WHILE ( MORE )
   ...
   CALL DAFADA ( DATA, N )             { Add data to array }
END DO
```

Once the entire array has been supplied, DAFENA makes the addition permanent.

```
CALL DAFENA                            { End new array }
```

If the process is aborted before DAFENA is called, the summary and name are not stored, and the new array does not become a permanent member of the file. Space allocated for elements of the array cannot be removed from the file; however, it will be overwritten by the elements of the next array added to the file.

One way to abort the addition of an array to a file is to call DAFBNA start a new array in the same file, without first ending the current array.

Adding Arrays to Multiple Array Files

It is possible add data to arrays in multiple files in an interleaved fashion: addition of data to an array in one file can be interrupted in order to add data to an array in another file. To accomplish this, it is necessary to tell DAFADA and DAFENA which file to act upon. This file is called the 'current file'.

When DAFBNA is used to begin an array, the file specified by the handle passed to DAFBNA becomes the current file. Calls to DAFADA or DAFENA will add data to or end the last array begun in this file. If DAFBNA is called again, this time with a different handle, the file specified by that handle becomes current. Files that are not current are not affected in any way by beginning, adding data to, or ending arrays in the current file.

In any given file, an array that is in progress—that is, an array begun by DAFBNA but not yet ended by DAFENA—is called the 'current array' for that file. No file can have more than one current array.

In order to continue or end an array in a file that is no longer current, the file in question is selected as the current file by a call to DAFCAD:

```
CALL DAFCAD ( HANDLE )                { DAF, continue adding data }
```

After this call, the file identified by HANDLE will be the current file, and calls to DAFADA will add data to the current array in this file. The usual sequence of calls has the form:

```
CALL DAFCAD ( HANDLE  )               { DAF, continue adding data }
CALL DAFADA ( DATA, N )               { DAF, add data to array }
```

Since DAFENA can be used to end arrays only in the current file, DAFCAD is also used to select a file as current so that an array can be ended in that file:

```
CALL DAFCAD ( HANDLE  )               { DAF, continue adding data }
CALL DAFENA                           { DAF, end new array }
```

Only files that already have an array in progress may be selected as current by DAFCAD. An error will be signalled if DAFCAD is used to select an array file that does not have an array in progress.

The following example illustrates the use of DAFCAD:

We write data obtained from the routine GET_DATA (which is not a SPICELIB routine) into two separate array files. The first N/2 elements of the array DATA will be written to the first file; the rest of the array will be written to the second file.

Open the array files for write access, using either DAFOPW (if the files already exist) or DAFONW (if they do not).

```
CALL DAFOPW ( FNAME1, HANDL1 )
CALL DAFOPW ( FNAME2, HANDL2 )
```

Begin the new array files by calling DAFBNA.

```
CALL DAFBNA ( HANDL1, SUM1, NAME1 )
CALL DAFBNA ( HANDL2, SUM2, NAME2 )
```

Add data to the arrays, using DAFCAD to select the current file and DAFADA to add data to the current array in the current file.

```
CALL GET_DATA ( DATA, N, FOUND )

DO WHILE ( FOUND )

   CALL DAFCAD   ( HANDL1                           )
   CALL DAFADA   ( DATA,              N/2           )

   CALL DAFCAD   ( HANDL2                           )
   CALL DAFADA   ( DATA( N/2 + 1 ),   N - N/2       )

   CALL GET_DATA ( DATA, N, FOUND )

END DO
```

End each array by calling DAFENA, selecting the file in which to end the array by calling DAFCAD:

```
CALL DAFCAD ( HANDL1 )
CALL DAFENA

CALL DAFCAD ( HANDL2 )
CALL DAFENA
```

The notions of 'current array file' and 'current array' apply to both adding data to arrays and to searching array files. However, the files and arrays regarded as current for the purpose of searching are unrelated to those regarded as current for the purpose of adding data.

Reordering Arrays

Once arrays are written to a DAF, it is conceivable that their order may need to be changed. Suppose that the arrays in a DAF are to be ordered according to the arithmetic mean of the data they contain. Also suppose that the arithmetic mean of data in an array is stored in the second double precision component of the summary. After reading each summary and creating a vector IORDER with dimension N that specifies the new order of the arrays, the subroutine DAFRA can be used to reorder the arrays.

```
CALL DAFRA ( HANDLE, IORDER, N )       { Reorder arrays }
```

Searching

The process of locating an array of interest within an array file is known as 'searching'. The organization of the arrays as a doubly-linked list makes it possible to conduct searches in forward or backward order.

Subroutines DAFBFS and DAFFNA are used to search an array file in forward order. DAFBFS places a pointer at the head of the doubly-linked list formed by the arrays in the file. Each call to DAFFNA moves the pointer to the next array in the list. (The first call to DAFFNA moves the pointer to the first array.) DAFFNA returns a logical flag which is true whenever another array has been found, and is false when the tail of the list has been reached. All forward searches are variations on the following template:

```
CALL DAFBFS ( HANDLE )                 { Begin forward search }
CALL DAFFNA ( FOUND  )                 { Find next array }

DO WHILE ( FOUND )
   ...
   CALL DAFFNA ( FOUND )               { Find next array }
END DO
```

Subroutines DAFBBS and DAFFPA are likewise used to search an array file in backward order. DAFBBS moves the pointer to the tail (instead of the head) of the list; DAFFPA moves the pointer to the previous (instead of the next) array in the list. The template shown above is modified to conduct backward searches by replacing calls to DAFBFS and DAFFNA with calls to DAFBBS and DAFFPA, respectively:

```
CALL DAFBBS ( HANDLE )                 { Begin backward search }
CALL DAFFPA ( FOUND  )                 { Find previous array }
```

```
DO WHILE ( FOUND )
    ...
   CALL DAFFPA ( FOUND )              { Find previous array }
END DO
```

Once a search has begun, the pointer may be moved in either direction.

After the pointer has been moved to a new array, the summary and name of the array can be retrieved by DAFGS and DAFGN:

```
CALL DAFBBS ( HANDLE )                { Begin backward search }
CALL DAFFPA ( FOUND  )                { Find previous array }

DO WHILE ( FOUND )
   CALL DAFGS ( SUM  )                { Get summary }
   CALL DAFGN ( NAME )                { Get name }
    ...
   CALL DAFFNA ( FOUND )              { Find next array }
END DO
```

Once returned, a name can be examined directly. However, a summary must first be unpacked into its components by subroutine DAFUS:

```
CALL DAFUS ( SUM, ND, NI, DC, IC )    { Unpack summary }
```

The name and summary are used to determine whether the current array is of interest. For example, if the arithmetic mean of the elements in each array of an array file is stored in the second double precision component of the summary, then the following code fragment determines the initial and final addresses (IA and FA) of the array with the greatest average. (Assume function DPMIN returns the smallest double precision number supported in the host environment.)

```
MAXAVG = DPMIN()

CALL DAFBFS ( HANDLE )                { Begin forward search }
CALL DAFFNA ( FOUND  )                { Find next array }

DO WHILE ( FOUND )
   CALL DAFGS ( SUM )                 { Get summary }
   CALL DAFUS ( SUM, ND, NI, DC, IC ) { Unpack summary }

   IF ( DC(2) .GT. MAXAVG ) THEN
      MAXAVG  = DC(2)
      IA      = IC(NI-1)
      FA      = IC(NI  )
   END IF

   CALL DAFFNA ( FOUND )              { Find next array }
END DO
```

Recall that the final two integer components of any array summary—IC(NI-1) and IC(NI)—contain the initial and final addresses of the array.

Searching Multiple Array Files

Searching multiple array files simultaneously is a little like adding data to multiple files simultaneously: in each case, it becomes necessary to identify the file to act upon, when calling routines that don't accept an input handle argument.

As with adding data, the notions of 'current array file' and 'current array' apply to searching. Starting a search in an array file by calling either DAFBFS or DAFBBS makes that file the 'current file'. Subsequent calls to DAFFNA or DAFFPA advance or back up the array pointer in the current file. The last array found by DAFFNA or DAFFPA in the 'current file' is the 'current array' for that file. As mentioned above, there is no relation between the files or arrays that are considered current for searching and those considered current for adding data.

If, after a search is started in one array file, DAFBFS or DAFBBS are called to start a search in a second array file, the second file becomes current: DAFFNA, DAFFPA, DAFGN, and DAFGS will all operate on the second file.

The complete set of DAF routines that act on the current file (for searching) is:

```
DAFFNA                        { DAF, find next array }
DAFFPA                        { DAF, find previous array }
DAFGS                         { DAF, get summary }
DAFGN                         { DAF, get name }
DAFGH                         { DAF, get handle }
DAFRS                         { DAF, replace summary }
DAFRN                         { DAF, replace name }
DAFWS                         { DAF, write summary }
```

The routine DAFCS is used to continue a search in an array file that is no longer current. Calling DAFCS makes the file specified by the input handle argument the current file for searching:

```
CALL DAFCS  ( HANDLE )                  { DAF, continue search }
```

After this call, the routines in the above list will act upon the file designated by HANDLE. For example, to continue a forward search in that file,

```
CALL DAFCS  ( HANDLE )                { DAF, continue search }
CALL DAFFNA ( FOUND  )                { DAF, find next array }
```

and to continue a backward search,

```
CALL DAFCS  ( HANDLE )                { DAF, continue search     }
CALL DAFFPA ( FOUND  )                { DAF, find previous array }
```

while to get the name and summary of the current array in the file,

```
CALL DAFCS ( HANDLE )                 { DAF, continue search }
CALL DAFGN ( NAME )                   { DAF, get name }
CALL DAFGS ( SUM  )                   { DAF, get summary }
```

A search must have been started by DAFBFS or DAFBBS before it can be continued. An error will be signalled if DAFCS is used to continue a search in an array file in which no search has been started.

Accessing Array Elements

After an array of interest has been located, the entire array or any contiguous set of elements can be accessed—read or updated—by supplying a pair of addresses. Elements are read by DAFRDA and written by DAFWDA.

The following code fragment continues the example above by subtracting the average from each of the elements in the array. (Recall that IA and FA contain the initial and final addresses of the array.)

```
CALL DAFRDA ( HANDLE, IA, FA, DATA )   { Read data
                                         from address }
DO I = 1, FA - IA + 1
   DATA(I) = DATA(I) - MAXAVG
END DO

CALL DAFWDA ( HANDLE, IA, FA, DATA )   { Write data
                                         to address }
```

Note that it is not necessary to retrieve the entire array at once. The following code fragment illustrates how to process an array of unknown size using a fixed amount of local storage. The local array DATA is declared to be size CHUNK. DAFRDA reads a maximum of CHUNK elements from the double precision array and DAFWDA writes them. This technique is useful when the arrays stored in an array file may be arbitrarily large.

```
FIRST = IA
```

```
DO WHILE ( FIRST .LE. FA )
   LAST  = MIN ( FA,     FIRST + CHUNK - 1 )
   CALL DAFRDA ( HANDLE, FIRST, LAST, DATA )   { Read data
                                                 from address }
   NUMELE = LAST - FIRST + 1

   DO I = 1, NUMELE
      DATA(I) = DATA(I) - MAXAVG
   END DO

   CALL DAFWDA ( HANDLE, FIRST, LAST, DATA )   { Write data
   FIRST = FIRST + CHUNK                         to address }
END DO
```

Updating Summaries and Names

In the previous example, once the average value of the array has been subtracted from each element of the array, the value for the average stored in the summary is no longer valid (the average is now zero) and should be changed.

Subroutines DAFRS and DAFRN are analogous to subroutines DAFGS and DAFGN. DAFGS 'gets' the summary for the array to which the pointer currently points; DAFRS replaces it. DAFGN 'gets' the name of the array to which the pointer currently points; DAFRN replaces it.

If the index, K, of the updated array is known, then the new average for the array (zero) is stored by the following code fragment.

```
CALL DAFBFS ( HANDLE )                 { Begin forward search }

DO I = 1, K
   CALL DAFFNA ( FOUND  )              { Find next array }
END DO

CALL DAFGS ( SUM )                     { Get summary }
CALL DAFUS ( SUM, ND, NI, DC, IC )     { Unpack summary }

DC(2) = 0.D0

CALL DAFPS ( ND, NI, DC, IC, SUM )     { Pack summary }
CALL DAFRS ( SUM )                     { Replace summary }
```

Buffering

Unless the value of CHUNK is 128 (the number of double precision words in a record) and the initial address of the array happens to correspond to the first word of a physical record (neither of which is very likely), each call to DAFRDA or DAFWDA will involve reading partial records—data that spans across records. In general, successive calls will refer to different parts of at least one record.

In fact, as records are read from array files they are saved in an internal buffer maintained by the DAF subroutines. If any part of a record is needed, it can frequently be returned directly from the buffer, without accessing the file again. In particular, when an entire array is accessed sequentially, as in the example above, each of the necessary records is read exactly one time. When the elements of an array are accessed more randomly, the number of file accesses may increase somewhat.

It is possible, at any point in a program, to determine the number of file accesses prevented by the buffering scheme. The subroutine DAFNRR returns the number of physical records actually read, and the number of records or partial records that have been requested, as illustrated below:

```
CALL DAFNRR ( READS, REQS )           { Number of reads,
                                        requests         }
RATIO  = DBLE(READS) / DBLE(REQS)
PERCNT = INT ( RATIO * 100.D0 )
WRITE (*,*) 'Reads/requests (%) = ', PERCNT
```

Ideally, the ratio of reads to requests should approach zero. In the worst case, where it approaches one, the size of the buffer should probably be adjusted. (The module headers for DAFRDR and DAFWDR provide details on adjusting the buffer size.)

Conversion and Transfer of DAF's

In order to be transferred to a new environment, a binary file is converted to an equivalent SPICE transfer file—a formatted, sequential file that contains only printable ASCII characters and blanks (ASCII 32-126). In order to be used in the new environment, it is converted back to a binary file. This conversion process must occur because binary representations of numbers vary from machine to machine, the SPICE transfer representations do not.

There are two routines for converting DAFs from binary to transfer and transfer to binary formats: DAFBT and DAFTB.

DAFTB creates a new binary file. It then converts and writes the information from a previously opened SPICE transfer file to the binary file. Before returning to the calling program it closes the binary file. DAFTB leaves the SPICE transfer file open.

DAFBT opens an existing binary file. It then converts and writes information from it to a previously opened SPICE transfer file. Before returning to the calling program it closes the binary file. DAFBT leaves the SPICE transfer file open.

Note that the routines DAFBT and DAFTB make no use of the DAF reserved record area. They only convert the data portion of the DAF file.

When converting a binary file for transfer (or archiving), it may be necessary to add additional information—catalog or history information, for example—to the resulting text file. The following code fragment creates a text file containing an ASCII array file preceded and followed by markers. The SPICELIB routine TXTOPN opens a new text file; TXTOPR opens an existing text file for read access.

```
BEGMRK = '*** BEGIN ARRAY FILE ***'
ENDMRK = '*** END ARRAY FILE ***'

CALL TXTOPN ( FILENM, UNIT )
WRITE (UNIT,*) BEGMRK

CALL DAFBT ( BINARY, UNIT )          { Binary to transfer }

WRITE ( UNIT,* ) ENDMRK
CLOSE ( UNIT   )
```

The following code fragment converts the resulting text file into a binary array file.

```
CALL TXTOPR ( FILENM, UNIT )
READ ( UNIT,FMT='(A)' ) BEGMRK

CALL DAFTB ( UNIT, BINARY )         { Transfer to binary }

READ  ( UNIT,FMT='(A)' ) ENDMRK
CLOSE ( UNIT )
```

Structure

Every array file is a Fortran-77 direct access file, created by the following statement (or an equivalent statement producing the same results):

```
OPEN ( UNIT   = unit,
       FILE   = file name,
       ACCESS = 'DIRECT',
       RECL   = record length,
       STATUS = 'NEW'           )
```

The record length is processor dependent. The smallest possible value should be selected by the user such that each record in the file is large enough to contain 128 double precision numbers or 1000 characters, whichever is larger. Some suitable values for several compilers are shown below.

```
Compiler                          Record length
--------------                   -------------
HP Workstation
  /HP-UX
  /HP Fortran                          1024

Macintosh
  /Language Systems Fortran            1024

NeXT
  /Absoft Fortran                      1024

PC
  /Lahey                               1024

PC
  /Microsoft Fortran PowerStation      1024

Silicon Graphics
  /IRIX
  /SGI Fortran                          256

Sun
  /SunOS and Solaris
  /Sun FORTRAN                         1024

VAX
  /OpenVMS
  /VAX Fortran                          256

VAX
  /OSF/1
  /DEC Fortran                          256

VAX
  /VMS
  /VAX Fortran                          256
```

Organization

An array file contains five types of physical records:

1. A single 'file record'. This contains global information about the file.
2. An arbitrary number of 'reserved records'. These records are provided so that the user can store information about the data within the DAF. Typical information might include the source of the data, or the names of programs used to process and interpret it.
3. Some number of 'summary records'. These contain array summaries and pointers to other summary records. The number of summary records in a particular array file is a function of the number of arrays stored in the file.
4. Some number of 'name records'. These contain array names. An array file contains one name record for each summary record.
5. An arbitrary number of 'element records'. These contain elements of the arrays stored in the array file.

The File Record

The file record is always the first physical record in an array file. It contains seven items.

1. An identification word ('DAF/xxxx'). Where 'xxxx' is a string of four characters or less indicating the type of data stored in the DAF file. This is used by the SPICELIB subroutines to verify that a particular file is in fact an array file and not merely a direct access file with the same record length. When an array file is opened, an error is signalled if this keyword is not present.
2. The value of ND, the number of double precision components in each array summary.
3. The value of NI, the number of integer components in each array summary.
4. The internal name (60 characters) of the array file.
5. The record number of the initial summary record in the file.
6. The record number of the final summary record in the file.
7. The first free address in the file. This is the address at which the first element of the next array to be added to the file will be stored.

Reserved Records

By definition, reserved records are invisible to DAF subroutines. The contents and formats of reserved records are left entirely to the user. The initial reserved record is located in the second record of the file; the final reserved record immediately precedes the initial summary record of the file.

Reserved records may be used to store information about the data such as its source, names and sites of programs that processed it, names and sites of programs to interpret it, people to contact for support, its peculiarities or omissions, just to name a few.

Summary Records

A summary record contains a maximum of 128 double precisions words. The first three words of each summary record are reserved for the following control information:

1. The record number of the next summary record in the file. (Zero if this is the final summary record.)
2. The record number of the previous summary record in the file. (Zero if this is the initial summary record.)
3. The number of summaries stored in this record.

The record pointers form the basis of the array list. Each summary record is linked to two other summary records, allowing the summaries to be retrieved in forward or backward order. (The links between adjacent summaries in a summary record are implicit.) The names can be retrieved from the corresponding name records. And the locations (initial and final addresses) of the arrays themselves are stored in the summaries.

Although the control items are integer values, they are stored as double precision numbers. This allows summary records and element records, which contain only double precision numbers, to be buffered using the same mechanism.

The control items are followed immediately by the summaries themselves. The number of summaries (NS) that can fit in a single summary record depends on the size of a single summary (SS), a function of NI and ND:

```
                   (NI + 1)
SS       =  ND + --------          (Note that this is
                       2            integer division.)

SS * NS <=  125
```

```
NS       <=  125/SS                (Note that NS must be an
                                    integer greater than or
                                    equal to one.)
```

A summary record can be depicted as written below, where the numbers correspond to the number of the double precision word in the record.

```
-------------------------------------------------------------
| 1 | 2 | 3 | 4 | 5 | 6 |        ...       | 126 | 127 | 128 |
-------------------------------------------------------------
  ^   ^   ^
NEXT  |   |
     PREV |
         NSUM
```

If SS is the size (in double precision words) of each summary array, then the first summary is stored in words 4 through SS+3; the second summary is stored in words SS+4 through 2(SS)+3; and so on. For example, if SS is equal to 3, 41 (125/3) summaries can fit in the summary record, leaving two words empty. In this case the record can be pictorially represented as written below, where the label below the record indicates the summary number.

```
-------------------------------------------------------------
| 1 | 2 | 3 | 4 | 5 | 6 | ... | 124 | 125 | 126 | 127 | 128 |
-------------------------------------------------------------
  ^   ^   ^ { Summary 1 } ... {     Summary 41   }  ^     ^
NEXT  |   |                                         |     |
     PREV |                                        These words
         NSUM                                      are unused.
```

Unlike arrays, summaries are never split across physical record boundaries, so the end of each summary record may remain unused. Whenever the number of summaries stored in the current summary record reaches the maximum number that will fit, a new (empty) summary record is added to the end of the file.

It is now clear that the bounds on the values for ND and NI are determined by their relation to the number of double precision words used for summary information in the summary record. ND and NI must satisfy the following inequalities:

```
     (NI + 1)                      (Note that this is
ND + -------- <= 125                integer division.)
        2

0 <= ND <= 124

2 <= NI <= 250
```

Name Records

Each name record contains nothing but array names. A new name record is added to the file each time a new summary record is added. The new name record is located in the record immediately following the new summary record. Because a DAF is written in this manner, the number of summary records is equal to the number of name records.

Each time a new summary is added to a summary record, a new name is added to the corresponding name record. Therefore, the number of summaries in a summary record is equal to the number of names in the corresponding name record.

The values for ND and NI determine the maximum length for a name in the name record, NC:

```
                     (NI + 1)
NC  =   8 * ( ND + -------- )     (Note that this is
                        2          integer division.)
```

If the numbers in the summary record represent double precision words, and the numbers in the name record represent characters, the two records can be depicted as written below for a DAF whose format is specified by ND = 2 and NI = 2.

```
-------------------------------------------------------------
| 1 | 2 | 3 | 4 | 5 | 6 | ... | 124 | 125 | 126 | 127 | 128 |
-------------------------------------------------------------
  ^   ^   ^ { Summary 1 } ... {    Summary 41   }  ^     ^
NEXT  |   |                                        |     |
     PREV |                                       Words 127 and
         NSUM                                     128 are unused.

-------------------------------------------------------------
| 1 |...| 24 |.......| 961 | ... | 984 | 985 |.......| 1000 |
-------------------------------------------------------------
{   Name 1   }.......{    Name 41      }  ^              ^
                                          |              |
                                  Characters 985 through
                                  1000 are unused.
```

The first name is stored in characters 1 through NC of the record; the second name is stored in characters NC+1 through 2(NC); and so on.

Element Records

Most of the records in any array file are element records. Element records hold the elements of the arrays stored in the file. (The other records are used for accounting purposes only.)

Each element record contains up to 128 double precision numbers. An element record is always full (contains 128 numbers) unless it immediately precedes a summary record, in which case it may be partially filled.

The elements stored in a particular element record may belong to more than one array. However, elements belonging to the same array are stored contiguously within the record.

For example, suppose three arrays exist: A, B, and C. Array A has 10 elements, array B has 100 elements, and array C has 15 elements. If all of the elements are stored in the same element record, it could be pictorially represented as written below:

```
A[1]
A[2]
A[3]
  .
  .
  .
A[10]
B[1]
B[2]
B[3]
  .
  .
  .
B[100]
C[1]
C[2]
C[3]
  .
  .
  .
C[15]
```

A particular element record always lies between two summary/name record pairs, or between a summary/name record pair and the end of the file.

Designing a DAF

During the Voyager-2 encounter with Neptune, NAIF processed pictures using an image center finding technique in order to determine instrument pointing. For each body in each picture, a set of limb points—pixel and line pairs—was selected and an ellipse was fitted to the set, producing a center for the body.

Suppose that the pixel and line pairs for the points are to be stored so that existing and future models can be used to interpret the data. While it's true that the pixel and line numbers are integers, they must be converted to double precision numbers for the ellipse fitting processing. As double precision numbers, they could be stored in a DAF.

Data stored in a DAF is characterized by a set of double precision and integer values. In this case, each picture has several values associated with it that could be used to describe it: the picture number, the spacecraft identification code, the time at which the picture was taken, the identifier for the command load, the identifier for the camera, the identification code for the body whose center is described by the data. Examples of each of these items is given below:

```
Picture number        :         9230.4
Spacecraft event time :  -332927103.9324339
Spacecraft ID         :          -32
Load                  :          901
Camera ID             :            1
Body ID               :          899
```

To specify the 'format' of this DAF, the values of ND and NI must be determined. The number of double precision values used to describe the picture is two (picture number and spacecraft event time) so ND = 2. The number of integer values used to describe the picture is 4 (spacecraft ID, load, camera ID, and body ID). Because two of the integer components in the summary are used for storing locations of the arrays themselves, NI must always be two more than the number of integer values used to describe the data. In this example, NI = 4 + 2 = 6. Thus, the format of the DAF used to store the limb points is specified by ND = 2 and NI = 6.

Having determined the values for ND and NI, the value of NC can be computed. NC is defined as :

```
                      (NI + 1)
NC = 8 * ( ND + -------- )
                          2
```

So, for this example:

```
                      (6 + 1)
NC = 8 * ( 2 + -------- ) = 8 * ( 2 + 3 ) = 40
                         2
```

In this example, each array name may contain a maximum of forty characters. The array names could describe the science activity of the scan platform. They would then give a clue as to what the picture might contain. For example, 'Slew to center of Neptune' could be used to describe picture number 9230.4.

After the format has been determined, the user needs to write software to transfer the limb point data to the DAF, and to read and interpret limb point data that has been stored in a DAF. Higher level DAF-based software, much like the SPK or CK software, can be written to achieve this functionality.

Creating an Array File

The following example illustrates the use of addresses and lists within an array file by showing how a simple array file might be created, and how arrays might be added to that file.

Throughout the example, the following notations will be used:

- Within the file record there are several values: IDWORD, ND, NI, RI, RF, and FFA. IDWORD is a character string that contains the file architecture, DAF, and code for the type of data stored in the DAF file. This code is a string consisting of four characters or less and is described in the header of the routine DAFONW. ND and NI are the values of the parameters that define the format of the file. RI and RF are the record numbers of the initial and final summary records in the file. FFA is the first free address in the file.

- Within a particular summary record, NEXT and PREV are the record numbers of the next and previous summary records in the file, and NSUM is the number of summaries stored in the record.

The first step in creating a file is to determine the name to be given to the type of data stored in the DAF file. For this example, the data type will be 'Xmpl'. For more information about the restrictions on the character string describing the type of data in the DAF file, see the header of the routine DAFONW. The IDWORD written to the new file is the concatenation of the string 'DAF/' with the data type string. So, for this example, the IDWORD written to the new file is 'DAF/Xmpl'.

The next step is to select values for ND and NI. Normally, these are relatively small, allowing several summaries to fit in each summary record and thus increasing the speed with which the file can be searched. A new file is opened by calling DAFONW, specifying the selected values for ND and NI. Recall that the final two integer components of any array summary—IC(NI-1) and IC(NI)—contain the initial and final addresses of the array, so NI must be at least 2.

The example will be easier to follow, however, if the number of summaries that can fit in a summary record is minimized. Therefore, in this example ND and NI will take on unusually large values:

```
ND = 25
NI = 27
```

Each array summary requires 39 double precision words of storage:

```
     (NI + 1)
ND + -------- =  25 + 14 = 39
        2
```

Therefore, each summary record can hold 3 summaries:

```
125 (words per record)
-----------------------  =  3 (summaries per record)
 39 (words per summary)
```

If 'Summary(i)[j]' represents the j'th element of the i'th summary array, then the layout of a typical summary record is shown below.

```
Word     Value
----     ----------------
   1     NEXT
   2     PREV
   3     NSUM
   4     Summary( 1)[ 1]
   5     Summary( 1)[ 2]
           ...
  42     Summary( 1)[39]
  43     Summary( 2)[ 1]
           ...
  81     Summary( 2)[39]
  82     Summary( 3)[ 1]
           ...
 120     Summary( 3)[39]
 121     Unused
           ...
 128     Unused
```

The number of names that an array record can hold is equivalent to the number of summaries that the summary record can hold. In this example it's three.

Recall that NC, the maximum number of characters in an array name, is determined by the values of ND and NI. For this example, where ND = 25 and NI = 27, the value of NC is computed below:

```
                     (NI + 1)
NC  =    8 * ( ND + -------- )  = 8 * ( 25 + 14 ) = 8 * 39 = 312
                        2
```

Each array name may use up to 312 characters of storage. An array name does not have to be exactly NC characters long. NC is simply the limit on the length of the array name.

If 'Name(i)[j]' represents the j'th character of the i'th name, then the layout of a typical name record is shown below.

```
Character    Value
---------    --------------
        1     Name( 1)[  1]
        2     Name( 1)[  2]
                ...
      312     Name( 1)[312]
      313     Name( 2)[  1]
                ...
      624     Name( 2)[312]
      625     Name( 3)[  1]
                ...
      936     Name( 3)[312]
      937     Unused
                ...
     1000     Unused
```

Assume that RESV, the number of reserved records, is 10. When DAFONW opens the new file, it stores the file record information in record 1, the reserved records in records 2 through 11, and the initial summary record in record 12. Because the file is empty, the initial summary record, RI, is also the final summary record, RF.

```
RI = 12
RF = 12
```

DAFONW stores the lone name record for the file immediately after the summary record, in record 13. Therefore the first free address, FFA, in the file is the first word in record 14:

```
FFA = word + (record - 1) * 128
```

```
=    1 + (14     - 1) * 128

=    1 + 1664

=    1665
```

DAFONW also writes the internal file name to the file record. For this example the internal file name will be 'TESTFILE'. For the rest of the example, the file record will be depicted as a collection of values enclosed by braces and preceded by a record number:

```
r { IDWORD=x, ND=a, NI=b, IFNAME=c, RI=d, RF=e, FFA=f }
```

So, the file record for this example file is initially:

```
1 { IDWORD='DAF/Xmpl', ND=25, NI=27, IFNAME='TESTFILE',
                       RI=12, RF=12, FFA=1665           }
```

Because there is only one summary record, the values of NEXT and PREV in that record are both zero. Because the file contains no arrays, the value of NSUM is also zero. The information needed to create the summary record is complete. For the rest of the example, each summary record will be depicted as a collection of values enclosed by angle brackets and preceded by a record number:

```
r < NEXT=a, PREV=b, NSUM=c, (d,e),(f,g),(h,i) >
```

The ordered pairs enclosed in parentheses are the initial and final addresses of the arrays whose summaries are contained in the record. The remaining components of each summary are ignored in order to make the example easier to follow. Thus, the lone summary record for this example file is initially:

```
12 < NEXT=0, PREV=0, NSUM=0, (0,0),(0,0),(0,0) >
```

Name records will always be depicted as

```
r < " " >
```

Element records will always be depicted as

```
r < N >
```

where N is the number of elements stored in the record.

Once the initial summary and name records have been written, the file is complete, if uninteresting:

```
 1 { IDWORD='DAF/Xmpl', ND=25, NI=27, IFNAME='TESTFILE',
                        RI=12, RF=12, FFA=1665          }
 2
 .
 . Records 2 through 11 are reserved records.
 .
11
12 < NEXT=0, PREV=0, NSUM=0, (0,0),(0,0),(0,0) >
13 < " " >
```

Assume that an array A1, containing 100 elements, is to be added to the file. The array will be stored contiguously, beginning at the first free address. Thus, its initial and final addresses will be 1665 and 1764, respectively. The entire array fits into a single record, so one element record will be added to the file. The value of NSUM in the summary record is incremented by one. The new value of FFA is the address following the final address of the new array: 1765. This is stored in the file record.

```
 1 { IDWORD='DAF/Xmpl', ND=25, NI=27, IFNAME='TESTFILE',
                        RI=12, RF=12, FFA=1765          }
 2
 .
 . Records 2 through 11 are reserved records.
 .
11
12 < NEXT=0, PREV=0, NSUM=1, (1665,1764),(0,0),(0,0) >
13 < " " >
14 < 100 > 100 words for A1
```

Assume that a second array A2, containing 200 elements, is to be added to the file. The elements will be stored between addresses 1765 and 1964. The array will fill the remainder of the first element record, all of a second record, and part of a third, so two element records will be added to the file. The value of NSUM in the summary record is incremented again. And the new value of FFA (1965) is stored in the file record.

```
 1 { IDWORD='DAF/Xmpl', ND=25, NI=27, IFNAME='TESTFILE',
                        RI=12, RF=12, FFA=1965          }
 2
 .
 . Records 2 through 11 are reserved records.
 .
11
12 < NEXT=0, PREV=0, NSUM=2, (1665,1764),(1765,1964),(0,0) >
13 < " " >
14 < 128 > 100 words for A1, 28 words for A2
```

```
15 < 128 > 128 words for A2
16 <  44 >  44 words for A2
```

To add a third array A3, containing 150 elements, the process is repeated. The elements will be stored between addresses 1965 and 2114. The array will fill the remainder of the third element record, and part of a fourth, so one new element record is added. The value of NSUM is in the summary record is incremented again. And the new value of FFA (2115) is in the file record.

```
 1 { IDWORD='DAF/Xmpl', ND=25, NI=27, IFNAME='TESTFILE',
                        RI=12, RF=12, FFA=2115           }
 2
 .
 . Records 2 through 11 are reserved records.
 .
11
12 < NEXT=0, PREV=0, NSUM=3, (1665,1764),(1765,1964),(1965,2114) >
13 < " " >
14 < 128 > 100 words for A1, 28 words for A2
15 < 128 > 128 words for A2
16 < 128 >  44 words for A2, 84 words for A3
17 <  66 >  66 words for A3
```

Note that the final summary record is full, so new summary and name records will added to the file. (Record 17 will remain only partially filled.) The values of NEXT and PREV in the summary records are adjusted so that the records point to each other:

```
 1 { IDWORD='DAF/Xmpl', ND=25, NI=27, IFNAME='TESTFILE',
                        RI=12, RF=12, FFA=2115           }
 2
 .
 . Records 2 through 11 are reserved records.
 .
11
12 < NEXT=18, PREV=0, NSUM=3, (1665,1764),(1765,1964),(1965,2114) >
13 < " " >
14 < 128 > 100 words for A1, 28 words for A2
15 < 128 > 128 words for A2
16 < 128 >  44 words for A2, 84 words for A3
17 <  66 >  66 words for A3
18 < NEXT=0, PREV=12, NSUM=0, (0,0),(0,0),(0,0) >
19 < " " >
```

The file record is updated so that the value of RF points to the new summary record, and the value of FFA in the file record will point to the first word in the first record following the new name record (address 2433):

```
 1 { IDWORD='DAF/Xmpl', ND=25, NI=27, IFNAME='TESTFILE',
                        RI=12, RF=18, FFA=2433           }
```

```
 2
 .
 . Records 2 through 11 are reserved records.
 .
11
12 < NEXT=18, PREV=0, NSUM=3, (1665,1764),(1765,1964),(1965,2114) >
13 < " " >
14 < 128 > 100 words for A1, 28 words for A2
15 < 128 > 128 words for A2
16 < 128 >  44 words for A2, 84 words for A3
17 <  66 >  66 words for A3
18 < NEXT=0, PREV=12, NSUM=0, (0,0),(0,0),(0,0) >
19 < " " >
```

Adding more arrays is identical to the previous example: the necessary element records are added; the summary and name records are updated; and the value of FFA is updated. However, every third array also adds new summary and name records, and the values of RF and FFA are updated as well.

Transfer Format for Porting

In order to transfer a DAF file from one computer platform to another, it should be converted to a SPICE transfer file. Previously, other SPICELIB routines such as DAFA2B, DAFB2A, DAFT2B, and DAFB2T were recommended for converting DAF files. These routines are now obsolete; however, they will remain in SPICELIB for backwards compatibility. Two routines replace them, DAFTB and DAFBT. Please use these routines in new software for converting DAF files. Older software does not need to changed to use the new routines. Software that calls the obsolete routines will continue to work properly, and the files they produce can be read using newer versions of the NAIF Toolkit.

Examples

The next several sections present example programs and subroutines to show how the DAF subroutines can be used to manipulate array files.

All subroutines and functions used in the examples are from SPICELIB or they have been provided as examples themselves.

Example 1: Converting Between Transfer and Binary Format

This example is a complete program to convert a binary double precision array file (DAF) to an equivalent SPICE transfer file, suitable for porting to a different environment. The program queries the user for the names of the binary file to be converted and the transfer file to be created.

```
PROGRAM B2T

CHARACTER*(128)       BINARY
CHARACTER*(128)       XFER
INTEGER               XFLUN

WRITE (*,*)          'Name of binary file?'
READ  (*,FMT='(A)')  BINARY

WRITE (*,*)          'Name of transfer file?'
READ  (*,FMT='(A)')  XFER

CALL TXTOPN ( XFER,   XFLUN )
CALL DAFBT  ( BINARY, CFLUN )          { Binary to transfer }

CLOSE ( XFLUN )

END
```

Now convert from SPICE transfer format back to binary:

```
PROGRAM T2B

CHARACTER*(128)       BINARY
CHARACTER*(128)       TXFER
INTEGER               TXTLUN


WRITE (*,*)          'Name of text file?'
READ  (*,FMT='(A)')  XFER

WRITE (*,*)          'Name of binary file?'
READ  (*,FMT='(A)')  BINARY

CALL TXTOPR ( XFER,   XFLUN )
CALL DAFTB  ( XFLUN, BINARY )          { Transfer to binary }

CLOSE ( XFLUN )

END
```

The subroutines DAFBT and DAFTB can be embedded in programs with more sophisticated interfaces as well, such as X windows.

Example 2: Summarizing the Contents of an Array

The next example is a subroutine, SUMARR, that summarizes the contents of an array in a DAF. SUMARR assumes that a DAF file is open, a search (forward or backward) is in progress, and that an array has been found.

The subroutine takes two character string arrays as inputs. Each character array contains labels describing what the double precision and integer components of the summary represent. A program calling SUMARR might print out each label followed by its corresponding value in the array summary.

The function LASTNB returns the index of the last non-blank character in a string.

```
      SUBROUTINE SUMARR ( DNAMES, INAMES )

      CHARACTER*(*)       DNAMES ( * )
      CHARACTER*(*)       INAMES ( * )

C
C     SPICELIB functions
C
      INTEGER             LASTNB

C
C     Local variables
C
      CHARACTER*(80)      PNAME

      DOUBLE PRECISION    DC     ( 125 )
      DOUBLE PRECISION    SUM    ( 125 )

      INTEGER             HANDLE
      INTEGER             I
      INTEGER             IC     ( 250 )
      INTEGER             ND
      INTEGER             NI
      INTEGER             LONG

C
C     Look up the handle of the file, and the summary of the
C     array most recently found.
C
      CALL DAFGH  ( HANDLE         )
      CALL DAFHSF ( HANDLE, ND, NI )

C
C     Get, and unpack, the summary of the array. Note that SUM,
C     DC, and IC are dimensioned large enough to hold the
C     biggest possible summary.
C
      CALL DAFGS  ( SUM )
      CALL DAFUS  ( SUM, ND, NI, DC, IC )

C
```

```
C     Find the length of the longest print name. All names will
C     be transferred into a temporary string for printing.
C     Shorter names will be followed by enough blanks to make
C     the components line up.
C
      LONG = 1

      DO I = 1, ND
         LONG = MAX ( LONG, LASTNB ( DNAMES(I) ) )
      END DO

      DO I = 1, NI
         LONG = MAX ( LONG, LASTNB ( INAMES(I) ) )
      END DO

C
C     Write the summary: an aligned list of components, each
C     preceded by a descriptive name.
C
      WRITE (*,*)
      WRITE (*,*) 'Summary'
      WRITE (*,*) '-------'
      WRITE (*,*)

      DO I = 1, ND
         PNAME = DNAMES(I)
         WRITE (*,*) PNAME(1:LONG), ' :', DC(I)
      END DO

      DO I = 1, NI
         PNAME = INAMES(I)
         WRITE (*,*) PNAME(1:LONG), ' :', IC(I)
      END DO

      RETURN
      END
```

The following program uses SUMARR to summarize all of the arrays in a specific kind of array file. Each summary in the file contains four double precision components (minimum, maximum, average, standard deviation) and three integer components (sort flag, initial address, final address).

Note that the program opens a DAF and begins a forward search. After an array is found by DAFFNA, it is summarized by SUMARR. This process of searching and summarizing continues until all of the arrays in the file have been summarized.

```
      PROGRAM SUMDAF

      INTEGER             ND
      PARAMETER         ( ND = 4 )

      INTEGER             NI
      PARAMETER         ( NI = 3 )
```

```
      CHARACTER*(40)    DNAMES  ( ND )
      CHARACTER*(40)    INAMES  ( NI )
      CHARACTER*(128)   FILE

      INTEGER           HANDLE

      LOGICAL           FOUND

      DATA              DNAMES     / 'Largest value',
     .                               'Smallest value',
     .                               'Average value',
     .                               'Standard deviation'   /

      DATA              INAMES     / 'Sorted (1=yes, 0=no)',
     .                               'Begins at address',
     .                               'Ends at address'      /


      WRITE (*,*)
      WRITE (*,*)         'Name of file?'
      READ  (*,FMT='(A)')  FILE

      CALL DAFOPR ( FILE, HANDLE )

      CALL DAFBFS ( HANDLE )
      CALL DAFFNA ( FOUND  )

      DO WHILE ( FOUND )
         CALL SUMARR ( DNAMES, INAMES )
         CALL DAFFNA ( FOUND )
      END DO

      END
```

The summary of a typical array is shown below:

```
Summary
-------

Largest value        : 221.42123212345
Smallest value       : 17.332467369560
Average value        : 148.37378239493
Standard deviation   : 21.263546586965
Sorted (1=yes, 0=no) : 0
Begins at address    : 21463
Ends at address      : 29271
```

Example 3: Copying Arrays from One File to Another

The next example is a subroutine to copy an entire array from one array file to another. It assumes that the array file that is to be updated already exists, a search (forward or backward) is in progress, and that an array has been found.

It takes a single input: the name of the file to which the array is to be copied.

```
      SUBROUTINE COPYA ( FILE )

      CHARACTER*(*)      FILE

C
C     Local variables
C
      CHARACTER*(1000)   NAME

      INTEGER            FA
      INTEGER            FIRST
      INTEGER            FROM
      INTEGER            HANDLE
      INTEGER            IA
      INTEGER            IC       ( 250 )
      INTEGER            LAST
      INTEGER            ND
      INTEGER            NI
      INTEGER            TO

      DOUBLE PRECISION   DATA     ( 100 )
      DOUBLE PRECISION   DC       ( 125 )
      DOUBLE PRECISION   SUM      ( 250 )

C
C     Get the handle of the source file, and the values
C     of ND and NI for that file.
C
      CALL DAFGH  ( FROM         )
      CALL DAFHSF ( FROM, ND, NI )

C
C     Open the target file for write access.
C
      CALL DAFOPW ( FILE, TO )

C
C     Get the summary and name for the array to be copied.
C     Start the new array.
C
      CALL DAFGS ( SUM  )
      CALL DAFGN ( NAME )

      CALL DAFBNA ( TO, SUM, NAME )

C
C     Unpack the summary to get the initial and final addresses
```

```
C     of the original array.
C
      CALL DAFUS ( SUM, ND, NI, DC, IC )
      IA = IC(NI-1)
      FA = IC(NI  )

C
C     Copy the elements in groups of 100.
C
      FIRST = IA

      DO WHILE ( FIRST .LE. FA )

         LAST  = MIN ( FA, FIRST + 100 - 1 )
         CALL DAFRDA ( FROM, FIRST, LAST, DATA )
         CALL DAFADA ( DATA, LAST - FIRST + 1 )
         FIRST = FIRST + 100

      END DO

C
C     Make the addition permanent, then close the target file.
C
      CALL DAFENA
      CALL DAFCLS ( TO )

      RETURN
      END
```

Example 4: Copying Arrays to a New File in Sorted Order

The final example is a complete program to copy the arrays in one file to a second file, so that the arrays in the new file are sorted according to the value of the first double precision component of each summary.

The program assumes that the file contains fewer than 1000 arrays.

Subroutine ORDERD creates an order vector for a double precision array. Subroutine COPYA, defined in the previous example, is used to copy the arrays.

```
      PROGRAM DAFSRT

      CHARACTER*(128)       SOURCE
      CHARACTER*(128)       TARGET
      CHARACTER*(80)        ARCH
      CHARACTER*(80)        TYPE

      DOUBLE PRECISION      DC      (  125 )
      DOUBLE PRECISION      SUM     (  125 )
      DOUBLE PRECISION      VALUES  ( 1000 )
```

```
      INTEGER               HANDLE
      INTEGER               IC      (  250 )
      INTEGER               NA
      INTEGER               ND
      INTEGER               NI
      INTEGER               ORDER   ( 1000 )
      INTEGER               TO
      INTEGER               I
      INTEGER               J
      INTEGER               TARHAN

      LOGICAL               FOUND


C
C     Prompt for the names of the source and target files.
C
      WRITE (*,*)
      WRITE (*,*)         'Name of source (unsorted) file?'
      READ  (*,FMT='(A)')  SOURCE

      WRITE (*,*)
      WRITE (*,*)         'Name of target (sorted) file?'
      READ  (*,FMT='(A)')  TARGET

C
C     Determine the file architecture and the type of data
C     in the file.
C
      CALL GETFAT ( SOURCE, ARCH,   TYPE     )

C
C     Open the source file, and look up the values of ND and NI
C     for the file. That's needed for the call to DAFONW for
C     opening the new DAF file later on.
C
      CALL DAFOPR ( SOURCE, HANDLE          )
      CALL DAFHSF (         HANDLE, ND,   NI )



C
C     Collect the values of the first double precision
C     component of each summary.
C
      CALL DAFBFS ( HANDLE )
      CALL DAFFNA ( FOUND  )

      NA = 0
      DO WHILE ( FOUND )
         CALL DAFGS ( SUM )
         CALL DAFUS ( SUM, ND, NI, DC, IC )

         NA         = NA + 1
         VALUES(NA) = DC(1)

         CALL DAFFNA ( FOUND )
      END DO

C
```

```
C     Create an order vector for the values, such that ORDER(1)
C     is the index of the array with the smallest value,
C     ORDER(2) is the index of the array with the next smallest
C     value, and so on.
C
      CALL ORDERD ( VALUES, NA, ORDER )

C
C     Open the new file to initialize it, then close it
C     immediately. We'll set the type of data in the file
C     to be the same as the type of data in the original
C     file. COPYA will reopen the file and copy an array
C     to it.
C
      CALL DAFONW ( TARGET, TYPE, ND, NI, 'Sorted file', 0,
     .              TARHAN                                 )
      CALL DAFCLS ( TARHAN )

C
C     Look up the arrays in the specified order (starting from
C     the beginning of the file each time). Copy each one as
C     it is found to the target file.
C
      DO I = 1, NA
         CALL DAFBFS ( HANDLE )

         DO J = 1, ORDER(I)
            CALL DAFFNA ( FOUND )
         END DO

         CALL COPYA ( TARHAN )

      END DO

      END
```

Summary of Mnemonics

SPICELIB contains a family of subroutines that can be used to create, populate, and manipulate double precision array files. The name of each routine begins with the letters 'DAF', followed by a two- or three-character mnemonic. For example, the routine that begins a forward search of an array file is named DAFBFS, pronounced 'DAF-B-F-S'. The following is a complete list of mnemonics and translations, in alphabetical order.

```
ADA     Add data to array
ARR     Add reserved records
ARW     Address to record/word
BBS     Begin backward search
BFS     Begin forward search
BNA     Begin new array
CAD     Continue adding data
CLS     Close
CS      Continue search
ENA     End new array
FNH     File name to handle
FNA     Find next array
FPA     Find previous array
GH      Get handle
GN      Get name
GS      Get summary
HFN     Handle to file name
HSF     Handle to summary format
HLU     Handle to logical unit
HOF     Handles of open files
LUH     Logical unit to handle
NRR     Number of reads, requests
ONW     Open new
OPR     Open for read
OPW     Open for write
PS      Pack summary
RA      Re-order arrays
RCR     Read character record
RDA     Read data from address
RDR     Read double precision record
RFR     Read file record
RN      Replace name
RRR     Remove reserved records
RS      Replace summary
RWA     Record/word to address
SIH     Signal invalid handles
US      Unpack summary
WCR     Write character record
WDA     Write data to address
WDR     Write double precision record
WFR     Write file record
```

Many of the subroutines listed here are not normally used except to support other subroutines. For example, because the subroutines that read and write records (RCR, RDR, RFR, WCR, WDR, WFR) are low level routines, they are not usually called by a typical user, but instead by higher level DAF routines.

Summary of Calling Sequences

The calling sequences for the DAF subroutines are summarized below. Subroutines are grouped by function.

Opening and closing files:

```
ONW ( FNAME, FTYPE, ND, NI, IFNAME, RESV, HANDLE )
OPR ( FNAME,                            HANDLE )
OPW ( FNAME,                            HANDLE )
CLS (                                   HANDLE )
```

Modifying reserved records:

```
ARR ( HANDLE, RESV )
RRR ( HANDLE, RESV )
```

Adding an array to a file:

```
BNA ( HANDLE, SUM, NAME )
ADA ( DATA,   N         )
CAD ( HANDLE            )
ENA
```

Finding an array within a file:

```
BFS ( HANDLE )
FNA ( FOUND  )

BBS ( HANDLE )
FPA ( FOUND  )

CS  ( HANDLE )

GH ( HANDLE )
GN ( NAME   )
GS ( SUM    )
RN ( NAME   )
RS ( SUM    )

US  ( SUM, ND, NI, DC, IC      )
PS  (      ND, NI, DC, IC, SUM )
```

Reading and writing arrays:

```
RDA ( HANDLE, BEGIN, END, DATA )
WDA ( HANDLE, BEGIN, END, DATA )
```

Reordering arrays:

```
RA ( HANDLE, IORDER, N )
```

Converting array files:

```
BT ( BINARY, TXTLUN )
TB ( TXTLUN, BINARY )
```

Reading, writing physical records:

```
RFR ( HANDLE, ND, NI, IFNAME, FWARD, BWARD, FREE )
WFR ( HANDLE, ND, NI, IFNAME, FWARD, BWARD, FREE )

RCR ( HANDLE, RECNO, CREC )
WCR ( HANDLE, RECNO, CREC )

RDR ( HANDLE, RECNO, BEGIN, END, DATA, FOUND )
WDR ( HANDLE, RECNO,             DREC        )
NRR ( READS,  REQS                           )
```

Internal conversions:

```
HSF ( HANDLE, ND, NI )

FNH ( FNAME,  HANDLE )
HFN ( HANDLE, FNAME  )
HLU ( HANDLE, UNIT   )
LUH ( UNIT,   HANDLE )

ARW ( ADDR, RECNO, WORDNO       )
RWA (       RECNO, WORDNO, ADDR )
```

Error handling utilities:

```
HOF ( FHSET          )
SIH ( HANDLE, ACCESS )
```

Related routine–Determining file architecture and type:

```
GETFAT ( FILE, ARCH, TYPE )
```

Obsolete Routines

The following is a list of routines and their replacements. NAIF's policy is to maintain the old routines to assure that existing software continues to function. Bugs will be fixed; however, no new functionality will be added to them. The replacement routines should be used in any new software that is being developed. Existing software does not need to be updated.

As always, for more information about a given routine, see its header documentation.

```
Routine   Replacement   Description
--------  ------------  ----------------------------------
DAFOPN    DAFONW        Open new DAF file
DAFA2B    DAFTB         Convert transfer to binary format
DAFB2A    DAFBT         Convert binary to transfer format
DAFB2T    DAFBT         Convert binary to transfer format
DAFT2B    DAFTB         Convert transfer to binary format
```