CFD General Notation System
Advanced Data Format (ADF) User's Guide

Document Version 1.1.1

Contents

1 Introduction

Advanced Data Format (ADF) is a library of basic database management and I/O subroutines that implements a relatively simple hierarchical database. ADF is written in ANSI C to enhance portability of the software, and the design of the database allows for portability of files from platform to platform. There is also a Fortran interface. The files are self-describing (i.e., it is possible to browse files and determine their contents) and extensible (i.e., many different pieces of software on different platforms may add or modify information).

The routines allow the user to construct a tree structure with their data. (See Figure 1 on p. 4.) This structure is very similar to the directory structures of the UNIX or DOS operating systems. ADF also allows links between nodes within the same file or to different files. This feature works somewhat like "soft links" in the UNIX operating system. The major difference between the aforementioned directory structures and ADF is that the nodes not only contain information about their children (next lower-level nodes) but may also contain data.

The installation package includes the source code to ADF, the Fortran interface, sample files, and a simple file browser.

1.1 History of Project

ADF was developed as part of the CFD General Notation System(CGNS) project. The purpose of the CGNS project is to define the data models for Navier-Stokes based Computational Fluid Dynamics (CFD) technology, develop a set of standard interface data structures for those data models, and develop the software that will allow implementation of those data structures in existing and future CFD analysis tools. The CGNS system consists of a collection of conventions, and software conforming to these conventions, for the storage and retrieval of CFD data. Adherence to these conventions is intended to facilitate the exchange of CFD data between sites, between application codes such as solvers and grid generators, and across computing platforms.

Once the data models (called the Standard Interface Data Structures, or SIDS) were defined, a project was started to write software that could faithfully reproduce that information on disk. ADF is the result of that project. While ADF was developed specifically for the CFD process, it is quite general and has no built-in knowledge of CFD; therefore, it should be applicable to storing any type of data that lends itself to a hierarchical definition.

1.2 Other Software Interfaces and ADF

Two other software packages were investigated as part of this project. The first database interface investigated was the Hierarchical Data Format (HDF) developed at the National Center for Supercomputing Applications at the University of Illinois. This database system has a large user base and support and has been in existence more than 6 years with utilities and graphical routines written with both a C and a Fortran interface. The limitation of HDF was that it was not truly hierarchical despite its name. Any hierarchy has to be built using naming conventions. Since the CGNS data models indicated a natural hierarchical structure, it seemed appropriate to develop database software that worked in that mode by design. The ADF design considerations are summarized in Appendix E.

The second was the Common File Format (CFF) developed by McDonnell Douglas Aerospace. CFF is a second-generation database management system that provides a unifying file structure for CFD data. The purpose of the Common File is to insulate the user from the myriad of different computer types that make up computer systems so that the user or application programmer may process a file from another machine without performing explicit conversions. CFF was written in Fortran; however, it was felt that portability and extensibility could be enhanced using C. Much of the experience gained from the McDonnell Douglas group is incorporated into ADF due to the cooperative efforts of personnel from the CFD group at McDonnell Douglas Aerospace.

1.3 Organization of Manual

The main section of this manual explains the basics of the hierarchical structure of an ADF database. In addition, the concept of a node as the basic building block of the hierarchy is developed in detail. The remaining sections of this manual are extensive. They provide a glossary of terms and conventions, as well as information related to the ADF version releases, version control numbering, and architectures that are supported by ADF. There are two examples, one in Fortran and one in C, in Appendix J and Appendix K respectively, that implement the structure illustrated in Figure 1 on p. 4. These examples should help familiarize the new user with ADF. Design considerations, file optimization, and portability issues are also discussed. The individual ADF core routines are described in detail in Appendix I. They are categorized into database-level routines, data structure and management subroutines, data query subroutines, data I/O subroutines, and some miscellaneous utility subroutines. Numerous examples are included to clarify the use of each subroutine. Appendix G provides a summary of ADF error messages, and Appendix H lists default values for various parameters and limits on dimensions of arrays.

2 The ADF Software Library

The ADF library is a hierarchical database system that is built around the concept of a "node". Each node contains information about itself and its ancestors and possibly data (e.g., arrays, vectors, character strings, etc.). Each of these nodes, in turn, can be connected to an arbitrary number of children, each of which is itself a node. In this system, an ADF node contains user-accessible information related to identification, name, type, and amount of data associated with it, and pointers to child nodes. Basic nodal information includes:

- a unique ID (essentially a file location pointer)
- a name (character string) used to describe the node and its data
- a label (character string) an additional field used to describe the node and its data. It is analogous to, but not exactly the same as, the name.
- information describing the type and amount of data
- data
- IDs of child nodes

There are no restrictions on the number of child nodes that a node can have associated with it in the ADF database. This structure allows the construction of a hierarchical database as shown in Figure 1 on p. 4. As illustrated in the figure, it is possible to reference nodes in a second file (*ADF_File_Two*) from the original file (*ADF_File_One*). This is the concept of "linking."

A node knows about itself and its children, but it does not know anything about its parent. This means that it is possible to traverse "down" the tree by making queries about what lies below the current node, but it is not possible to traverse "up" the tree by making queries about nodes above a given node. If it is desired to move back up the tree, the user must keep track of that information.

All ADF files start with the root node, named "ADF MotherNode". This node is created automatically when a new file is opened. There is only one root node in an ADF file.

2.1 ADF Node Attributes

There is a single building block called a node (see Figure 1) used to construct the hierarchical structure. Each node may have zero to many subnodes that are associated with it, as well as its own data. The following are a list of attributes accessible by the user for a node in the ADF hierarchical database system.

Child Table	A table of file pointers and names for each of the node's children.
Data	The data associated with a node.
Data Type	A 32-byte character field, blank filled, case sensitive. Specifies the type of data (e.g., real, integer, complex) associated with this node. The supported data types are listed in Table 1 on p. 5.
Dimensions	An integer vector containing the number of elements within each dimension. For example, if the array `A` was declared (using Fortran) as `A(10,20)`, the Dimension vector would contain two entries (10,20).

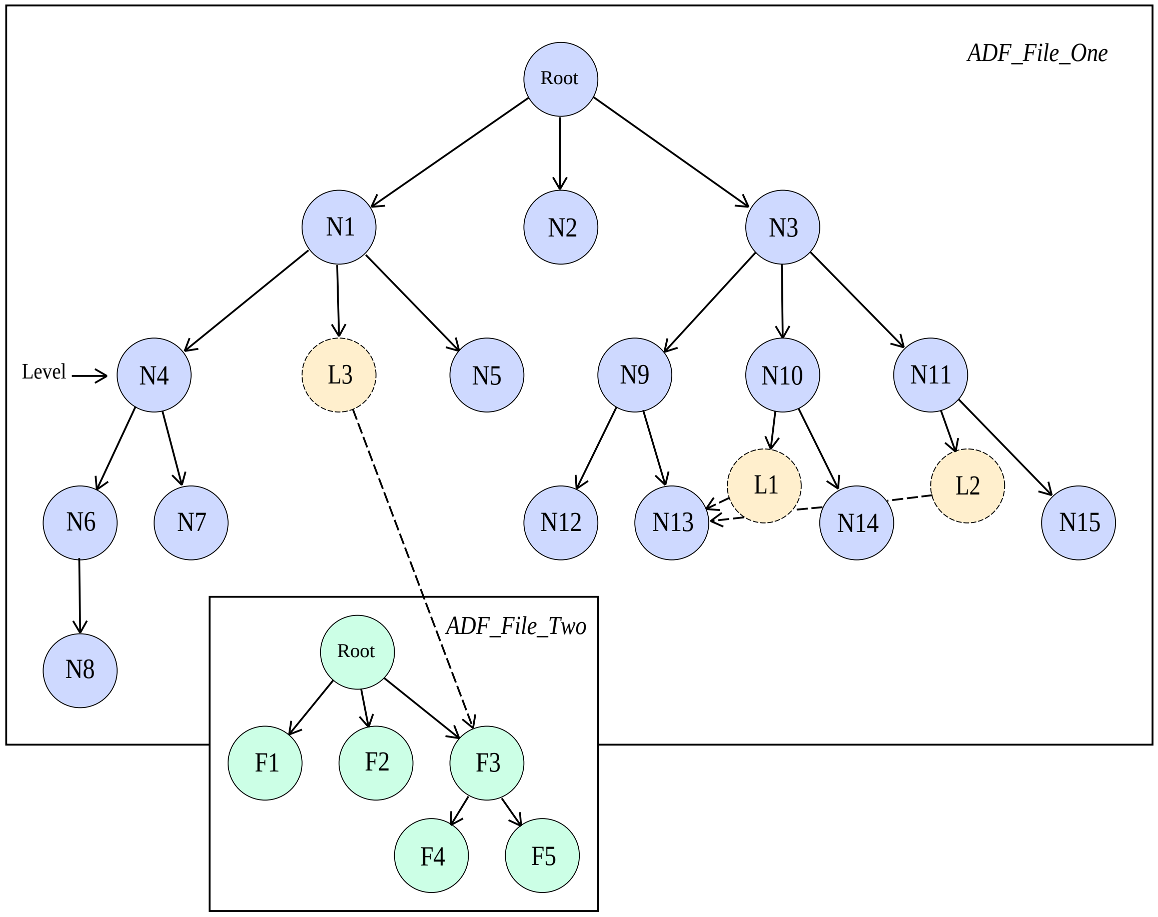

Figure 1: Example Database Hierarchy of Nodes

ID	A unique identifier to access a given node within a file. This field contains sufficient information for ADF to locate the node within a file. For any given node, the ID is generated only after the file it resides in has been opened by a program and the user requests information about the node. The ID is valid only within the program that opened the file and while that file is open. If the file is closed and reopened, the ID for any given node will be different. Within different programs, the node ID for the same node will be different. The ID is never actually written into a file.
Label	A 32-byte character field. The rules for Labels are identical to those for Names. Unlike names, Labels do not have to be unique. The Label field was introduced to allow "data typing" similar to the "typedef" concept in C. Using the Label field in this way allows programs to know some additional information about the use of the node itself or its child nodes and to call specific subroutines to read the data or react in specific ways upon detection of the type.
Name	A 32-byte character field. The names of child nodes directly attached

Table 1: Data Types

Data Type	**Notation**
No Data	`MT` (i.e., eMpTy)
Integer 32	`I4`
Integer 64	`I8`
Unsigned Integer 32	`U4`
Unsigned Integer 64	`U8`
Real 32	`R4`
Real 64	`R8`
Complex 64	`X4`
Complex 128	`X8`
Character (unsigned byte)	`C1`
Byte (unsigned byte)	`B1`
Link	`LK`

to a parent node must be unique. For example, in Figure 1, all nodes directly attached to N3 must have unique names. When a request to create a new node is made, ADF checks the requested name against the other names of the child nodes of the specified parent. If the requested name is not unique, ADF returns an error.

Legal characteristics of a name are a A-Z, a-z, 0-9, and special characters (ASCII values from 32 to 126, except for the forward slash "/" (ASCII number 47)). Names will be blank filled to 32 bytes; they are case sensitive. Leading blanks are discarded and trailing blanks are ignored, whereas internal blanks are significant.

Note: Names passed to ADF from C must have the null "\0" character appended to them. Names returned from ADF through the C interface will have the null character appended to them. Therefore, C programs should allocate 33 bytes for any Name in order to accommodate the null character.

Fortran programs can allocate 32 characters for Names. The Fortran interface takes care of adding or removing the null character as required.

Names of Subnodes	A list of names of the subnodes (children) of a node. (This is the information contained in the child table.)
Number of Dimensions	The dimensionality of the data. ADF views all data as an array and can handle from zero (i.e., no data) to 12 dimensions. A "0" is used if the data type is empty. Thus, a scalar is viewed as a vector with one dimension and length 1.
Number of Subnodes	The number of child nodes directly attached to any given node. Each node can have zero or more child nodes directly associated with it.

Pointer	An address, from the point of view of a programming language. Pointers are like jumps, leading from one part of the data structure to another.

2.1.1 Data Type Definitions

Structure
It is possible to define a "structure" in ADF similar to the way that "struct" is defined in the C programming language. ADF will treat each instance of the structure as a single instance of data. A structure is described by using a string. For example: "`I4,I4,R8,R4`".

Notes:

- Structures can be only as complex as can be described in 32 characters.
- This construct is not very portable; therefore, its use is highly discouraged.

Character
A character type is intended for ASCII-type character information. There may be different system architectures that use different representations for translating data. Byte data type should be used for pure byte or bit data usage.

Precision
The `R4` and `R8` are single and double precision, respectively, on a 32-bit system architecture. On the Cray, single precision is `R8` and double precision may be `R8` or `R16`, depending on the compile settings. ADF tracks the number of bits to guarantee precision.

Link
A link is denoted by "`LK`" in Table 1 on p. 5 and defines the linkage between nodes and subnodes. A link provides a mechanism for referring to a node that physically resides in a different part of the hierarchy or a different file. A link within ADF parallels a soft link in the UNIX operating system in that it does not guarantee that the referenced node exists. ADF will "resolve" the link only when information is requested about the linked node.

2.2 Acquiring the Software and Documentation

The ADF software is distributed as part of the CGNS Library, available from SourceForge, at http://sourceforge.net/projects/cgns. This manual, as well as the other CGNS documentation, is available in both HTML and PDF format from the CGNS documentation web site, at http://www.grc.nasa.gov/www/cgns/.

A ADF Glossary of Terms

ADF	The initialism (acronym) for Advanced Data Format.
Child	One of the subnodes of a Parent. A child node does not have knowledge of its parent node. The user must keep track of this relationship.
Database	A hierarchy of ADF nodes. By use of links, it may physically span multiple files.
File	An ADF file, which a single root node and its underlying structure.
ID	A unique identifier to access a given node within a file. This 8-byte field contains sufficient information for ADF to locate the node within a file. For any given node, the ID is generated only after the file it resides in has been opened by a program and the user requests information about the node. The ID is valid only within the program that opened the file and while that file is open. If the file is closed and reopened, the ID for any given node will be different. Within different programs, the node-ID for the same node will be different. The ID is never actually written into a file.
Link-Node	A special type of node. Links are created using the `ADF_Link` subroutine. The data type of this node is `LK`, and its data is a one-dimensional array containing the name of the file (if other than the current file) containing the node to be linked and the full path name in that file from the root node to the desired node. Links provide a mechanism for referring to a node that physically resides in a different part of the hierarchy. The node pointed to by a link may or may not reside in the same file as the link itself. A link within ADF is very similar to a "soft" link in the UNIX operating system in that it does not guarantee that the referenced node exists. ADF will "resolve" the link only when information is requested about the node. If the ID of a link-node is used in an ADF call, the effect of the call is the same as if the ID of the linked-to node was used. Note that a link node does not have children itself. In Figure 1 on p. 4, the children seen for `L3` are `F4` and `F5`. If a child is "added" to `L3`, then in reality, the child is added to `F3`. There are specialized subroutines provided to create link nodes and extract the link details.
Node	The single component used to construct an ADF database.
Node name	A node has a 32-character name. Every child node directly under a given parent must have a unique name. Legal characteristics in a name are `A-Z`, `a-z`, `0-9`, and special characters (ASCII values from 32 to 126, omitting the forward slash "`/`", ASCII number 47). Names will be blank filled to 32 bytes; they are case sensitive. Leading blanks are discarded and trailing blanks are ignored, whereas internal blanks are significant.
Parent	A node that has subnodes directly associated with it.
Pathname	Within a database, nodes can be referenced using the name of a node along with its parent ID, or by using a "pathname" whose syntax is roughly the same as a path name in the UNIX environment. A pathname that begins with a leading slash "`/`" is assumed to begin at the root node of the file. If no leading slash is given, the name is assumed to begin at the node specified by the parent ID. Although there is a 32-character limitation on the node Name, there is no restriction on

the length of the pathname. For example, equivalent ways to refer to node `N8` in Figure 1 are:

- Node-ID for `N6` and name = "`N8`"
- Node-ID for `N4` and name = "`N6/N8`"
- Node-ID for `N1` and name = "`N4/N6/N8`"
- Node-ID for the `Root_Node` and name = "`/N1/N4/N6/N8`"

B History of ADF Version Releases

This section contains a list of all the versions of ADF Software Libraries that have been released since the first release in December 1995. Along with each release, we include a brief synopsis of the improvements made in this version. In the future, newer versions may be implemented and released and they will also be accessible on the World Wide Web.

ADF A01 (December 1995)	This was the first release of the software. It includes the majority of the subroutines that are summarized in this document, and it was internally distributed for use at Boeing and to members of the working group (NASA and McDonnell Douglas).
ADF A02, A03 (July 1996)	This release includes improvements and some bug fixes for the A01 version.
ADF B01 (September 1996)	This release includes the first release of the ADF library and with the following subroutines added: Delete a Node, Add a Name (Change) on a Node, Change the Parent Node (move a Child).
ADF C00 (October 1996)	This release includes the performance enhancements and support for more platforms (Cray T90, SGI 6.2, DEC Alpha, Intel Paragon).
ADF AXX (Future Release)	It is anticipated that there will be future releases of the ADF core libraries, and they will include at a minimum the following additional routines (plus others that will provide enhancements as they are needed): Delete an Existing Database and Garbage Collection. Garbage Collection will redistribute the ADF file to use free-space that is not located at the end of the file.

C ADF File System Architectures

The following platform architectures have been tested and used both for functionality testing of the ADF core software libraries and for testing and running the prototype.

Table 2: Platform Architectures

Release	**Machine**	**OS Version**	**Native Format**
A01	Cray	Unicos 8.0	Unicos 8.0
A01	SGI/IRIS	4.0.5	IEEE Big Endian[1]
B01	HP	9.05	IEEE Big Endian
B01	SGI/IRIS	5.03	IEEE Little Endian
C00	Intel Paragon	—	IEEE Little Endian
C00	Dec Alpha	—	IEEE Little Endian
C00	SGI/IRIS	6.2	IEEE Big Endian
C00	Cray T90	Unicos 9.02	Cray Format

[1] In the table, "Endian" refers to the ordering of bytes in a multi-byte number. Big endian is a computer architecture in which, within a given multi-byte numeric representation, the most significant byte has the lowest address (the word is stored "big-end-first"). Little endian is a computer architecture in which, within a given 16- or 32-bit word, bytes at lower addresses have lower significance (the word is stored "little-end-first").

D ADF File Version Control Numbering

The ADF file version control number scheme is described below. The format for the version number is a field of six digits or characters:

```
AXXxxx
```

where:

`A` Major revision number. Major internal structure changes. This number is not expected to change very often, if at all, because the backward compatibility is only available by explicit policy decision.

One alphabetic character.

Range of values: `A-Z`, `a-z`

In the unlikely event of reaching `z`, then use any other unused printable ASCII character, except the blank or symbols used by the "*what*" command

```
@()#~>\
```

`XX` Minor revision number. New features, minor changes, and bug fixes. Backward but not forward compatible.

Two-digit hexadecimal number (uppercase letters).

Range of values: `00-FF`

Reset to `00` with changes in major revision number.

`xxx` Incremental number. Incremented with every new version of the library (even if no changes are made to the file format). Files are forward and backward compatible.

Three-digit hexadecimal number (lowercase letters).

Range of values: `000-fff`

Does not reset.

The following definitions are used:

Forward compatible	Older versions of libraries can read and write to files created by newer versions of libraries.
Backward compatible	Newer versions of libraries can read and write to files created by older versions of libraries.

E ADF Design Considerations

This section provides a summary of the design considerations that were used in the construction of the ADF software library.

E.1 ADF File Header Information

Every ADF file has a header section that contains information about the file itself. The following information from this header is available to the user:

- ADF version number of the library that created the database.
- File creation date and time.
- File modification date and time.
- Data format used in database (IEEE big endian, IEEE little endian, etc.)

[Under data format, "endian" refers to the ordering of bytes in a multi-byte number. Big endian is a computer architecture in which, within a given multi-byte numeric representation, the most significant byte has the lowest address (the word is stored "big-end-first"). Little endian is a computer architecture in which, within a given 16- or 32-bit word, bytes at lower addresses have lower significance (the word is stored "little-end-first").]

E.2 ADF File Optimizations

To optimize the performance of ADF, the following techniques have been incorporated into the ADF software to enhance performance of the file:

- Use block-based, unbuffered (raw mode) I/O where available, with block sizes of 4096 bytes.
- Align medium to large data chunks on block boundaries.
- If the data size is equal to or greater than 2048, then
  - align data to the next block and
  - add extra to free (garbage) lists
- Avoid, where possible, small- to medium-sized chunks of data that span block boundaries.
- Align, where possible, to the next block and add extra to free (garbage) lists.

Allow data space to grow by linking data chunks. It is possible to increase the last dimension; doing so will extend the data. However, internal dimensional changes will corrupt the existing data.

The pointer table for child nodes will also contain the child node names.

E.3 ADF File Portability

To address code portability and future needs, the following design decisions were made:

- Use larger than 32-bit file pointers to allow for files larger than 4 Gigabytes. (C routine `lseek` may not handle this, but ADF files should allow it.)
- Use 48-bit pointers within a block of data. 32-bit pointers to 4096-byte blocks and a 16-bit pointer to a position within the block. This allows for files with 4 gigabyte blocks, in other words $2^{32} \times 4096 = 17.5922 + 12$ bytes or 17.59 Tera bytes.
- The ID pointers will be 64-bit coded IDs. Users may use double data type (`real*8`). This is parceled as follows: 32 bits for block number, 12 bits for block offset, and 20 bits for file identifier.
- Encode the integer information, other than data, in ASCII, Hex-based notation.

E.4 ADF File Error Checking

Error checking has been implemented for the ADF file and is summarized here. Each item in the ADF file will have item-specific boundary tags surrounding it to provide file-based corruption checking. For variable-sized items, the associated boundary tags will include file-based size information. Information will be written to the disk in a sequence that will not allow corrupt files. For example, when adding a new child node, the complete child information will be written before the parent's child-table is updated. An ADF-core subroutine for downloading data to disk will be provided.

E.5 ADF Source Code Considerations

The ADF library of source code will incorporate the use of UNIX "what" strings for the ADF version number and also RCS versioning information in the source code and in the object code. The source code is written in portable ANSI C, using POSIX-defined system calls.

E.6 ADF Node Header Information

The following information is contained in an ADF node header:

- node boundary tag
- name (32 characters)
- label (32 characters)
  - number of subnodes (`num_sub_nodes`) is an integer (ASCII, Hex encoded in file).
  - address pointer of the subnodes indicated by the variable.
- `sub_nodes`, which are pointers (ASCII, Hex encoded in file) to a table (the "child table") of file pointers and names for each of the node's children. Note that the child name information is redundant and included for performance.
- data type is specified in `data_type` (32 characters allowed).

- dimensionality of the data, called `num_dimensions`. It is ASCII, Hex encoded in the file.
- dimension values are listed in the integer array of 12, `dimension_values`. The dimension values are Hex encoded.
- integer value for the number of data chunks is found in `num_data_chunks` (ASCII, Hex encoded).
- data address, which is as follows:
  - if `num_data_chunks` = 1, then a file pointer to data
  - if `num_data_chunks` > 1, then a file pointer to a table of `num_data_chunks` file pointers and associated data sizes
- ending node boundary tag.

E.7 Fortran Character Array Portability Concerns

Fortran character arrays are different from any other array type because they inherently include declared length information. Abstractly, they are a compound type: an array and an integer. The ANSI standards do not specify the implementation mechanism for handling this data type and so it is left to the vendor. As one might expect, vendors have devised different policies. This is particularly evident in how argument lists are created and used. The matter is further complicated when writing functions in other languages that are to be called from Fortran.

To keep the interfaces simple and to keep the Fortran and C data I/O calls similar (as opposed to having separate data I/O functions for character data), ADF suggests abiding by the following rules (these are required for Cray T90-mode users):

- Do not pass character arrays as the actual data arguments to any ADF read or write function unless that node has been defined with a data type of `C1`.
- If a node has been defined as data type `C1`, then pass character arrays only as the actual data arguments to all ADF read and write function.

Note to Cray T90-mode users: The above rules must be followed. In addition, the given node must be available and have its data type correctly defined. Error handling is not possible otherwise, and ADF will abort or fail regardless of how the error state flag is set.

E.8 Integer 64 Data Type Portability Concerns

For portability reasons, it is suggested that the use of the `I8` data type be restricted to a 64-bit environment.

E.9 Compound Data Types Portability Concerns

For the transportability reasons discussed below, use of compound data types is not recommended.

When using compound data types (e.g., with structures in C), it is important to be aware of data alignment issues. If one is not careful, the actual size of the structure in memory may be larger than the sum of the individual members. The total size depends on the order and word boundary alignment requirements of the specific data types. This is platform and compiler dependent and not handled by ADF. In order to provide the greatest portability (at least up to 64-bit environments), it is recommended that

- 8-byte data types (`I8`, `R8`) be aligned on 8 byte boundaries, and
- data types smaller than the word size be padded to a size equal to the word size.

So a 4-byte data type (e.g., `I4[1]`) needs "padding" (e.g., `I4[2]`) if it is to be followed by an 8-byte data type. And assuming a word size of 4 bytes, all `C1`-data-type elements need dimension values of multiples of 4 bytes (e.g., `C1[4]`, `C1[8]`, etc.). To be even more careful, size everything in multiples of 8 bytes, for example, "`C1[8]`, `I4[2]`, `C1[16]`, `R4[6]`, `R8[5]`, `I8[1]`".

For a given architecture and compiler, and taking into consideration the restrictions given above, compound data types should work. It is more portable and highly recommended that users write out the individual components of a structure into separate nodes. It would probably be best to copy the individual components of a list of structures into an appropriate array type and write the temporary array out using the write-all or write-strided routines.

F ADF Conventions and Implementations

C — All input strings are to be null terminated. All returned strings will have the trailing blanks removed and will be null terminated. Variables declared to hold Names, Labels, and Data-Types should be at least 33 characters long. *ADF.h* has a number of variables defined. An example declaration would be:

```
char name[ADF_NAME_LENGTH+1];
```

Fortran — Strings will be determined using inherited length. Returned strings will be blank filled to the specified length. All returned names will be left justified and blank filled on the right. There will be no null character. An example declaration would be:

```
PARAMETER ADF_NAME_LENGTH=32
CHARACTER*(ADF_NAME_LENGTH) NAME
```

ID — A unique identifier to access a given node within a file. This 8-byte field contains sufficient information for ADF to locate the node within a file. For any given node, the ID is generated only after the file it resides in has been opened by a program and the user requests information about the node. The ID is valid only within the program that opened the file and while that file is open. If the file is closed and reopened, the ID for any given node will be different. Within different programs, the node ID for the same node will be different. The ID is not ever actually written into a file.

The declaration for variables that will hold node IDs should be for an 8-byte real number.

The ID is actually a 64-bit combination of a system-generated file index along with the block and offset of the location of the node on the disk. In general, users do not need to know the internal coding of this information.

`error_return` — The error code for the ADF routines is the following:

-1	No error.
$n\ (> 0)$	ADF error code. The routine `ADF_Error_Message` is used to get a textual description of the error.
0	ADF should never return the value zero.

Indexing — All indexing is Fortran-like in that the starting index is 1 and the last is `N` for N items in an index or array dimension. The array structure is assumed to be the same as in Fortran with the first array dimension varying the fastest and the last dimension varying the slowest.

The index starting at one is used in `ADF_Read_Data`, `ADF_Write_Data`, and `ADF_Children_Names`.

The user should be aware of the differences in array indexing between Fortran and C. The subroutines `ADF_Read_All_Data` and `ADF_Write_All_Data` merely take a pointer to the beginning of the data, compute how much data is to be read/written, and process as many bytes as have been requested. Thus, these routines effectively make a copy of memory onto disk or vice versa. Given this convention, it is possible for a C program to use standard C conventions for array indexing and use `ADF_Write_All_Data` to store the array on disk. Then a Fortran program might use `ADF_Read_All_Data` to read the data set. Unless the user is aware of the structure of the data, it is possible for the array to be transposed relative to what is expected.

The implications of the assumed array structure convention can be quite subtle. The subroutines `ADF_Write_Data` and `ADF_Read_Data` assume the Fortran array structure in order to index the data. Again, unless the user is aware of the implications of this, it is possible to write an array on disk and later try to change a portion of the data and not change the correct numbers.

As long as users are aware of how their data structure maps onto ADF, there will not be any problems.

G ADF Error Messages

Table 3: ADF Error Messages

Number	Error Message
-1	No Error
1	Integer number is less than a given minimum value
2	Integer value is greater than given maximum value
3	String length of zero of blank string detected
4	String length longer than maximum allowable length
5	String length is not an ASCII-Hex string
6	Too many ADF files opened
7	ADF file status was not recognized
8	ADF file open error
9	ADF file not currently opened
10	ADF file index out of legal range
11	Block/offset out of legal range
12	A string pointer is null
13	FSEEK error
14	FWRITE error
15	FREAD error
16	Internal error: Memory boundary tag bad
17	Internal error: Disk boundary tag bad
18	File Open Error: NEW - File already exists[2]
19	ADF file format was not recognized
20	Attempt to free the RootNode disk information
21	Attempt to free the FreeChunkTable disk information
22	File Open Error: OLD - File does not exist[3]
23	Entered area of unimplemented code
24	Subnode entries are bad
25	Memory allocation failed
26	Duplicate child name under a parent node
27	Node has no dimensions
28	Node's number of dimensions is not in legal range
29	Specified child is not a child of the specified parent
30	Data-Type is too long
31	Invalid Data-Type
32	A pointer is null

Continued on next page

[2]The user is trying to create a new file and give it a name. The system has responded that the name has already been used.

[3]The user wants to open an existing file that supposedly has the given name. The system has responded that no file by that name exists.

Table 3: ADF Error Messages (*Continued*)

Number	Error Message
33	Node had no data associated with it
34	Error zeroing out of memory
35	Requested data exceeds actual data available
36	Bad end value
37	Bad stride values
38	Minimum value is greater than maximum value
39	The format of this machine does not match a known signature[4]
40	Cannot convert to or from an unknown native format
41	The two conversion formats are equal; no conversion done
42	The data format is not supported on a particular machine
43	File close error
44	Numeric overflow/underflow in data conversion
45	Bad start value
46	A value of zero is not allowable
47	Bad dimension value
48	Error state must be either a 0 (zero) or a 1 (one)
49	Dimensional specifications for disk and memory are unequal
50	Too many link levels are used; may be caused by a recursive link
51	The node is not a link. It was expected to be a link.
52	The linked-to node does not exist
53	The ADF file of a linked node is not accessible
54	A node ID of 0.0 is not valid
55	Incomplete data when reading multiple data blocks
56	Node name contains invalid characters
57	ADF file version incompatible with this library version
58	Nodes are not from the same file
59	Priority stack error
60	Machine format and file format are incomplete
61	Flush error

[4] When ADF wakes up, it tries to figure out what data format the machine it is running on uses (e.g., IEEE big endian, IEEE little endian, Cray). If it doesn't recognize the format, it can't convert files created on other platforms to the current one, so it issues this error message and punts.

H Default Values and Sizes and Limits of Dimensions and Arrays

The following default values, sizes, and limits are defined in the header file *ADF.h*.

Table 4: Default Values and Sizes

Attribute	Limit, Size, or Value
Data type length	32-byte character field
Date length	32-byte character field
File name length	1024-byte character field
Format length	20-byte character field
Label length	32-byte character field
Maximum link depth	100
Maximum dimension	12
Maximum length of error string	80 characters
Maximum link data size	4096
Name length	32-byte character field
Length of status	32-byte character field
Version length	32-byte character field
Maximum children	None
Pointer size	48-bit pointer is used for a block size of 4096
	32-bit pointer is used for blocks less than 4096
	16-bit pointer is used for blocks within blocks

I ADF Library of Subroutines

This appendix contains a listing of all the ADF routines for release 2.0. Each is described in detail for both C and Fortran use. There are examples, hints, and diagnostics included in some of the synopses and discussions of the routines.

I.1 Major Functional Groupings of Subroutines

The routines are summarized under the following major functional groupings:

Database-Level Routines

C	Fortran	Description	Page
ADF_Database_Close	ADFDCLO	Close an opened database	31
ADF_Database_Delete	ADFDDEL	Delete an existing database[5]	32
ADF_Database_Get_Format	ADFDGF	Get the data format in existing database	33
ADF_Database_Open	ADFDOPN	Open a database	29
ADF_Database_Set_Format	ADFDSF	Set the data format in an existing database	34

Data Structure and Management Routines

C	Fortran	Description	Page
ADF_Create	ADFCRE	Create a node	35
ADF_Delete	ADFDEL	Delete a node	39
ADF_Children_Names	ADFCNAM	Get the child names of a node	43
ADF_Number_of_Children	ADFNCLD	Get the number of children of a node	47
ADF_Get_Node_ID	ADFGNID	Get a unique identifier of a node	48
ADF_Get_Name	ADFGNAM	Get the name of a node	51
ADF_Put_Name	ADFPNAM	Put (change) the name of a node	52
ADF_Move_Child	ADFMOVE	Change a parent (move a child)	55
ADF_Link	ADFLINK	Create a link	59
ADF_Is_Link	ADFISLK	Test if a node is a link	66
ADF_Get_Link_Path	ADFGLKP	Get the path information from a link	68
ADF_Get_Root_ID	ADFGRID	Get the root ID of the ADF file	71

[5]Not implemented in the current release

Data Query Routines

C	Fortran	Description	Page
ADF_Get_Label	ADFGLB	Get a label	73
ADF_Set_Label	ADFSLB	Set a label	75
ADF_Get_Data_Type	ADFGDT	Get the data type	76
ADF_Get_Number_of_Dimensions	ADFGND	Get the number of dimensions	79
ADF_Get_Dimension_Values	ADFGDV	Get the dimension values of a node	80
ADF_Put_Dimension_Information	ADFPDIM	Set the data type and dimension information	81

Data I/O Routines

C	Fortran	Description	Page
ADF_Read_Data	ADFREAD	Read the data from a node (with partial capabilities)	83
ADF_Read_All_Data	ADFRALL	Read all the data into a contiguous memory space	90
ADF_Read_Block_Data	ADFRBLK	Read a contiguous block of data from a node	91
ADF_Write_Data	ADFWRIT	Write the data to a node (with partial capabilities)	92
ADF_Write_All_Data	ADFWALL	Write all the data from a contiguous memory space	100
ADF_Write_Block_Data	ADFWBLK	Write a contiguous block of data to a node	101

Miscellaneous Routines

C	Fortran	Description	Page
ADF_Flush_to_Disk	ADFFTD	Flush the data to a disk	102
ADF_Database_Garbage_Collection	ADFDGC	Garbage collection[6]	103
ADF_Error_Message	ADFERR	Return an error message	104
ADF_Set_Error_State	ADFSES	Set the error state	105
ADF_Get_Error_State	ADFGES	Get the error state	107
ADF_Database_Version	ADFDVER	Get the ADF file version ID	109
ADF_Library_Version	ADFLVER	Get the ADF library version ID	111

[6]Not implemented in the current release

I.2 Alphabetical Listing of Subroutines

All the routines are listed alphabetically below, according to their C name, along with the page number for each routine.

C	Fortran	Description	Page
ADF_Children_Names	ADFCNAM	Get the child names of a node	43
ADF_Create	ADFCRE	Create a node	35
ADF_Database_Close	ADFDCLO	Close an opened database	31
ADF_Database_Delete	ADFDDEL	Delete an existing database[7]	32
ADF_Database_Garbage_Collection	ADFDGC	Garbage collection[8]	103
ADF_Database_Get_Format	ADFDGF	Get the data format in existing database	33
ADF_Database_Open	ADFDOPN	Open a database	29
ADF_Database_Set_Format	ADFDSF	Set the data format in an existing database	34
ADF_Database_Version	ADFDVER	Get the ADF file version ID	109
ADF_Delete	ADFDEL	Delete a node	39
ADF_Error_Message	ADFERR	Return an error message	104
ADF_Flush_to_Disk	ADFFTD	Flush the data to a disk	102
ADF_Get_Data_Type	ADFGDT	Get the data type	76
ADF_Get_Dimension_Values	ADFGDV	Get the dimension values of a node	80
ADF_Get_Error_State	ADFGES	Get the error state	107
ADF_Get_Label	ADFGLB	Get a label	73
ADF_Get_Link_Path	ADFGLKP	Get the path information from a link	68
ADF_Get_Name	ADFGNAM	Get the name of a node	51
ADF_Get_Node_ID	ADFGNID	Get a unique identifier of a node	48
ADF_Get_Number_of_Dimensions	ADFGND	Get the number of dimensions	79
ADF_Get_Root_ID	ADFGRID	Get the root ID of the ADF file	71
ADF_Is_Link	ADFISLK	Test if a node is a link	66
ADF_Library_Version	ADFLVER	Get the ADF library version ID	111
ADF_Link	ADFLINK	Create a link	59
ADF_Move_Child	ADFMOVE	Change a parent (move a child)	55
ADF_Number_of_Children	ADFNCLD	Get the number of children of a node	47
ADF_Put_Dimension_Information	ADFPDIM	Set the data type and dimension information	81
ADF_Put_Name	ADFPNAM	Put (change) the name of a node	52
ADF_Read_All_Data	ADFRALL	Read all the data into a contiguous memory space	90
ADF_Read_Block_Data	ADFRBLK	Read a contiguous block of data from a node	91

Continued on next page

[7] Not implemented in the current release
[8] Not implemented in the current release

C	Fortran	Description	Page
ADF_Read_Data	ADFREAD	Read the data from a node (with partial capabilities)	83
ADF_Set_Error_State	ADFSES	Set the error state	105
ADF_Set_Label	ADFSLB	Set a label	75
ADF_Write_All_Data	ADFWALL	Write all the data from a contiguous memory space	100
ADF_Write_Block_Data	ADFWBLK	Write a contiguous block of data to a node	101
ADF_Write_Data	ADFWRIT	Write the data to a node (with partial capabilities)	92

I.3 Database-Level Routines

`ADF_Database_Open` — *Open a Database*

`ADF_Database_Open (filename,status,format,root_ID,error_return)`

Language	C	Fortran
Routine Name	`ADF_Database_Open`	`ADFDOPN`

Input	`const char *filename`
	`const char *status`
	`const char *format`
		`character*(*) filename`
		`character*(*) status`
		`character*(*) format`

Output	`double *root_ID`
	`int *error_return`
		`real*8 root_ID`
		`integer error_return`

filename A legal file name that may include a relative or absolute path where it is directly usable by the C `fopen()` system routine (no environment expansion is done).

status Similar to a Fortran `OPEN()` status. Input is required: there is no default. Allowable values are:

`READ_ONLY`	File must exist; writing is not allowed.
`OLD`	File must exist; reading and writing are allowed.
`NEW`	File must not exist.
`SCRATCH`	Temporary new file is created with a system name, and *filename* is ignored. The temporary file is deleted when the program exits or the file is closed.
`UNKNOWN`	`OLD` if file exists or else `NEW` is used.

format Specifies the numeric format for the file. This field is used only when a file is created and is ignored when *status* = `OLD`. Allowable values are:

`NATIVE`	Use the native numeric format of the computer that creates the file. This is the default for a new file if the input string for format is null. Note that if the native numeric format is not one of the supported formats listed here, then the file cannot be read on machines using any other format.
`IEEE_BIG`	Use the IEEE big endian format.
`IEEE_LITTLE`	Use the IEEE little endian format.
`CRAY`	Use the native CRAY format.

root_ID The root identity of the database.

error_return Error return code. (See Appendix G.)

This routine, `ADF_Database_Open`, opens a new or existing database. If links to other ADF files exist in the current file, they will be opened only as required. Using this routine is similar to opening a file in Fortran with the corresponding clarifiers, such as whether it is `READ_ONLY`, `OLD`, `NEW`, or named as `SCRATCH` file.

The format of the file, which is ignored when the status of the file is `OLD`, is used when the file is first created. Big endian is a binary format in which the most significant byte or bit comes first, whereas in little endian, the most significant byte or bit comes last. To specify the format more explicitly, you can use the following formats:

```
IEEE_BIG_32
IEEE_BIG_64
IEEE_LITTLE_32
IEEE_LITTLE_64
```

for IEEE big or little endian formats.

Example

This example opens a new database using the native format of the host computer. Note that the default format is specified by using the empty string. In the C programming language, a null string could have been used.

```
PROGRAM TEST
   CHARACTER*(80) MSG
   REAL*8 RID
   INTEGER IERR
   CALL ADFDOPN('db.adf','NEW',' ',RID,IERR)
   IF (IERR .GT. 0) THEN
      CALL ADFERR(IERR,MSG)
      PRINT *,MSG
      STOP
   ENDIF
   STOP
   END
```

`ADF_Database_Close` — *Close a Database*

`ADF_Database_Close (root_ID,error_return))`		
Language	**C**	**Fortran**
Routine Name	`ADF_Database_Close`	`ADFDCLO`
Input	`const double *root_ID`	`real*8 root_ID`
Output	`int *error_return`	`integer error_return`

root_ID	The root identification of the database. This can be a valid node ID for this database.
error_return	Error return code. (See Appendix G.)

This routine, `ADF_Database_Close`, closes an existing database, as well as the other ADF files that may be attached through links. For example, if there is another ADF file that is opened and linked to this database, only the file stream associated with this database will be closed. This routine is similar to the close of a file in Fortran.

Example

This example closes a database. Note that while the root ID is used in the call to `ADFDCLO`, any valid node ID for this file will work. Also, in general, it is not necessary to close open ADF files when the program exits normally. The standard shutdown procedures will flush all buffers and bring files up to date. The primary use of `ADFDCLO` is to clean up file tables or to release unused files.

```
PROGRAM TEST
   CHARACTER*(80) MSG
   REAL*8 RID
   INTEGER IERR
   CALL ADFDOPN('db.adf','NEW',' ',RID,IERR)
   IF (IERR .GT. 0) THEN
      CALL ADFERR(IERR,MSG)
      PRINT *,MSG
      STOP
   ENDIF
   .
   ...do useful stuff (hopefully)
   .
   CALL ADFDCLO(RID,IERR)
   STOP
   END
```

`ADF_Database_Delete` — *Delete a File*

`ADF_Database_Delete (filename,error_return))`		
Language	**C**	**Fortran**
Routine Name	`ADF_Database_Delete`	`ADFDDEL`
Input	`const char *filename`	`character*(*) filename`
Output	`int *error_return`	`integer error_return`

filename	A legal file name of an existing ADF database. The filename may include a relative or absolute path where it is directly usable by the C `fopen()` system routine (no environment expansion is done).
error_return	Error return code. (See Appendix G.)

This routine, `ADF_Database_Delete`, deletes an existing database file. It does not delete links referenced in the database. This routine is similar to the deletion of a file in Fortran.

Note: This routine will be implemented in a future release.

`ADF_Database_Get_Format` — *Get the Data Format*

`ADF_Database_Get_Format (root_ID,format,error_return))`

Language	C	Fortran
Routine Name	`ADF_Database_Get_Format`	`ADFDGF`
Input	`const double *root_ID`	`real*8 root_ID`

Output	`char *format`
	`int *error_return`
		`character*(*) format`
		`integer error_return`

root_ID	Any valid node ID for the given ADF file.
format	The format for the file.
error_return	Error return code. (See Appendix G.)

This routine, `ADF_Database_Get_Format`, gets the data format for an existing database.

Example

This example opens an existing ADF database, creates a new node, and then uses the node ID for the new node to ask what the file type is. Note that the file *format* is ignored because the database already exists.

```
PROGRAM TEST
   CHARACTER*(80) MSG
   CHARACTER*(32) FORM
   REAL*8 RID,CID
   INTEGER IERR
   CALL ADFDOPN('db.adf','OLD',' ',RID,IERR)
   IF (IERR .GT. 0) THEN
      CALL ADFERR(IERR,MSG)
      PRINT *,MSG
      STOP
   ENDIF
   CALL ADFCRE(RID,'junk_node',CID,IERR)
   CALL ADFDGF(CID,FORM,IERR)
   PRINT *,'FILE FORMAT = ',FORM
   STOP
   END
```

`ADF_Database_Set_Format` — *Set the Data Format*

`ADF_Database_Set_Format (root_ID,format,error_return))`

Language	**C**	**Fortran**
Routine Name	`ADF_Database_Set_Format`	`ADFDSF`
Input	`const double *root_ID`	`real*8 root_ID`

Output	`char *format`
	`int *error_return`
		`character*(*) format`
		`integer error_return`

root_ID	The root identity of the database.
format	The numeric format for the file.
error_return	Error return code. (See Appendix G.)

This routine, **`ADF_Database_Set_Format`**, sets the data format in an existing database.

Note: Use with extreme caution. This routine is needed only for the data conversion utilities and not intended for the general user.

I.4 Data Structure and Management Routines

`ADF_Create` — *Create a Node*

ADF_Create (PID,name,ID,error_return)		
Language	**C**	**Fortran**
Routine Name	`ADF_Create`	`ADFCRE`

Input	`const double PID`
	`const char *name`
		`real*8 PID`
		`character*(*) name`

Output	`double *ID`
	`int *error_return`
		`real*8 ID`
		`integer error_return`

PID	ID of the parent node of the created child node.
name	Name of the parent node.
ID	The ID of the newly created child node.
error_return	Error return code. (See Appendix G.)

This routine, `ADF_Create`, creates a new node (not a link) as a child of a given parent node.

Default node header values in this new node are:

- label = blank
- number of subnodes = 0
- datatype = `MT`
- number of dimensions = 0
- data = `NULL`

Example

This example opens a database and creates a node under the root node. Note that the default values for a newly created node are label = `' '`, datatype = `MT`, dimension = `null`, data = none. These are reset as required using the routines `ADFSLB`, `ADFPDIM` and `ADFWALL`/`ADFWRIT`. Note also that the root node is named "`ADF MotherNode`". This name is generated when the database is first opened. If desired, it could be reset using `ADFPNAM`.

```
PROGRAM TEST
C
      PARAMETER (MAXCHR=32)
C
      CHARACTER*(MAXCHR) NODNAM
C
C *** NODE IDS
C
      REAL*8 RID,PID,CID
      INTEGER I,J,IERR,NUMCLD
```

```
C
C *** OPEN DATABASE
C
      CALL ADFDOPN('db.adf','NEW',' ',RID,IERR)
C
C *** CREATE NODES AT FIRST LEVEL
C
      DO 150 I = 1,3
         WRITE(NODNAM,'(A7,I1)')'PARENT.',I
         CALL ADFCRE(RID,NODNAM,PID,IERR)
C
C ****** CREATE NODES AT SECOND LEVEL
C
         NUMCLD = I*I
         DO 110 J = 1,NUMCLD
            WRITE(NODNAM,'(A6,I1,A1,I1)')'CHILD.',I,'x',J
            CALL ADFCRE(PID,NODNAM,CID,IERR)
  110    CONTINUE
C
C ****** PRINT NODE NAMES JUST CREATED
C
         CALL PRTCLD(PID)
  150 CONTINUE
C
C *** PRINT NAMES OF NODES ATTACHED TO ROOT NODE
C
      CALL PRTCLD(RID)
C
      STOP
      END
C
C ************* SUBROUTINES ****************
C
      SUBROUTINE ERRCHK(IERR)
C
C *** CHECK ERROR CONDITION
C
      CHARACTER*80 MESS
      IF (IERR .GT. 0) THEN
         CALL ADFERR(IERR,MESS)
         PRINT *,MESS
         CALL ABORT('ADF ERROR')
      ENDIF
      RETURN
      END

      SUBROUTINE PRTCLD(PID)
C
C *** PRINT TABLE OF CHILDREN GIVEN A PARENT NODE-ID
C
      PARAMETER (MAXCLD=10)
      PARAMETER (MAXCHR=32)
      REAL*8 PID
      CHARACTER*(MAXCHR) NODNAM,NDNMS(MAXCLD)
      CALL ADFGNAM(PID,NODNAM,IERR)
      CALL ERRCHK(IERR)
      CALL ADFNCLD(PID,NUMC,IERR)
      CALL ERRCHK(IERR)
      WRITE(*,120)NODNAM,NUMC
  120 FORMAT(/,' PARENT NODE NAME = ',A,/,
     X       '     NUMBER OF CHILDREN = ',I2,/,
     X       '     CHILDREN NAMES: ')
      NLEFT = NUMC
      ISTART = 1
C     --- TOP OF DO-WHILE LOOP
  130 CONTINUE
         CALL ADFCNAM(PID,ISTART,MAXCLD,LEN(NDNMS),
     X                NUMRET,NDNMS,IERR)
         CALL ERRCHK(IERR)
         WRITE(*,140)(NDNMS(K),K=1,NUMRET)
  140    FORMAT(2(8X,A))
         NLEFT = NLEFT - MAXCLD
         ISTART = ISTART + MAXCLD
      IF (NLEFT .GT. 0) GO TO 130
      RETURN
      END
```

The resulting output is:

```
PARENT NODE NAME = PARENT.1
    NUMBER OF CHILDREN =  1
    CHILDREN NAMES:
       CHILD.1.1

PARENT NODE NAME = PARENT.2
    NUMBER OF CHILDREN =  4
    CHILDREN NAMES:
       CHILD.2.1                       CHILD.2.2
       CHILD.2.3                       CHILD.2.4

PARENT NODE NAME = PARENT.3
    NUMBER OF CHILDREN =  9
    CHILDREN NAMES:
       CHILD.3.1                       CHILD.3.2
       CHILD.3.3                       CHILD.3.4
       CHILD.3.5                       CHILD.3.6
       CHILD.3.7                       CHILD.3.8
       CHILD.3.9

PARENT NODE NAME = ADF MotherNode
    NUMBER OF CHILDREN =  3
    CHILDREN NAMES:
       PARENT.1                                PARENT.2
       PARENT.3
```

ADF_Delete — *Delete a Node*

ADF_Delete (PID,ID,error_return)		
Language	**C**	**Fortran**
Routine Name	ADF_Delete	ADFDEL
Input	const double PID const double ID	real*8 PID real*8 ID
Output	int *error_return	integer error_return

PID	The ID of the node's parent.
ID	The ID of the node to use.
error_return	Error return code. (See Appendix G.)

In general, this routine, `ADF_Delete`, deletes a node and all of its children. Given the starting node, a recursive search is done down the hierarchy, deleting all nodes. If a "link" is encountered during the deletion (i.e., the specified node or any of its children), then the link is deleted, and the downward search stops. That is, the link information is deleted, but not the actual node it refers to.

To understand the deletion of a node that is a link, it must be remembered that a link is merely a reference to another node. Therefore, the deletion of a node that is a link is the deletion of that reference, not the referred node itself. The reason for this is that a link may actually point to data in another file that may be owned by another user. Therefore, it would not be proper for ADF to try to delete that node. Therefore, ADF stops at the link.

When a node is deleted, any links that reference it are left "dangling." In other words, the existing links to the node still reference the node, but if ADF is asked to resolve that reference, it will determine that the referred to node doesn't exist and will return an error flag.

Note that the parent ID of the node to be deleted is required. This is due to the fact that child nodes do not know the ID of their parent node. Thus when a node is deleted, in order for the child table of the parent to be updated properly, the parent ID must be supplied as an input.

Example

This example opens a database and creates three nodes attached to the root node. It also generates nodes to each of these base nodes. Then one of the base nodes is deleted. Not only is the node "`PARENT.2`" deleted; all of its children are deleted at the same time.

```
PROGRAM TEST
C
      PARAMETER (MAXCHR=32)
C
      CHARACTER*(MAXCHR) NODNAM
C
C *** NODE IDS
C
      REAL*8 RID,PID,CID
      INTEGER I,J,IERR,NUMCLD
C
C *** OPEN DATABASE
C
      CALL ADFDOPN('db.adf','NEW',' ',RID,IERR)
C
C *** CREATE NODES AT FIRST LEVEL
C
      DO 150 I = 1,3
         WRITE(NODNAM,'(A7,I1)')'PARENT.',I
         CALL ADFCRE(RID,NODNAM,PID,IERR)
C
C ****** CREATE NODES AT SECOND LEVEL
C
         NUMCLD = I*I
         DO 110 J = 1,NUMCLD
            WRITE(NODNAM,'(A6,I1,A1,I1)')'CHILD.',I,'.',J
            CALL ADFCRE(PID,NODNAM,CID,IERR)
 110     CONTINUE
 150  CONTINUE
C
C *** PRINT NAMES OF NODES ATTACHED TO ROOT NODE
C
      CALL PRTCLD(RID)
C
C *** PRINT NAMES OF CHILDREN UNDER PARENT.2
C
      CALL ADFGNID(RID,'PARENT.2',PID,IERR)
      CALL PRTCLD(PID)
C
C *** NOW DELETE PARENT.2
C
      CALL ADFDEL(RID,PID,IERR)
      CALL PRTCLD(RID)
C
C *** JUST FOR GRINS, LOOK FOR CHILDREN UNDER ORIGINAL ID
C
      CALL ADFGNID(RID,'/PARENT.2/CHILD.2.1',CID,IERR)
      CALL ERRCHK(IERR)
C
      STOP
      END
C
C ************* SUBROUTINES ***************
C
      SUBROUTINE ERRCHK(IERR)
C
C *** CHECK ERROR CONDITION
C
      CHARACTER*80 MESS
      IF (IERR .GT. 0) THEN
```

```
         CALL ADFERR(IERR,MESS)
         PRINT *,MESS
         CALL ABORT('ADF ERROR')
      ENDIF
      RETURN
      END

      SUBROUTINE PRTCLD(PID)
C
C *** PRINT TABLE OF CHILDREN GIVEN A PARENT NODE-ID
C
      PARAMETER (MAXCLD=10)
      PARAMETER (MAXCHR=32)
      REAL*8 PID
      CHARACTER*(MAXCHR) NODNAM,NDNMS(MAXCLD)
      CALL ADFGNAM(PID,NODNAM,IERR)
      CALL ERRCHK(IERR)
      CALL ADFNCLD(PID,NUMC,IERR)
      CALL ERRCHK(IERR)
      WRITE(*,120)NODNAM,NUMC
  120 FORMAT(/,' PARENT NODE NAME = ',A,/,
     X       '     NUMBER OF CHILDREN = ',I2,/,
     X       '     CHILDREN NAMES:')
      NLEFT = NUMC
      ISTART = 1
C     --- TOP OF DO-WHILE LOOP
  130 CONTINUE
         CALL ADFCNAM(PID,ISTART,MAXCLD,LEN(NDNMS),
     X                NUMRET,NDNMS,IERR)
         CALL ERRCHK(IERR)
         WRITE(*,140)(NDNMS(K),K=1,NUMRET)
  140    FORMAT(2(8X,A))
         NLEFT = NLEFT - MAXCLD
         ISTART = ISTART + MAXCLD
      IF (NLEFT .GT. 0) GO TO 130
      RETURN
      END
```

The resulting output is:

```
PARENT NODE NAME = ADF MotherNode
    NUMBER OF CHILDREN = 3
    CHILDREN NAMES:
       PARENT.1                        PARENT.2
       PARENT.3

PARENT NODE NAME = PARENT.2
    NUMBER OF CHILDREN = 4
    CHILDREN NAMES:
       CHILD.2.1                       CHILD.2.2
       CHILD.2.3                       CHILD.2.4
```

```
 PARENT NODE NAME = ADF MotherNode
     NUMBER OF CHILDREN = 2
     CHILDREN NAMES:
        PARENT.1                                PARENT.3

 ADF 29: Specified child is NOT a child of the specified parent.
IOT Trap
Abort - core dumped
```

ADF_Children_Names — *Get the Names of the Child Nodes*

```
ADF_Children_Names (PID,istart,imax_num,imax_name_length,inum_ret,names,
                    error_return)
```

Language	**C**	**Fortran**
Routine Name	`ADF_Children_Names`	`ADFCNAM`

Input	`const double PID`
	`const int istart`
	`const int imax_num`
	`const int imax_name_length`
		`real*8 PID`
		`integer istart`
		`integer imax_num`
		`integer imax_name_length`

Output	`int *inum_ret`
	`char *names`
	`int *error_return`
		`integer inum_ret`
		`character*(*) names`
		`integer error_return`

PID	The ID of the parent node to use.
istart	The *n*th child's name (to start with the first, use *istart* = 1).
imax_num	The maximum number of names to return.
imax_name_length	The number of characters allocated to hold the name of each child node.
inum_ret	The number of names returned.
names	The names of the children.
error_return	Error return code. (See Appendix G.)

This routine, `ADF_Children_Names`, returns the child names directly associated with a parent node. The names of the children are not guaranteed to be returned in any particular order. For example, if four child nodes were created in the order: `node1`, `node2`, `node3`, `node4`, when the call to `ADFCNAM` is made, there is no guarantee that the order of the node names in the character array names will be the same.

The reason for not guaranteeing node ordering has to do with efficient use of disk space. Although the concept of "linked lists" works fine in central memory, it is not particularly efficient on disk. Therefore, static tables are used to maintain parent/child lists. The order in which children names are returned is the order found in the static table. If a child node is deleted, an empty slot is created and will be used by the next child node created under that parent node.

The indexing of a child is Fortran-like and begins with 1, but as noted above, this does not imply a notion of node ordering. To start with a child node listed as the first index, use an *istart* value of 1.

C Programming Notes

- Node names can be up to 32 characters. ADF appends the null (`\0`) character to the end of each node name. Therefore, at least 33 characters should be allocated for each node name to be returned.

- *imax_name_length* is used to "stride" through the character array passed to ADF. For example, if 50 characters per name were allocated, the first character of the first name would be in the first allocated byte (position [0]), and the second name would start in the 51st byte (position [50]) and so on. The length of the individual names can be determined using the `strlen` function.

Fortran Programming Notes

- Node names can be up to 32 characters. The returned names are left justified and blank filled within the array. No null (`\0`) character is appended to the name; therefore, an appropriate declaration would be `CHARACTER*(32)`.

Example

This example creates three nodes attached to the root node. It then creates a varying number of child nodes under each of the base nodes. Lastly, it queries the database to find out how many children were created and then gets the child names. The thing to notice in this example is that the character array `NDNMS` is not large enough to hold all the names under `PARENT.3`. The array `NDNMS` will hold only five names at a time; therefore, a loop was set up to read a subset of the name list during each pass. Note also that the call to `ADFCNAM` may request more names than are present. If this occurs, the routine will return all that is available in the output array *names* and return that number in the *inum_ret* variable.

```
PROGRAM TEST
C
      PARAMETER (MAXCLD=5)
      PARAMETER (NDATA=10)
      PARAMETER (MAXCHR=32)
C
      CHARACTER*(MAXCHR) NODNAM,NDNMS(MAXCLD)
C
C     RID - ROOT ID
C     CID - CHILD ID
C     PID - PARENT ID
C
      REAL*8 RID,CID,PID
      INTEGER I,J,K,IERR,NUMCLD,NLEFT,ISTART
C
C *** OPEN DATABASE
C
      CALL ADFDOPN('DB.ADF','NEW',' ',RID,IERR)
      CALL ERRCHK(IERR)
C
C *** CREATE NODES AT FIRST LEVEL
C
      DO 150 I = 1,3
         WRITE(NODNAM,'(A7,I1)')'PARENT.',I
         CALL ADFCRE(RID,NODNAM,PID,IERR)
         CALL ERRCHK(IERR)
C
C ****** CREATE NODES AT SECOND LEVEL
```

```
C
         NUMCLD = I*I
         DO 110 J = 1,NUMCLD
            WRITE(NODNAM,'(A6,I1,A1,I1)')'CHILD.',I,'.',J
            CALL ADFCRE(PID,NODNAM,CID,IERR)
            CALL ERRCHK(IERR)
  110    CONTINUE
C
C ****** GET NUMBER AND NAMES OF CHILDREN JUST CREATED
C        AND PRINT THEM OUT
C
         CALL ADFGNAM(PID,NODNAM,IERR)
         CALL ERRCHK(IERR)
         CALL ADFNCLD(PID,NUMC,IERR)
         CALL ERRCHK(IERR)
         WRITE(*,120)I,NODNAM,NUMC
  120    FORMAT(' LEVEL = ',I2,' PARENT NODE NAME = ',A,/,
     X          '     NUMBER OF CHILDREN = ',I2,/,
     X          '     CHILDREN NAMES:')
         NLEFT = NUMC
         ISTART = 1
C        --- TOP OF DO-WHILE LOOP
  130    CONTINUE
            CALL ADFCNAM(PID,ISTART,MAXCLD,LEN(NDNMS),
     X                   NUMRET,NDNMS,IERR)
            CALL ERRCHK(IERR)
            PRINT *,' FETCHED: ',NUMRET,' NAMES'
            WRITE(*,140)(NDNMS(K),K=1,NUMRET)
  140       FORMAT(8X,A)
            NLEFT = NLEFT - MAXCLD
            ISTART = ISTART + MAXCLD
         IF (NLEFT .GT. 0) GO TO 130
  150 CONTINUE
      STOP
      END
C
C ************* SUBROUTINES ****************
C
      SUBROUTINE ERRCHK(IERR)
      CHARACTER*80 MESS
      IF (IERR .GT. 0) THEN
         CALL ADFERR(IERR,MESS)
         PRINT *,MESS
         CALL ABORT('ADF ERROR')
      ENDIF
      RETURN
      END
```

The resulting output is:

```
PARENT = 1 PARENT NODE NAME = PARENT.1
```

```
      NUMBER OF CHILDREN =  1
      CHILDREN NAMES:
       FETCHED            1 NAMES
         CHILD.1.1

  PARENT = 2 PARENT NODE NAME = PARENT.2
      NUMBER OF CHILDREN =  4
      CHILDREN NAMES:
       FETCHED            4 NAMES
         CHILD.2.1
         CHILD.2.2
         CHILD.2.3
         CHILD.2.4

  PARENT = 3 PARENT NODE NAME = PARENT.3
      NUMBER OF CHILDREN =  9
      CHILDREN NAMES:
       FETCHED            5 NAMES
         CHILD.3.1
         CHILD.3.2
         CHILD.3.3
         CHILD.3.4
         CHILD.3.5
       FETCHED            4 NAMES
         CHILD.3.6
         CHILD.3.7
         CHILD.3.8
         CHILD.3.9
```

`ADF_Number_of_Children` — *Get the Number of Children Nodes*

ADF_Number_of_Children (PID,num_children,error_return)		
Language	**C**	**Fortran**
Routine Name	`ADF_Number_of_Children`	`ADFNCLD`
Input	`const double PID`	`real*8 PID`

Output	`int *num_children`
	`int *error_return`
		`integer num_children`
		`integer error_return`

PID	The ID of the parent node to use.
num_children	The number of children directly associated with this node.
error_return	Error return code. (See Appendix G.)

This routine, `ADF_Number_of_Children`, returns the number of child names directly associated with a parent node.

Example

See the example for `ADF_Children_Names`.

`ADF_Get_Node_ID` — *Get the ID of a Child Node*

`ADF_Get_Node_ID (PID,name,ID,error_return)`		
Language	**C**	**Fortran**
Routine Name	`ADF_Get_Node_ID`	`ADFGNID`

Input	`const double PID`
	`const char *name`
		`real*8 PID`
		`character*(*) name`

Output	`double *ID`
	`int *error_return`
		`real*8 ID`
		`integer error_return`

PID	The ID of the parent node.
name	The name of the node.
ID	The ID of the named node.
error_return	Error return code. (See Appendix G.)

This routine, `ADF_Get_Node_ID`, returns the ID of a child node, given the parent node ID and the name of the child node. To return the ID of the root node in an ADF file, use any known ID in the ADF file and a name of "/". The syntax for *name* is essentially the same as the "path name" within the UNIX operating system. It might look like `/level.1/level.2/node`. The name of the node may be one of two forms. If the name begins with a "/", then the name is relative to the root node for the associated database. If the name does not begin with a "/", then the name is relative to the parent node associated with the given PID.

To return the ID of the root node an ADF file, use any known ID in the ADF file and a name of "/".

Example

This example illustrates the various ways to access a node. Note that when a full path is specified (i.e., a leading slash "/" is specified), all that is required for the ID is any valid ID for the associated database. It will probably be clearer to others if the root ID is used in that situation.

```
PROGRAM TEST
C
      PARAMETER (MAXCHR=32)
C
      CHARACTER*(MAXCHR) NODNAM
C
C     RID - ROOT ID
C     AL1ID - LEVEL 1 ID
C     AL2ID - LEVEL 2 ID
C     AL3ID - LEVEL 3 ID
C     CID - CHILD ID
C
      REAL*8 RID,AL1ID,AL2ID,AL3ID,CID
```

```
C
C *** OPEN DATABASE
C
      CALL ADFDOPN('DB.ADF','NEW',' ',RID,IERR)
      CALL ERRCHK(IERR)
C
C *** CREATE NODE AT FIRST LEVEL
C
      CALL ADFCRE(RID,'LEVEL.1',AL1ID,IERR)
      CALL ERRCHK(IERR)
C
C *** CREATE NODE AT SECOND LEVEL
C
      CALL ADFCRE(AL1ID,'LEVEL.2',AL2ID,IERR)
      CALL ERRCHK(IERR)
C
C *** CREATE NODE AT THIRD LEVEL
C
      CALL ADFCRE(AL2ID,'LEVEL.3',AL3ID,IERR)
      CALL ERRCHK(IERR)
C
C *** EQUIVALENT WAYS TO GET THE LOWER LEVEL NODE ID
C
C ****** FULL PATH NAME
C
      CALL ADFGNID(RID,'/LEVEL.1/LEVEL.2/LEVEL.3',CID,IERR)
      CALL ERRCHK(IERR)
      CALL ADFGNAM(CID,NODNAM,IERR)
      PRINT *,' '
      PRINT *,' FULL PATH EXAMPLE: ROOT NODE ID: NODE NAME = ',NODNAM
C
C ****** FULL PATH NAME - GIVEN ANY VALID NODE ID FOR FILE
C
      CALL ADFGNID(AL3ID,'/LEVEL.1/LEVEL.2/LEVEL.3',CID,IERR)
      CALL ERRCHK(IERR)
      CALL ADFGNAM(CID,NODNAM,IERR)
      PRINT *,' '
      PRINT *,' FULL PATH EXAMPLE - VALID NODE ID: NODE NAME = ',NODNAM
C
C ****** PARTIAL PATH NAME
C
      CALL ADFGNID(AL1ID,'LEVEL.2/LEVEL.3',CID,IERR)
      CALL ERRCHK(IERR)
      CALL ADFGNAM(CID,NODNAM,IERR)
      PRINT *,' '
      PRINT *,' PARTIAL PATH EXAMPLE: NODE NAME = ',NODNAM
C
C ****** DIRECT USE OF PARENT ID
C
      CALL ADFGNID(AL2ID,'LEVEL.3',CID,IERR)
      CALL ERRCHK(IERR)
      CALL ADFGNAM(CID,NODNAM,IERR)
      PRINT *,' '
      PRINT *,' GIVEN PARENT NAME EXAMPLE: NODE NAME = ',NODNAM
      STOP
      END
C
C ************* SUBROUTINES ****************
C
      SUBROUTINE ERRCHK(IERR)
      CHARACTER*80 MESS
      IF (IERR .GT. 0) THEN
         CALL ADFERR(IERR,MESS)
         PRINT *,MESS
         CALL ABORT('ADF ERROR')
      ENDIF
      RETURN
      END
```

The resulting output is:

```
FULL PATH EXAMPLE: ROOT NODE ID: NODE NAME = LEVEL.3

FULL PATH EXAMPLE - VALID NODE ID: NODE NAME = LEVEL.3

PARTIAL PATH EXAMPLE: NODE NAME = LEVEL.3

GIVEN PARENT NAME EXAMPLE: NODE NAME = LEVEL.3
```

`ADF_Get_Name` — *Get the Name of a Node*

`ADF_Get_Name (ID,name,error_return)`

Language	**C**	**Fortran**
Routine Name	`ADF_Get_Name`	`ADFGNAM`
Input	`const double ID`	`real*8 ID`

Output	`char *name`
	`int *error_return`
		`character*(*) name`
		`integer error_return`

ID	The ID of the node to use.
name	The simple name of the node.
error_return	Error return code. (See Appendix G.)

This routine, `ADF_Get_Name`, returns the 32-character name of a node, given the node's ID. In C, the name will be null terminated after the last nonblank character; therefore, 33 characters should be used (32 for the name, plus 1 for the null). In Fortran, the null character is not returned; therefore, the character variable declaration for name should be for 32 characters (e.g., `CHARACTER*(32) NAME`).

Example

See the example for `ADF_Get_Node_ID`.

`ADF_Put_Name` — *Put a Name on a Node*

`ADF_Put_Name (PID,ID,name,error_return)`

Language	**C**	**Fortran**
Routine Name	`ADF_Put_Name`	`ADFPNAM`

Input	`const double PID`
	`const double ID`
	`const char *name`
		`real*8 PID`
		`real*8 ID`
		`character*(*) name`

Output	`int *error_return`	`integer error_return`

PID	The ID of the node's parent.
ID	The ID of the node to use.
name	The new name of the node.
error_return	Error return code. (See Appendix G.)

This routine, `ADF_Put_Name`, changes the name of a node.

Note that the parent ID of the node to be deleted is required. This is due to the fact that child nodes do not know the ID of their parent node. Thus, when a node is deleted, in order for the child table of the parent to be updated properly, the parent ID must be supplied as an input.

Warning: If the node is pointed to by a link node, changing the node's name will break the link.

Example

This example illustrates the creation of a node with an initial name. Later, the name of the node is changed. The routine `ADFCNAM` is used to get the new name from the parent's information table.

```
PROGRAM TEST
C
      PARAMETER (MAXCHR=32)
C
      CHARACTER*(MAXCHR) NODNAM,CLDNAM
C
C     RID - ROOT ID
C     CID - CHILD ID
C
      REAL*8 RID,AL1ID,AL2ID,AL3ID,CID
C
C *** OPEN DATABASE
C
      CALL ADFDOPN('db.adf','NEW',' ',RID,IERR)-*9+
C
      CALL ERRCHK(IERR)
C
C *** CREATE NODE
```

```
C
      CALL ADFCRE(RID,'LEVEL.1',CID,IERR)
      CALL ERRCHK(IERR)
C
C *** GET NODE NAME AND CHECK PARENTS TABLE
C
      CALL ADFGNAM(CID,NODNAM,IERR)
      CALL ERRCHK(IERR)
      PRINT *,' '
      PRINT *,' NODE NAME = ',NODNAM
      CALL ADFCNAM(RID,1,1,LEN(CLDNAM),
     X             NUMRET,CLDNAM,IERR)
      CALL ERRCHK(IERR)
      PRINT *,' NODE NAME IN PARENTS TABLE = ',CLDNAM
C
C *** CHANGE THE NODE NAME
C
      CALL ADFPNAM(RID,CID,'NEW_NAME',IERR)
      CALL ERRCHK(IERR)
C
C *** GET NEW NODE NAME AND CHECK PARENTS TABLE
C
      CALL ADFGNAM(CID,NODNAM,IERR)
      CALL ERRCHK(IERR)
      PRINT *,' '
      PRINT *,' NEW NODE NAME = ',NODNAM
      CALL ADFCNAM(RID,1,1,LEN(CLDNAM),
     X             NUMRET,CLDNAM,IERR)
      CALL ERRCHK(IERR)
      PRINT *,' NODE NAME IN PARENTS TABLE = ',CLDNAM
      STOP
      END
C
C ************* SUBROUTINES ****************
C
      SUBROUTINE ERRCHK(IERR)
      CHARACTER*80 MESS
      IF (IERR .GT. 0) THEN
         CALL ADFERR(IERR,MESS)
         PRINT *,MESS
         CALL ABORT('ADF ERROR')
      ENDIF
      RETURN
      END
```

The resulting output is:

```
NODE NAME     = LEVEL.1
NODE NAME IN PARENTS TABLE = LEVEL.1

NEW NODE NAME = NEW_NAME
```

```
   NODE NAME IN PARENTS TABLE = NEW_NAME
```

`ADF_Move_Child` — *Move a Child Node to a Different Parent*

`ADF_Move_Child (PID,ID,NPID,error_return)`

Language	C	Fortran
Routine Name	`ADF_Move_Child`	`ADFMOVE`

Input	`const double PID`
	`const double ID`
	`double NPID`
		`real*8 PID`
		`real*8 ID`
		`real*8 NPID`

Output	`int *error_return`	`integer error_return`

PID	ID of the node's current parent.
ID	ID of the node to use.
NPID	ID of the node's new parent.
error_return	Error return code. (See Appendix G.)

This routine, `ADF_Move_Child`, deletes the given child node from the current parent's child table and adds it to the new parent's child table. `ADF_Move_Child` is restricted to moves within the same physical file. If the node is pointed to by a link-node, moving the node's name will break the link.

Example

This example creates a simple hierarchy. It then picks up a node from its original parent and moves it to a new parent node.

```
PROGRAM TEST
C
      PARAMETER (MAXCHR=32)
C
      CHARACTER*(MAXCHR) NODNAM
C
C *** NODE IDS
C
      REAL*8 RID,PID,CID,PID1,PID3
      INTEGER I,J,IERR
C
C *** OPEN DATABASE
C
      CALL ADFDOPN('db.adf','NEW',' ',RID,IERR)
      CALL ERRCHK(IERR)
C
C *** CREATE NODES AT FIRST LEVEL
C
      WRITE(*,100)
  100 FORMAT(/,' *** ORIGINAL DATABASE SETUP ***')
      DO 150 I = 1,3
```

```
         WRITE(NODNAM,'(A7,I1)')'PARENT.',I
         CALL ADFCRE(RID,NODNAM,PID,IERR)
         CALL ERRCHK(IERR)
C
C ****** CREATE NODES AT SECOND LEVEL
C
         NUMCLD = I*I
         DO 110 J = 1,NUMCLD
            WRITE(NODNAM,'(A6,I1,A1,I1)')'CHILD.',I,'.',J
            CALL ADFCRE(PID,NODNAM,CID,IERR)
            CALL ERRCHK(IERR)
  110    CONTINUE
C
C ****** GET NUMBER AND NAMES OF CHILDREN JUST CREATED
C        AND PRINT THEM OUT
C
         CALL PRTCLD(PID)
  150 CONTINUE
C
C *** PICK UP NODE /PARENT.3/CHILD.3.4 AND MOVE IT
C     TO /PARENT.1
C
      CALL ADFGNID(RID,'PARENT.3',PID3,IERR)
      CALL ERRCHK(IERR)
C
      CALL ADFGNID(PID3,'CHILD.3.4',CID,IERR)
      CALL ERRCHK(IERR)
C
      CALL ADFGNID(RID,'PARENT.1',PID1,IERR)
      CALL ERRCHK(IERR)
C
      CALL ADFMOVE(PID3,CID,PID1,IERR)
      CALL ERRCHK(IERR)
C
C *** CHECK TO MAKE SURE THE NODE WAS ACTUALLY MOVED
C
      WRITE(*,160)
  160 FORMAT(/,'*** PARENT.1 AND PARENT.3 AFTER MOVE ***')
      CALL PRTCLD(PID1)
      CALL PRTCLD(PID3)
C
      STOP
      END
C
C ************* SUBROUTINES ****************
C
      SUBROUTINE ERRCHK(IERR)
C
C *** CHECK ERROR CONDITION
C
      CHARACTER*80 MESS
      IF (IERR .GT. 0) THEN
         CALL ADFERR(IERR,MESS)
         PRINT *,MESS
         CALL ABORT('ADF ERROR')
      ENDIF
      RETURN
      END

      SUBROUTINE PRTCLD(PID)
C
C *** PRINT TABLE OF CHILDREN GIVEN A PARENT NODE-ID
C
      PARAMETER (MAXCLD=10)
      PARAMETER (MAXCHR=32)
      REAL*8 PID
      CHARACTER*(MAXCHR) NODNAM,NDNMS(MAXCLD)
      CALL ADFGNAM(PID,NODNAM,IERR)
      CALL ERRCHK(IERR)
      CALL ADFNCLD(PID,NUMC,IERR)
      CALL ERRCHK(IERR)
      WRITE(*,120)NODNAM,NUMC
  120 FORMAT(/,' PARENT NODE NAME = ',A,/,
     X       '     NUMBER OF CHILDREN = ',I2,/,
     X       '     CHILDREN NAMES:')
      NLEFT = NUMC
      ISTART = 1
C     --- TOP OF DO-WHILE LOOP
  130 CONTINUE
         CALL ADFCNAM(PID,ISTART,MAXCLD,LEN(NDNMS),
     X                NUMRET,NDNMS,IERR)
         CALL ERRCHK(IERR)
         WRITE(*,140)(NDNMS(K),K=1,NUMRET)
  140    FORMAT(2(8X,A))
         NLEFT = NLEFT - MAXCLD
         ISTART = ISTART + MAXCLD
      IF (NLEFT .GT. 0) GO TO 130
      RETURN
      END
```

The resulting output is:

```
*** ORIGINAL DATABASE SETUP ***

PARENT NODE NAME = PARENT.1
    NUMBER OF CHILDREN =  1
    CHILDREN NAMES:
       CHILD.1.1

PARENT NODE NAME = PARENT.2
    NUMBER OF CHILDREN =  4
    CHILDREN NAMES:
       CHILD.2.1                       CHILD.2.2
       CHILD.2.3                       CHILD.2.4

PARENT NODE NAME = PARENT.3
    NUMBER OF CHILDREN =  9
    CHILDREN NAMES:
       CHILD.3.1                       CHILD.3.2
       CHILD.3.3                       CHILD.3.4
       CHILD.3.5                       CHILD.3.6
       CHILD.3.7                       CHILD.3.8
       CHILD.3.9

*** PARENT.1 AND PARENT.3 AFTER MOVE ***

PARENT NODE NAME = PARENT.1
    NUMBER OF CHILDREN =  2
    CHILDREN NAMES:
       CHILD.1.1                       CHILD.3.4

PARENT NODE NAME = PARENT.3
    NUMBER OF CHILDREN =  8
    CHILDREN NAMES:
       CHILD.3.1                       CHILD.3.2
       CHILD.3.3                       CHILD.3.5
       CHILD.3.6                       CHILD.3.7
       CHILD.3.8                       CHILD.3.9
```

ADF_Link — *Create a Link to a Node*

ADF_Link (PID,name,file,name_in_file,ID,error_return)		
Language	**C**	**Fortran**
Routine Name	ADF_Link	ADFLINK
Input	const double PID const char *name const char *file const char *name_in_file	real*8 PID character*(*) name character*(*) file character*(*) name_in_file
Output	double ID int *error_return	real*8 ID integer error_return

PID	The ID of the node's parent.
name	The name of the link node.
file	The file name to use for the link directly usable by a C `open()` routine. If blank (null), the link is assumed to be within the same file as the parent (*PID*).
name_in_file	The name of the node that the link will point to. This can be a simple or a compound node.
ID	The ID of the created node.
error_return	Error return code. (See Appendix G.)

This routine, `ADF_Link`, will create a link (reference) to a node somewhere within the same ADF database file or another ADF database file. The node that the newly created link node refers to (points to) does not have to exist when the link is created. The ADF library does not check to make sure that the referenced node actually exists at the time `ADF_Link` is called. However, when information from the referenced node is requested by routines such as `ADF_Get_Label` or `ADF_Read_Data`, the referenced node is then accessed. If the referenced node is not in existence at that time, an error will occur. `ADF_Link` behaves like a "soft link" in the UNIX operating system.

Example 1

This example creates a link to another node that exists in the same physical file. Note that the length of *name_in_file* is not limited to 32 characters but can be any length required to fully specify the desired node.

```
      PROGRAM TEST
C
      PARAMETER (MAXCHR=32)
C
      CHARACTER*(MAXCHR) NODNAM,NODLBL,TSTLBL
      CHARACTER*(72) FN,PATH
C
C *** NODE IDS
```

```
C
      REAL*8 RID,PID,CID,PID1,PID3
      INTEGER I,J,IERR
C
C *** OPEN DATABASE
C
      CALL ADFDOPN('db.adf','NEW',' ',RID,IERR)
      CALL ERRCHK(IERR)
C
C *** CREATE NODES AT FIRST LEVEL
C
      WRITE(*,100)
  100 FORMAT(/,' *** ORIGINAL DATABASE SETUP ***')
      DO 150 I = 1,3
         WRITE(NODNAM,'(A7,I1)')'PARENT.',I
         CALL ADFCRE(RID,NODNAM,PID,IERR)
         CALL ERRCHK(IERR)
C
C ****** CREATE NODES AT SECOND LEVEL
C
         NUMCLD = I*I
         DO 110 J = 1,NUMCLD
            WRITE(NODNAM,'(A6,I1,A1,I1)')'CHILD.',I,'.',J
            CALL ADFCRE(PID,NODNAM,CID,IERR)
            CALL ERRCHK(IERR)
            WRITE(NODLBL,105)I,J
  105       FORMAT('LABEL STRING IN CHILD.',I1,'.',I1)
            CALL ADFSLB(CID,NODLBL,IERR)
            CALL ERRCHK(IERR)
  110    CONTINUE
C
C ****** GET NUMBER AND NAMES OF CHILDREN JUST CREATED
C        AND PRINT THEM OUT
C
         CALL PRTCLD(PID)
  150 CONTINUE
C
C *** LINK NODE /PARENT.3/CHILD.3.4 TO /PARENT.1
C
      CALL ADFGNID(RID,'PARENT.1',PID1,IERR)
      CALL ERRCHK(IERR)
C
      CALL ADFLINK(PID1,'LINKED_NODE',' ',
     X             '/PARENT.3/CHILD.3.4',CID,IERR)
      CALL ERRCHK(IERR)
C
C *** CHECK TO MAKE SURE THE NODE WAS ACTUALLY LINKED
C
      WRITE(*,160)
  160 FORMAT(/,'*** PARENT.1 AFTER LINK ***')
      CALL PRTCLD(PID1)
```

```
C
C *** FOR FINAL CONFIRMATION, GET ORIGINAL LABEL
C     GOING THROUGH NEW LINK
C
      CALL ADFGLB(CID,TSTLBL,IERR)
      CALL ERRCHK(IERR)
      WRITE(*,170)TSTLBL
  170 FORMAT(/,'LINKED_NODE LABEL = ',A)
C
      STOP
      END
C
C ************* SUBROUTINES ****************
C
      SUBROUTINE ERRCHK(IERR)
C
C *** CHECK ERROR CONDITION
C
      CHARACTER*80 MESS
      IF (IERR .GT. 0) THEN
         CALL ADFERR(IERR,MESS)
         PRINT *,MESS
         CALL ABORT('ADF ERROR')
      ENDIF
      RETURN
      END

      SUBROUTINE PRTCLD(PID)
C
C *** PRINT TABLE OF CHILDREN GIVEN A PARENT NODE-ID
C
      PARAMETER (MAXCLD=10)
      PARAMETER (MAXCHR=32)
      REAL*8 PID
      CHARACTER*(MAXCHR) NODNAM,NDNMS(MAXCLD)
      CALL ADFGNAM(PID,NODNAM,IERR)
      CALL ERRCHK(IERR)
      CALL ADFNCLD(PID,NUMC,IERR)
      CALL ERRCHK(IERR)
      WRITE(*,120)NODNAM,NUMC
  120 FORMAT(/,' PARENT NODE NAME = ',A,/,
     X       '     NUMBER OF CHILDREN = ',I2,/,
     X       '     CHILDREN NAMES:')
      NLEFT = NUMC
      ISTART = 1
C     --- TOP OF DO-WHILE LOOP
  130 CONTINUE
         CALL ADFCNAM(PID,ISTART,MAXCLD,LEN(NDNMS),
     X                NUMRET,NDNMS,IERR)
         CALL ERRCHK(IERR)
         WRITE(*,140)(NDNMS(K),K=1,NUMRET)
  140    FORMAT(2(8X,A))
         NLEFT = NLEFT - MAXCLD
         ISTART = ISTART + MAXCLD
      IF (NLEFT .GT. 0) GO TO 130
      RETURN
      END
```

The resulting output is:

```
*** ORIGINAL DATABASE SETUP ***

PARENT NODE NAME = PARENT.1
    NUMBER OF CHILDREN =  1
    CHILDREN NAMES:
       CHILD.1.1

PARENT NODE NAME = PARENT.2
    NUMBER OF CHILDREN =  4
    CHILDREN NAMES:
       CHILD.2.1                       CHILD.2.2
       CHILD.2.3                       CHILD.2.4

PARENT NODE NAME = PARENT.3
    NUMBER OF CHILDREN =  9
    CHILDREN NAMES:
       CHILD.3.1                       CHILD.3.2
       CHILD.3.3                       CHILD.3.4
       CHILD.3.5                       CHILD.3.6
       CHILD.3.7                       CHILD.3.8
       CHILD.3.9

*** PARENT.1 AFTER LINK ***

PARENT NODE NAME = PARENT.1
    NUMBER OF CHILDREN =  2
    CHILDREN NAMES:
       CHILD.1.1                       LINKED_NODE

LINKED_NODE LABEL = LABEL STRING IN CHILD.3.4
```

Example 2

This example illustrates the linking of nodes across files.

```
PROGRAM TEST
C
      PARAMETER (MAXCHR=32)
C
      CHARACTER*(MAXCHR) TSTLBL
C
```

```
C *** NODE IDS
C
      REAL*8 RID,PID,CID
      INTEGER IERR
C
C *** 1.) OPEN 1ST DATABASE
C     2.) CREATE TWO NODES
C     3.) PUT LABEL ON 2ND NODE
C     4.) CLOSE DATABASE
C
      CALL ADFDOPN('db1.adf','NEW',' ',RID,IERR)
      CALL ERRCHK(IERR)
      CALL ADFCRE(RID,'DB1_NODE1',PID,IERR)
      CALL ERRCHK(IERR)
      CALL ADFCRE(PID,'DB1_NODE2',CID,IERR)
      CALL ERRCHK(IERR)
      CALL ADFSLB(CID,'LABEL IN FILE.1: NODE2',IERR)
      CALL ERRCHK(IERR)
      CALL ADFDCLO(RID,IERR)
      CALL ERRCHK(IERR)
C
C *** 1.) OPEN 2ND DATABASE
C
      CALL ADFDOPN('db2.adf','NEW',' ',RID,IERR)
      CALL ERRCHK(IERR)
      CALL ADFCRE(RID,'DB2_NODE1',PID,IERR)
      CALL ERRCHK(IERR)
      CALL ADFCRE(PID,'DB2_NODE2',CID,IERR)
      CALL ERRCHK(IERR)
C
C *** LINK NODE /DB1_NODE1/DB1_NODE2 TO /DB2_NODE1
C
      CALL ADFLINK(PID,'LINKED_NODE','db1.adf',
     X             '/DB1_NODE1/DB1_NODE2',CID,IERR)
      CALL ERRCHK(IERR)
C
C *** CHECK TO MAKE SURE THE NODE WAS ACTUALLY LINKED
C
      WRITE(*,160)
  160 FORMAT(/,'*** PARENT AFTER LINK ***')
      CALL PRTCLD(PID)
C
C *** FOR FINAL CONFIRMATION, GET ORIGINAL LABEL
C     GOING THROUGH NEW LINK
C
      CALL ADFGLB(CID,TSTLBL,IERR)
      CALL ERRCHK(IERR)
      WRITE(*,170)TSTLBL
  170 FORMAT(/,'LINKED_NODE LABEL = ',A)
C
      STOP
      END
C
C ************* SUBROUTINES ****************
C
      SUBROUTINE ERRCHK(IERR)
C
C *** CHECK ERROR CONDITION
C
      CHARACTER*80 MESS
      IF (IERR .GT. 0) THEN
         CALL ADFERR(IERR,MESS)
         PRINT *,MESS
         CALL ABORT('ADF ERROR')
      ENDIF
      RETURN
      END

      SUBROUTINE PRTCLD(PID)
C
C *** PRINT TABLE OF CHILDREN GIVEN A PARENT NODE-ID
C
      PARAMETER (MAXCLD=10)
      PARAMETER (MAXCHR=32)
      REAL*8 PID
      CHARACTER*(MAXCHR) NODNAM,NDNMS(MAXCLD)
      CALL ADFGNAM(PID,NODNAM,IERR)
      CALL ERRCHK(IERR)
      CALL ADFNCLD(PID,NUMC,IERR)
      CALL ERRCHK(IERR)
      WRITE(*,120)NODNAM,NUMC
  120 FORMAT(/,' PARENT NODE NAME = ',A,/,
     X       '     NUMBER OF CHILDREN = ',I2,/,
     X       '     CHILDREN NAMES:')
      NLEFT = NUMC
      ISTART = 1
C     --- TOP OF DO-WHILE LOOP
  130 CONTINUE
         CALL ADFCNAM(PID,ISTART,MAXCLD,LEN(NDNMS),
     X                NUMRET,NDNMS,IERR)
         CALL ERRCHK(IERR)
         WRITE(*,140)(NDNMS(K),K=1,NUMRET)
  140    FORMAT(2(8X,A))
         NLEFT = NLEFT - MAXCLD
         ISTART = ISTART + MAXCLD
      IF (NLEFT .GT. 0) GO TO 130
      RETURN
      END
```

The resulting output is:

```
*** PARENT AFTER LINK ***
```

```
 PARENT NODE NAME = DB2_NODE1
     NUMBER OF CHILDREN =  2
     CHILDREN NAMES:
        DB2_NODE2                       LINKED_NODE

LINKED_NODE LABEL = LABEL IN FILE.1: NODE2
```

`ADF_Is_Link` — *See If the Node Is a Link*

`ADF_Is_Link (ID,link_path_length,error_return)`		
Language	**C**	**Fortran**
Routine Name	`ADF_Is_Link`	`ADFISLK`
Input	`const double ID`	`real*8 ID`

Output	`int *link_path_length`
	`int *error_return`
		`integer link_path_length`
		`integer error_return`

ID	The ID of the node to use.
link_path_length	This returned value is zero if the node is not a link. If the node referenced by *ID* is a link within the same file, the number of characters in the path is returned. If the node referenced by *ID* is a link in another file, the sum of the number of characters in the referenced file name and the number of characters in the path +1 is returned.
error_return	Error return code. (See Appendix G.)

This routine, `ADF_Is_Link`, tests to see if the node is a link. If the actual data type of the node is `LK` (created with `ADF_Link`), the routine returns the link path length; otherwise it returns 0.

Example

This example creates a link into a second file, and then calls `ADF_Is_Link` to determine whether the requested node is indeed a link or a normal node.

```
PROGRAM TEST
C
      PARAMETER (MAXCHR=32)
C
      CHARACTER*(MAXCHR) TSTLBL
C
C *** NODE IDS
C
      REAL*8 RID,PID,CID
      INTEGER IERR
C
C *** 1.) OPEN 1ST DATABASE
C     2.) CREATE TWO NODES
C     3.) PUT LABEL ON 2ND NODE
C     4.) CLOSE DATABASE
C
      CALL ADFDOPN('db1.adf','NEW',' ',RID,IERR)
      CALL ERRCHK(IERR)
      CALL ADFCRE(RID,'DB1_NODE1',PID,IERR)
      CALL ERRCHK(IERR)
      CALL ADFCRE(PID,'DB1_NODE2',CID,IERR)
      CALL ERRCHK(IERR)
      CALL ADFDCLO(RID,IERR)
      CALL ERRCHK(IERR)
C
C *** 1.) OPEN 2ND DATABASE
C
      CALL ADFDOPN('db2.adf','NEW',' ',RID,IERR)
      CALL ERRCHK(IERR)
      CALL ADFCRE(RID,'DB2_NODE1',PID,IERR)
      CALL ERRCHK(IERR)
      CALL ADFCRE(PID,'DB2_NODE2',CID,IERR)
      CALL ERRCHK(IERR)
C
C *** LINK NODE FILE 1:/DB1_NODE1/DB1_NODE2 TO /DB2_NODE1
C
      CALL ADFLINK(PID,'LINKED_NODE','db1.adf',
     X             '/DB1_NODE1/DB1_NODE2',CID,IERR)
      CALL ERRCHK(IERR)
C
C *** CHECK TO MAKE SURE THE NODE WAS ACTUALLY LINKED
C
      CALL ADFISLK(CID,LEN,IERR)
      CALL ERRCHK(IERR)
      PRINT *,' PATH LENGTH FROM LINK IS: ',LEN
C
      STOP
      END
C
C ************* SUBROUTINES ****************
C
      SUBROUTINE ERRCHK(IERR)
C
C *** CHECK ERROR CONDITION
C
      CHARACTER*80 MESS
      IF (IERR .GT. 0) THEN
         CALL ADFERR(IERR,MESS)
         PRINT *,MESS
         CALL ABORT('ADF ERROR')
      ENDIF
      RETURN
      END
```

The resulting output is:

```
PATH LENGTH FROM LINK IS:          28
```

`ADF_Get_Link_Path` — *Get the Path Information From a Link*

`ADF_Get_Link_Path (ID,file,name_in_file,error_return)`

Language	**C**	**Fortran**
Routine Name	`ADF_Get_Link_Path`	`ADFGLKP`
Input	`const double ID`	`real*8 ID`

Output	`char *file`
	`char *name_in_file`
	`int *error_return`
		`character*(*) file`
		`character*(*) name_in_file`
		`integer error_return`

ID	The ID of the node to use.
file	The file name to use for the link. It is directly usable by the C `open()` routine. If blank (null), the link is within the same file.
name_in_file	The name of the node that the link points to.
error_return	Error return code. (See Appendix G.)

This routine, `ADF_Get_Link_Path`, gets the path information from a link. If the node is a link node, the routine returns the path information; otherwise it returns an error.

Example

This example opens two ADF files. A link is created in the second file (*db2.adf*) that references a node in the first file (*db1.adf*). `ADFGLKP` is used to extract the file name and path that the link actually points to. It is not anticipated that this information will be useful to the normal user. For most applications, it is best to allow ADF to resolve the link internally, transparent to the user.

```
PROGRAM TEST
C
      PARAMETER (MAXCHR=32)
C
      CHARACTER*(MAXCHR) TSTLBL
      CHARACTER*(40) FILENM,PATH
C
C *** NODE IDS
C
      REAL*8 RID,PID,CID
      INTEGER IERR
C
C *** 1.) OPEN 1ST DATABASE
C     2.) CREATE TWO NODES
C     3.) PUT LABEL ON 2ND NODE
C     4.) CLOSE DATABASE
C
      CALL ADFDOPN('db1.adf','NEW',' ',RID,IERR)
      CALL ADFCRE(RID,'DB1_NODE1',PID,IERR)
      CALL ADFCRE(PID,'DB1_NODE2',CID,IERR)
      CALL ADFDCLO(RID,IERR)
C
C *** 1.) OPEN 2ND DATABASE
C
      CALL ADFDOPN('db2.adf','NEW',' ',RID,IERR)
      CALL ADFCRE(RID,'DB2_NODE1',PID,IERR)
      CALL ADFCRE(PID,'DB2_NODE2',CID,IERR)
C
C *** LINK NODE FILE 1:/DB1_NODE1/DB1_NODE2 TO /DB2_NODE1
C
      CALL ADFLINK(PID,'LINKED_NODE','db1.adf',
     X             '/DB1_NODE1/DB1_NODE2',CID,IERR)
C
C *** CHECK TO MAKE SURE THE NODE WAS ACTUALLY LINKED
C
      CALL ADFGLKP(CID,FILENM,PATH,IERR)
      PRINT *,' INFORMATION FROM LINK:'
      PRINT *,' FILE: ',FILENM
      PRINT *,' PATH: ',PATH
C
      STOP
      END
C
C ************* SUBROUTINES ****************
C
      SUBROUTINE ERRCHK(IERR)
C
C *** CHECK ERROR CONDITION
C
      CHARACTER*80 MESS
      IF (IERR .GT. 0) THEN
         CALL ADFERR(IERR,MESS)
         PRINT *,MESS
         CALL ABORT('ADF ERROR')
      ENDIF
      RETURN
      END

      SUBROUTINE PRTCLD(PID)
C
C *** PRINT TABLE OF CHILDREN GIVEN A PARENT NODE-ID
C
      PARAMETER (MAXCLD=10)
      PARAMETER (MAXCHR=32)
      REAL*8 PID
      CHARACTER*(MAXCHR) NODNAM,NDNMS(MAXCLD)
      CALL ADFGNAM(PID,NODNAM,IERR)
      CALL ERRCHK(IERR)
      CALL ADFNCLD(PID,NUMC,IERR)
      CALL ERRCHK(IERR)
      WRITE(*,120)NODNAM,NUMC
  120 FORMAT(/,' PARENT NODE NAME = ',A,/,
     X       '     NUMBER OF CHILDREN = ',I2,/,
     X       '     CHILDREN NAMES:')
      NLEFT = NUMC
      ISTART = 1
C     --- TOP OF DO-WHILE LOOP
  130 CONTINUE
         CALL ADFCNAM(PID,ISTART,MAXCLD,LEN(NDNMS),
     X                NUMRET,NDNMS,IERR)
         CALL ERRCHK(IERR)
         WRITE(*,140)(NDNMS(K),K=1,NUMRET)
  140    FORMAT(2(8X,A))
         NLEFT = NLEFT - MAXCLD
         ISTART = ISTART + MAXCLD
      IF (NLEFT .GT. 0) GO TO 130
      RETURN
      END
```

The resulting output is:

```
INFORMATION FROM LINK:
FILE: db1.adf
PATH: /DB1_NODE1/DB1_NODE2
```

ADF_Get_Root_ID — *Get the Root ID for the ADF System*

ADF_Get_Root_ID (ID,root_ID,error_return)		
Language	**C**	**Fortran**
Routine Name	ADF_Get_Root_ID	ADFGRID
Input	const double ID	real*8 ID
Output	double root_ID int *error_return	real*8 root_ID integer error_return

ID	Any valid node ID for the given ADF database.
root_ID	The ID of the root node
error_return	Error return code. (See Appendix G.)

This routine, `ADF_Get_Root_ID`, returns the root ID for an ADF file when given any valid node ID for that file.

Example

This example illustrates that the root node ID can be obtained at any time by using a currently valid node ID for that file.

```
PROGRAM TEST
C
      PARAMETER (MAXCHR=32)
C
      CHARACTER*(MAXCHR) NODNAM,ROOTNM
C
C *** NODE IDS
C
      REAL*8 RID,PID,CID,TESTID
      INTEGER I,J,IERR,NUMCLD
C
C *** OPEN DATABASE
C
      CALL ADFDOPN('db.adf','NEW',' ',RID,IERR)
      CALL ADFGNAM(RID,ROOTNM,IERR)
      PRINT *,' AFTER OPENING FILE, ROOT NAME = ',ROOTNM
C
C *** CREATE NODES AT FIRST LEVEL
C
      DO 150 I = 1,2
         WRITE(NODNAM,'(A7,I1)')'PARENT.',I
         CALL ADFCRE(RID,NODNAM,PID,IERR)
         TESTID = 0.0
         ROOTNM =''
         CALL ADFGRID(PID,TESTID,IERR)
         CALL ADFGNAM(TESTID,ROOTNM,IERR)
         WRITE(*,100)NODNAM,ROOTNM
  100    FORMAT('USING NODE ID FROM: ',A,
     X          ' ROOT NAME = ',A)
C
C ****** CREATE NODES AT SECOND LEVEL
C
         NUMCLD = 2*I
         DO 110 J = 1,NUMCLD
            WRITE(NODNAM,'(A6,I1,A1,I1)')'CHILD.',I,'.',J
            CALL ADFCRE(PID,NODNAM,CID,IERR)
            TESTID = 0.0
            ROOTNM =''
            CALL ADFGRID(PID,TESTID,IERR)
            CALL ADFGNAM(TESTID,ROOTNM,IERR)
            WRITE(*,100)NODNAM,ROOTNM
  110    CONTINUE
  150 CONTINUE
C
      STOP
      END
```

The resulting output is:

```
 AFTER OPENING FILE, ROOT NAME = ADF MotherNode
USING NODE ID FROM: PARENT.1         ROOT NAME = ADF MotherNode
USING NODE ID FROM: CHILD.1.1        ROOT NAME = ADF MotherNode
USING NODE ID FROM: CHILD.1.2        ROOT NAME = ADF MotherNode
USING NODE ID FROM: PARENT.2         ROOT NAME = ADF MotherNode
USING NODE ID FROM: CHILD.2.1        ROOT NAME = ADF MotherNode
USING NODE ID FROM: CHILD.2.2        ROOT NAME = ADF MotherNode
USING NODE ID FROM: CHILD.2.3        ROOT NAME = ADF MotherNode
USING NODE ID FROM: CHILD.2.4        ROOT NAME = ADF MotherNode
```

I.5 Data Query Routines

ADF_Get_Label — *Get the String in a Node's Label Field*

```
ADF_Get_Label (ID,label,error_return)
```

Language	C	Fortran
Routine Name	`ADF_Get_Label`	`ADFGLB`
Input	`const double ID`	`real*8 ID`

Output	`char *label`
	`int *error_return`
		`character*(*) label`
		`integer error_return`

ID	The ID of the node to use.
label	The 32-character label of the node.
error_return	Error return code. (See Appendix G.)

This routine returns the 32-character string stored in the node's label field. In C, the label will be null terminated after the last nonblank character. Therefore, in general, 33 characters should be allocated (32 for the label, plus 1 for the null). In Fortran, the label is left justified and blank filled to the right. The null character is not returned in Fortran, therefore, the variable declaration can be for 32 characters (e.g., `CHARACTER*(32)`).

Example

```
PROGRAM TEST
C
      PARAMETER (MAXCHR=32)
C
      CHARACTER*(MAXCHR) NODNAM,LABL
C
C *** NODE IDS
C
      REAL*8 RID,CID
C
C *** OPEN DATABASE
C
      CALL ADFDOPN('db.adf','NEW',' ',RID,IERR)
      CALL ADFCRE(RID,'NODE 1',CID,IERR)
      CALL ADFSLB(CID,'THIS IS A NODE LABEL',IERR)
C
      CALL ADFGNAM(CID,NODNAM,IERR)
      CALL ADFGLB(CID,LABL,IERR)
C
      PRINT *,' NODE NAME = ',NODNAM
      PRINT *,' LABEL     = ',LABL
C
```

```
         STOP
         END
```

The resulting output is:

```
   NODE NAME = NODE 1
   LABEL     = THIS IS A NODE LABEL
```

`ADF_Set_Label` — *Set the String in a Node's Label Field*

`ADF_Set_Label (ID,label,error_return)`		
Language	**C**	**Fortran**
Routine Name	`ADF_Set_Label`	`ADFSLB`

Input	`const double ID`
	`const char *label`
		`real*8 ID`
		`character*(*) label`

Output	`int *error_return`	`integer error_return`

ID	The ID of the node to use.
label	The 32-character label for the node.
error_return	Error return code. (See Appendix G.)

This routine, `ADF_Set_Label`, sets the 32-character string in the node's label field.

Example

See the example for `ADF_Get_Label`.

`ADF_Get_Data_Type` — *Get the String in a Node's Data-Type Field*

`ADF_Get_Data_Type (ID,data_type,error_return)`

Language	**C**	**Fortran**
Routine Name	`ADF_Get_Data_Type`	`ADFGDT`
Input	`const double ID`	`real*8 ID`

Output	`char *data_type`
	`int *error_return`
		`character*(*) data_type`
		`integer error_return`

ID	The ID of the node to use.
data_type	The 32-character data type field stored in the node information header.
error_return	Error return code. (See Appendix G.)

This routine, `ADF_Get_Data_Type`, returns the 32-character string in the node's data-type field. In C, the label will be null terminated after the last nonblank; therefore, at least 33 characters (32 for the label, plus 1 for the null) should be allocated for the *data_type* string. In Fortran, the null character is not returned; therefore, the declaration for the string *data_type* can be for 32 characters.

Example

This example illustrates the process of creating a node, storing data in it, querying the node for the information in the header, and reading the data back out.

```
PROGRAM TEST
C
      PARAMETER (MAXCHR=32)
      PARAMETER (MAXROW=2)
      PARAMETER (MAXCOL=10)
C
      CHARACTER*(MAXCHR) NODNAM,LABL
      CHARACTER*(MAXCHR) DTYPE
      REAL R4DATI(MAXROW,MAXCOL),R4DATO(MAXROW,MAXCOL)
      INTEGER IDIMI(2),IDIMO(2)
C
C *** NODE IDS
C
      REAL*8 RID,CID
C
C *** OPEN DATABASE
C
      CALL ADFDOPN('db.adf','NEW',' ',RID,IERR)
C
C *** GENERATE SOME DATA
C
      IDIMI(1) = MAXROW
      IDIMI(2) = MAXCOL
      DO 200 ICOL = 1,MAXCOL
         DO 100 IROW = 1,MAXROW
            R4DATI(IROW,ICOL) = 2.0*ICOL*IROW
  100    CONTINUE
  200 CONTINUE
C
C *** GENERATE A NODE AND PUT DATA IN IT
C
      CALL ADFCRE(RID,'NODE 1',CID,IERR)
      CALL ADFSLB(CID,'LABEL FOR NODE 1',IERR)
      CALL ADFPDIM(CID,'R4',2,IDIMI,IERR)
      CALL ADFWALL(CID,R4DATI,IERR)
C
C *** GET INFORMATION FROM NODE
C
      CALL ADFGNAM(CID,NODNAM,IERR)
      CALL ADFGLB(CID,LABL,IERR)
      CALL ADFGDT(CID,DTYPE,IERR)
      CALL ADFGND(CID,NDIM,IERR)
      CALL ADFGDV(CID,IDIMO,IERR)
      CALL ADFRALL(CID,R4DATO,IERR)
C
      PRINT *,' NODE NAME            = ',NODNAM
      PRINT *,' LABEL                = ',LABL
      PRINT *,' DATA TYPE            = ',DTYPE
      PRINT *,' NUMBER OF DIMENSIONS = ',NDIM
      PRINT *,' DIMENSIONS           = ',IDIMO
      PRINT *,' DATA:'
      WRITE(*,300)((R4DATO(I,J),I=1,MAXROW),J=1,MAXCOL)
  300 FORMAT(2(5X,F10.2))
C
      STOP
      END
```

The resulting output is:

```
NODE NAME            = NODE 1
LABEL                = LABEL FOR NODE 1
DATA TYPE            = R4
NUMBER OF DIMENSIONS =           2
DIMENSIONS           =           2           10
DATA:
         2.00           4.00
         4.00           8.00
         6.00          12.00
         8.00          16.00
        10.00          20.00
        12.00          24.00
        14.00          28.00
        16.00          32.00
        18.00          36.00
```

```
         20.00          40.00
```

`ADF_Get_Number_of_Dimensions` — *Get the Number of Node Dimensions*

`ADF_Get_Number_of_Dimensions (ID,num_dims,error_return)`		
Language	**C**	**Fortran**
Routine Name	`ADF_Get_Number_of_Dimensions`	`ADFGND`
Input	`const double ID`	`real*8 ID`

Output	`int *num_dims`
	`int *error_return`
		`integer num_dims`
		`integer error_return`

ID	The ID of the node to use.
num_dims	The integer dimension value.
error_return	Error return code. (See Appendix G.)

This routine, `ADF_Get_Number_of_Dimensions`, returns the number of data dimensions used in the node. Values will be returned only for the number of dimensions defined in the node. If the number of dimensions for the node is zero, an error is returned.

Example

See the example for `ADF_Get_Data_Type`.

`ADF_Get_Dimension_Values` — *Get the Values of the Node Dimensions*

`ADF_Get_Dimension_Values (ID,dim_vals[],error_return)`

Language	**C**	**Fortran**
Routine Name	`ADF_Get_Dimension_Values`	`ADFGDV`
Input	`const double ID`	`real*8 ID`

Output	`int *dim_vals[]`
	`int *error_return`
		`integer dim_vals()`
		`integer error_return`

ID	The ID of the node to use.
dim_vals	The array (list) of dimension values.
error_return	Error return code. (See Appendix G.)

This routine, `ADF_Get_Dimension_Values`, returns the array (list) of dimension values for a node. Values will be returned only for the number of dimensions defined in the node. If the number of dimensions for the node is zero, an error is returned.

Example

See the example for `ADF_Get_Data_Type`.

`ADF_Put_Dimension_Information` — *Set or Change the Data Type and Dimensions of a Node*

`ADF_Put_Dimension_Information (ID,data_type,dims,dim_vals[],error_return)`		
Language	**C**	**Fortran**
Routine Name	`ADF_Put_Dimension_Information`	`ADFPDIM`

Input	`const double ID`
	`const char *data_type`
	`const int dims`
	`int dim_vals[]`
		`real*8 ID`
		`character*(*) data_type`
		`integer dims`
		`integer dim_vals()`

Output	`int *error_return`	`integer error_return`

ID The ID of the node to use.

data_type The 32-character data type of the node. The valid user-definable data types are:

Data Type	Notation
No Data	`MT`
Integer 32	`I4`
Integer 64	`I8`
Unsigned Integer 32	`U4`
Unsigned Integer 64	`U8`
Real 32	`R4`
Real 64	`R8`
Complex 64	`X4`
Complex 128	`X8`
Character (unsigned byte)	`C1`
Byte (unsigned byte)	`B1`

Compound data types can be defined as a combination of types (“`I4,I4,R8`”), an array (“`I4[25]`”), or a combination of types and arrays (“`I4,C1[20],R8[3]`”). They can contain up to 32 characters. This style of data type definition is not very portable across platforms; therefore, it is not recommended.

dims The number of dimensions of this node. *dims* can be a number from 0 to 12. “0” means no data. The dimension of an array can range from 1 (vector) to 12.

dim_vals The array (list) of dimension values for this node. *dim_vals* is a vector of integers that define the size of the array in each dimension as defined by *dims*. If the dims is zero, the *dims_vals* are not used. The valid range of *dim_vals* is from 1 to 2,147,483,648. The total data size in bytes, calculated by the *data_type* size times the dimension values, cannot exceed 2,147,483,648 for any one node.

error_return Error return code. (See Appendix G.)

This routine, `ADF_Put_Dimension_Information`, sets or changes the data type and dimension information for a node.

Note: When this routine is used to change the data type or number of dimensions of an existing node, any data currently associated with the node are lost. The dimension values can be changed and the data space will be extended as needed. Be very careful changing the dimension values. The layout of the data is assumed to be Fortran-like. If the left-most dimension values are changed, the data are not shifted around on disk to account for this; only the amount of data is changed. Therefore, the indexing into the data will be changed. In general, it is safe to change the right-most dimension value. For example, if the array of dimension values was (10,11,20,50), then changing the 10, 11, or 20 is very risky, whereas changing the 50 should be safe.

Note: See the discussion of Fortran character array portability in Appendix E.7.

Example

See the example for `ADF_Get_Data_Type`.

I.6 Data I/O Routines

Note: For all data I/O routines, the system is based on indexing starting from 1 and not 0. (That is, the first element in an array is indexed as 1 and not zero.)

`ADF_Read_Data` — *Read the Data From a Node Having Stride Capabilities*

```
ADF_Read_Data (ID,s_start[],s_end[],s_stride[],m_num_dims,m_dims[],m_start[],
              m_end[],m_stride[],data,error_return)
```

Language	**C**	**Fortran**
Routine Name	`ADF_Read_Data`	`ADFREAD`
Input	`const double ID`	`real*8 ID`
	`const int s_start[]`	`integer s_start()`
	`const int s_end[]`	`integer s_end()`
	`const int s_stride[]`	`integer s_stride()`
	`const int m_num_dims`	`integer m_num_dims`
	`const int m_dims[]`	`integer m_dims()`
	`const int m_start[]`	`integer m_start()`
	`const int m_end[]`	`integer m_end()`
	`const int m_stride[]`	`integer m_stride()`
Output	`char *data`	`character*(*) data`
	`int *error_return`	`integer error_return`

ID	The ID of the node to use.
s_start[]	The starting index to use for each dimension of the array within the database node (1D array; i.e., list of indices). The maximum number of dimensions an array is allowed in ADF is 12.
s_end[]	The ending index to use for each dimension of the array within the database node (1D array; i.e., list of indices).
s_stride[]	The stride value to use for each dimension of the array within the database node (1D array; i.e., list of indices).
m_num_dims	The number of dimensions to use in memory.
m_dims[]	The dimension values to use for the array in memory (1D array; i.e., list of indices).
m_start[]	The starting index to use for each dimension of the array in memory (1D array; i.e., list of indices).
m_end[]	The ending index to use for each dimension of the array in memory (1D array; i.e., list of indices).
m_stride[]	The stride value to use for each dimension of the array in memory (1D array; i.e., list of indices).

data	The starting address of the data in memory.
error_return	Error return code. (See Appendix G.)

This routine, `ADF_Read_Data`, provides general purpose read capabilities. It allows for a general specification of the starting location within the data well as fixed step lengths (strides) through the data from the initial position. This capability works for both the data on disk and the data being stored in memory. One set of integer vectors (*s_start*, etc.) is used to describe the mapping of the data within the node, and a second set of integer vectors (*m_start*) is used to describe the mapping of the desired data within memory.

There can be a significant performance penalty for using `ADF_Read_Data` when compared with `ADF_Read_All_Data`. If performance is a major consideration, it is best to organize data to take advantage of the speed of `ADF_Read_All_Data`.

The data are stored in both memory and on disk in "Fortran ordering." That is, the first index varies the fastest.

`ADF_Read_Data` will not accept "negative" indexing. That is, it is not possible to reverse the ordering of the data from the node into memory.

Be careful when writing data using `ADF_Write_All_Data` and then using `ADF_Read_Data` to randomly access the data. `ADF_Write_All_Data` takes a starting address in memory and writes *N* words to disk, making no assumption as to the order of the data. `ADF_Read_Data` assumes that the data have Fortran-like ordering to navigate through the data in memory and on disk. It assumes that the first dimension varies the fastest. It would be easy for a C program to use the default array ordering (last dimension varying fastest) and write the data out using `ADF_Write_All_Data`. Then another program might use `ADF_Read_Data` to access a subsection of the data, and the routine would not return what was expected.

Note: If all the data type of the node is a compound data type, such as ("`I4[3],R8`"), the partial capabilities will access one or more of these 20-byte data entities. You cannot access a subset of an occurrence of the data type.

Note: See the discussion of Fortran character array portability in Appendix E.7.

Example 1

This example shows `ADFREAD` being used to emulate the same capabilities as those in `ADFRALL`.

```
PROGRAM TEST
C
      PARAMETER (MAXROW=10)
      PARAMETER (MAXCOL=3)
C
      REAL R4ARRI(MAXROW,MAXCOL)
      REAL R4ARRO(MAXROW,MAXCOL)
      INTEGER IDIMI(2),IDIMO(2)
      INTEGER IDBEG(2),IDEND(2),IDINCR(2)
      INTEGER IMBEG(2),IMEND(2),IMINCR(2)
C
C *** NODE IDS
C
      REAL*8 RID,CID
C
```

```
C *** OPEN DATABASE
C
      CALL ADFDOPN('db.adf','NEW',' ',RID,IERR)
C
C *** GENERATE SOME DATA
C
      IDIMI(1) = MAXROW
      IDIMI(2) = MAXCOL
      DO 200 ICOL = 1,MAXCOL
         DO 100 IROW = 1,MAXROW
            R4ARRI(IROW,ICOL) = 2.0*ICOL*IROW
  100    CONTINUE
  200 CONTINUE
      PRINT *,' ORIGINAL ARRAY:'
      WRITE(*,300)((R4ARRI(I,J),J=1,MAXCOL),I=1,MAXROW)
  300 FORMAT(3(5X,F10.2))
C
C *** GENERATE A NODE AND PUT DATA IN IT
C
      CALL ADFCRE(RID,'NODE 1',CID,IERR)
      CALL ADFSLB(CID,'LABEL FOR NODE 1',IERR)
      CALL ADFPDIM(CID,'R4',2,IDIMI,IERR)
      CALL ADFWALL(CID,R4ARRI,IERR)
C
C *** GET INFORMATION FROM NODE
C
C *** GET DATA FROM NODE (EXACTLY EQUIVALENT TO ADFRALL)
C
      IDBEG(1) = 1
      IDEND(1) = MAXROW
      IDINCR(1) = 1
C
      IDBEG(2) = 1
      IDEND(2) = MAXCOL
      IDINCR(2) = 1
C
      IDIMO(1) = MAXROW
      IDIMO(2) = MAXCOL
C
      IMBEG(1) = 1
      IMEND(1) = MAXROW
      IMINCR(1) = 1
C
      IMBEG(2) = 1
      IMEND(2) = MAXCOL
      IMINCR(2) = 1
      CALL ADFREAD(CID,IDBEG,IDEND,IDINCR,
     X             2,IDIMO,IMBEG,IMEND,IMINCR,
     X             R4ARRO,IERR)
      CALL ERRCHK(IERR)
C
```

```
      PRINT *,' ARRAY PULLED FROM DISK USING ADFREAD:'
      WRITE(*,300)((R4ARRO(I,J),J=1,MAXCOL),I=1,MAXROW)
C
      STOP
      END

C
C ************* SUBROUTINES ****************
C
      SUBROUTINE ERRCHK(IERR)
C
C *** CHECK ERROR CONDITION
C
      CHARACTER*80 MESS
      IF (IERR .GT. 0) THEN
         CALL ADFERR(IERR,MESS)
         PRINT *,MESS
         CALL ABORT('ADF ERROR')
      ENDIF
      RETURN
      END
```

The resulting output is:

```
ORIGINAL ARRAY:
          2.00           4.00           6.00
          4.00           8.00          12.00
          6.00          12.00          18.00
          8.00          16.00          24.00
         10.00          20.00          30.00
         12.00          24.00          36.00
         14.00          28.00          42.00
         16.00          32.00          48.00
         18.00          36.00          54.00
         20.00          40.00          60.00

ARRAY PULLED FROM DISK USING ADFREAD:
          2.00           4.00           6.00
          4.00           8.00          12.00
          6.00          12.00          18.00
          8.00          16.00          24.00
         10.00          20.00          30.00
         12.00          24.00          36.00
         14.00          28.00          42.00
         16.00          32.00          48.00
         18.00          36.00          54.00
         20.00          40.00          60.00
```

Example 2

This example illustrates some of the flexibility available with `ADF_Read_Data`. An array is created and written to disk using ADFWALL. Then every other entry in the second column is read back into every other element of a vector.

```
PROGRAM TEST
C
      PARAMETER (MAXROW=10)
      PARAMETER (MAXCOL=3)
C
      REAL R4ARRI(MAXROW,MAXCOL),R4VECO(MAXROW)
      INTEGER IDIMD(2)
      INTEGER IDBEG(2),IDEND(2),IDINCR(2)
C
C *** NODE IDS
C
      REAL*8 RID,CID
C
C *** OPEN DATABASE
C
      CALL ADFDOPN('db.adf','NEW',' ',RID,IERR)
C
C *** GENERATE SOME DATA
C
      IDIMD(1) = MAXROW
      IDIMD(2) = MAXCOL
      DO 200 ICOL = 1,MAXCOL
         DO 100 IROW = 1,MAXROW
            R4ARRI(IROW,ICOL) = 2.0*ICOL*IROW
  100    CONTINUE
  200 CONTINUE
C
      DO 250 I = 1,MAXROW
         R4VECO(I) = 0.0
  250 CONTINUE
C
      PRINT *,' ORIGINAL ARRAY:'
      WRITE(*,300)((R4ARRI(I,J),J=1,MAXCOL),I=1,MAXROW)
  300 FORMAT(3(5X,F10.2))
C
C *** GENERATE A NODE AND PUT DATA IN IT
C
      CALL ADFCRE(RID,'NODE 1',CID,IERR)
      CALL ADFSLB(CID,'LABEL FOR NODE 1',IERR)
      CALL ADFPDIM(CID,'R4',2,IDIMD,IERR)
      CALL ADFWALL(CID,R4ARRI,IERR)
C
C *** GET DATA FROM NODE USING STRIDED READ
C
C ****** TAKE EVERY OTHER NUMBER FROM THE 2ND COLUMN OF THE ARRAY
C        AND PUT IT IN SEQUENTIALLY IN A VECTOR IN MEMORY
C
C *** DATABASE STRIDE INFORMATION
```

```
C
      IDBEG(1) = 1
      IDEND(1) = MAXROW
      IDINCR(1) = 2
C
      IDBEG(2) = 2
      IDEND(2) = 2
      IDINCR(2) = 1
C
C *** MEMORY STRIDE INFORMATION
C
      NDIMM = 1
      IDIMM = MAXROW
      IMBEG = 1
      IMEND = MAXROW
      IMINCR = 2
C
      CALL ADFREAD(CID,IDBEG,IDEND,IDINCR,
     X             NDIMM,IDIMM,IMBEG,IMEND,IMINCR,
     X             R4VECO,IERR)
      CALL ERRCHK(IERR)
C
      PRINT *,' VECTOR WITH DATA EXTRACTED FROM ARRAY'
      WRITE(*,400)(R4VECO(J),J=1,MAXROW)
  400 FORMAT(3(5X,F10.2))
C
      STOP
      END

C
C ************* SUBROUTINES ****************
C
      SUBROUTINE ERRCHK(IERR)
C
C *** CHECK ERROR CONDITION
C
      CHARACTER*80 MESS
      IF (IERR .GT. 0) THEN
         CALL ADFERR(IERR,MESS)
         PRINT *,MESS
         CALL ABORT('ADF ERROR')
      ENDIF
      RETURN
      END
```

The resulting output is:

```
ORIGINAL ARRAY:
          2.00           4.00           6.00
          4.00           8.00          12.00
          6.00          12.00          18.00
```

```
           8.00           16.00           24.00
          10.00           20.00           30.00
          12.00           24.00           36.00
          14.00           28.00           42.00
          16.00           32.00           48.00
          18.00           36.00           54.00
          20.00           40.00           60.00
VECTOR WITH DATA EXTRACTED FROM ARRAY
           4.00            0.00           12.00
           0.00           20.00            0.00
          28.00            0.00           36.00
           0.00
```

`ADF_Read_All_Data` — *Read All the Data From a Node*

`ADF_Read_All_Data (ID,data,error_return)`

Language	**C**	**Fortran**
Routine Name	`ADF_Read_All_Data`	`ADFRALL`
Input	`const double ID`	`real*8 ID`

Output	`char *data`
	`int *error_return`
		`character*(*) data`
		`integer error_return`

ID	The ID of the node to use.
data	The starting address of the data in memory.
error_return	Error return code. (See Appendix G.)

This routine, `ADF_Read_All_Data`, reads all data from a node. It reads all the node's data and returns them into a contiguous memory space.

The disk performance of `ADF_Read_All_Data` is very good. The routine issues a single read command to the system for the entire data set; therefore, it is as fast as the system can return the data.

Note: See the discussion of Fortran character array portability in Appendix E.7.

Example

See the example for `ADF_Get_Data_Type`.

`ADF_Read_Block_Data` — *Read a Contiguous Block of Data From a Node*

`ADF_Read_Block_Data (ID,b_start,b_end,data,error_return)`

Language	C	Fortran
Routine Name	`ADF_Read_Block_Data`	`ADFRBLK`

Input	`const double ID`
	`const long b_start`
	`const long b_end`
		`real*8 ID`
		`integer b_start`
		`integer b_end`

Output	`char *data`
	`int *error_return`
		`character*(*) data`
		`integer error_return`

ID	The ID of the node to use.
b_start	The starting point of the block in token space.
b_end	The ending point of the block in token space.
data	The starting address of the data in memory.
error_return	Error return code. (See Appendix G.)

This routine, `ADF_Read_Block_Data`, reads a block of data from a node and returns it into a contiguous memory space.

Note: See the discussion of Fortran character array portability in Appendix E.7.

`ADF_Write_Data` — *Write the Data to a Node Having Stride Capabilities*

```
ADF_Write_Data (ID,s_start[],s_end[],s_stride[],m_num_dims,m_dims[],m_start[],
                m_end[],m_stride[],data,error_return)
```

Language	**C**	**Fortran**
Routine Name	`ADF_Write_Data`	`ADFWRIT`
Input	`const double ID`	`real*8 ID`
	`const int s_start[]`	`integer s_start()`
	`const int s_end[]`	`integer s_end()`
	`const int s_stride[]`	`integer s_stride()`
	`const int m_num_dims`	`integer m_num_dims`
	`const int m_dims[]`	`integer m_dims()`
	`const int m_start[]`	`integer m_start()`
	`const int m_end[]`	`integer m_end()`
	`const int m_stride[]`	`integer m_stride()`
	`char *data`	`character*(*) data`
Output	`int *error_return`	`integer error_return`

ID	The ID of the node to use.
s_start[]	The starting index to use for each dimension of the array within the database node (1D array; i.e., list of indices). The maximum number of dimensions an array is allowed in ADF is 12.
s_end[]	The ending index to use for each dimension of the array within the database node (1D array; i.e., list of indices).
s_stride[]	The stride value to use for each dimension of the array within the database node (1D array; i.e., list of indices).
m_num_dims	The number of dimensions to use in memory.
m_dims[]	The dimension values to use for the array in memory (1D array; i.e., list of indices).
m_start[]	The starting index to use for each dimension of the array in memory (1D array; i.e., list of indices).
m_end[]	The ending index to use for each dimension of the array in memory (1D array; i.e., list of indices).
m_stride[]	The stride value to use for each dimension of the array in memory (1D array; i.e., list of indices).
data	The starting address of the data in memory.
error_return	Error return code. (See Appendix G.)

This routine, `ADF_Write_Data`, provides general purpose write capabilities. It allows offsets and strides within both the data in memory and the node on disk. One set of integer vectors (*s_start*, etc.) is used to describe the mapping of the data within the node, and a second set of integer vectors (*m_start*, etc.) is used to describe the mapping of the desired data within memory.

There can be a significant performance penalty for using `ADF_Write_Data` when compared with `ADF_Write_All_Data`. If performance is a major consideration, it is best to organize data to take advantage of the speed of `ADF_Write_All_Data`.

The data are stored in both memory and on disk in "Fortran ordering." That is, the first index varies the fastest.

`ADF_Write_Data` will not accept "negative" indexing. That is, it is not possible to reverse the ordering of the data from the node into memory.

Be careful when using `ADF_Read_All_Data` to randomly access data that has been written using `ADF_Write_Data`. `ADF_Read_All_Data` takes a starting address in memory and takes N contiguous words from disk, making no assumption as to the order of the data. `ADF_Write_Data` assumes that the data have Fortran-like ordering to navigate through the data on disk and in memory. It assumes that the first dimension varies the fastest.

Note: See the discussion of Fortran character array portability in Appendix E.7.

Example 1

This example uses `ADF_Write_Data` to perform exactly the same task as `ADF_Write_All_Data`. `ADF_Write_All_Data` should be used whenever possible for performance reasons.

```
PROGRAM TEST
C
      PARAMETER (MAXCHR=32)
      PARAMETER (MAXROW=10)
      PARAMETER (MAXCOL=3)
C
      CHARACTER*(MAXCHR) NODNAM,LABL
      CHARACTER*(MAXCHR) DTYPE
      REAL R4ARRI(MAXROW,MAXCOL)
      REAL R4ARRO(MAXROW,MAXCOL)
      INTEGER IDIMI(2),IDIMO(2)
      INTEGER IDBEG(2),IDEND(2),IDINCR(2)
      INTEGER IMBEG(2),IMEND(2),IMINCR(2)
C
C *** NODE IDS
C
      REAL*8 RID,CID
C
C *** OPEN DATABASE
C
      CALL ADFDOPN('db.adf','NEW',' ',RID,IERR)
C
C *** GENERATE SOME DATA
C
      IDIMI(1) = MAXROW
      IDIMI(2) = MAXCOL
      DO 200 ICOL = 1,MAXCOL
         DO 100 IROW = 1,MAXROW
            R4ARRI(IROW,ICOL) = 2.0*ICOL*IROW
  100    CONTINUE
  200 CONTINUE
      PRINT *,' ORIGINAL ARRAY:'
      WRITE(*,300)((R4ARRI(I,J),J=1,MAXCOL),I=1,MAXROW)
  300 FORMAT(3(5X,F10.2))
C
C *** GENERATE A NODE AND PUT DATA IN IT
C     THIS IS EXACTLY EQUIVALENT TO USING ADFWALL
C
      CALL ADFCRE(RID,'NODE 1',CID,IERR)
      CALL ADFSLB(CID,'LABEL FOR NODE 1',IERR)
      CALL ADFPDIM(CID,'R4',2,IDIMI,IERR)
C
      IDBEG(1) = 1
      IDEND(1) = MAXROW
      IDINCR(1) = 1
C
      IDBEG(2) = 1
      IDEND(2) = MAXCOL
      IDINCR(2) = 1
C
      IDIMO(1) = MAXROW
      IDIMO(2) = MAXCOL
C
      IMBEG(1) = 1
      IMEND(1) = MAXROW
      IMINCR(1) = 1
C
      IMBEG(2) = 1
      IMEND(2) = MAXCOL
      IMINCR(2) = 1
C
      CALL ADFWRIT(CID,IDBEG,IDEND,IDINCR,2,IDIMO,IMBEG,
     X             IMEND,IMINCR,R4ARRI,IERR)
      CALL ERRCHK(IERR)
C
C *** GET INFORMATION FROM NODE
C
      CALL ADFGNAM(CID,NODNAM,IERR)
      CALL ADFGLB(CID,LABL,IERR)
      CALL ADFGDT(CID,DTYPE,IERR)
      CALL ADFGND(CID,NDIM,IERR)
      CALL ADFGDV(CID,IDIMO,IERR)
      CALL ADFRALL(CID,R4ARRO,IERR)
      CALL ERRCHK(IERR)
C
      PRINT *,' '
      PRINT *,' NODE NAME            = ',NODNAM
      PRINT *,' LABEL                = ',LABL
      PRINT *,' DATA TYPE            = ',DTYPE
      PRINT *,' NUMBER OF DIMENSIONS = ',NDIM
      PRINT *,' DIMENSIONS           = ',IDIMO
      PRINT *,' ADFRALL DATA:'
      WRITE(*,300)((R4ARRO(I,J),J=1,MAXCOL),I=1,MAXROW)
C
      STOP
      END

C
C ************* SUBROUTINES ****************
C
      SUBROUTINE ERRCHK(IERR)
C
C *** CHECK ERROR CONDITION
C
      CHARACTER*80 MESS
      IF (IERR .GT. 0) THEN
         CALL ADFERR(IERR,MESS)
         PRINT *,MESS
         CALL ABORT('ADF ERROR')
      ENDIF
      RETURN
      END
```

The resulting output is:

```
ORIGINAL ARRAY:
          2.00          4.00          6.00
          4.00          8.00         12.00
          6.00         12.00         18.00
          8.00         16.00         24.00
         10.00         20.00         30.00
         12.00         24.00         36.00
         14.00         28.00         42.00
         16.00         32.00         48.00
         18.00         36.00         54.00
         20.00         40.00         60.00

NODE NAME            = NODE 1
LABEL                = LABEL FOR NODE 1
DATA TYPE            = R4
NUMBER OF DIMENSIONS =            2
DIMENSIONS           =           10           3
ADFRALL DATA:
          2.00          4.00          6.00
          4.00          8.00         12.00
          6.00         12.00         18.00
          8.00         16.00         24.00
         10.00         20.00         30.00
```

```
          12.00          24.00          36.00
          14.00          28.00          42.00
          16.00          32.00          48.00
          18.00          36.00          54.00
          20.00          40.00          60.00
```

Example 2

This example illustrates the capability to write a full matrix to an ADF file and then use `ADF_Write_Data` to rewrite selected portions of the matrix with new data from a much smaller data structure.

```
PROGRAM TEST
C
      PARAMETER (MAXCHR=32)
      PARAMETER (MAXROW=10)
      PARAMETER (MAXCOL=3)
C
      CHARACTER*(MAXCHR) NODNAM,LABL
      CHARACTER*(MAXCHR) DTYPE
      REAL R4ARRI(MAXROW,MAXCOL),R4VEC(MAXCOL)
      REAL R4ARRO(MAXROW,MAXCOL)
      INTEGER IDIMI(2),IDIMO(2),IDIMM(2)
      INTEGER IDBEG(2),IDEND(2),IDINCR(2)
      INTEGER IMBEG(2),IMEND(2),IMINCR(2)
C
C *** NODE IDS
C
      REAL*8 RID,CID
C
C *** OPEN DATABASE
C
      CALL ADFDOPN('db.adf','NEW',' ',RID,IERR)
C
C *** GENERATE SOME DATA
C
      IDIMI(1) = MAXROW
      IDIMI(2) = MAXCOL
      DO 200 ICOL = 1,MAXCOL
         DO 100 IROW = 1,MAXROW
            R4ARRI(IROW,ICOL) = 2.0*ICOL*IROW
  100    CONTINUE
         R4VEC(ICOL) = 2.2*ICOL
  200 CONTINUE
      PRINT *,' ORIGINAL ARRAY:'
      WRITE(*,300)((R4ARRI(I,J),J=1,MAXCOL),I=1,MAXROW)
  300 FORMAT(3(5X,F10.2))
C
C *** GENERATE A NODE AND WRITE THE ARRAY IN IT
C
      CALL ADFCRE(RID,'NODE 1',CID,IERR)
      CALL ADFSLB(CID,'LABEL FOR NODE 1',IERR)
      CALL ADFPDIM(CID,'R4',2,IDIMI,IERR)
      CALL ADFWALL(CID,R4ARRI,IERR)
      CALL ERRCHK(IERR)
C
C *** GET INFORMATION FROM NODE (JUST TO PROVE ITS RIGHT)
C
      CALL ADFGNAM(CID,NODNAM,IERR)
      CALL ADFGLB(CID,LABL,IERR)
      CALL ADFGDT(CID,DTYPE,IERR)
      CALL ADFGND(CID,NDIM,IERR)
      CALL ADFGDV(CID,IDIMO,IERR)
      CALL ADFRALL(CID,R4ARRO,IERR)
      CALL ERRCHK(IERR)
C
      PRINT *,' '
      PRINT *,' NODE NAME            = ',NODNAM
      PRINT *,' LABEL                = ',LABL
      PRINT *,' DATA TYPE            = ',DTYPE
      PRINT *,' NUMBER OF DIMENSIONS = ',NDIM
      PRINT *,' DIMENSIONS           = ',IDIMO
      PRINT *,' ORIGINAL DATA ON DISK:'
      WRITE(*,300)((R4ARRO(I,J),J=1,MAXCOL),I=1,MAXROW)
C
C *** NOW, USING A VECTOR WITH NEW DATA IN IT, SCATTER
C     IT INTO THE DATABASE (THIS MODIFIES THE 5TH ROW
C     OF THE MATRIX)
C
      IDBEG(1)  = 5
      IDEND(1)  = 5
      IDINCR(1) = 1
C
      IDBEG(2)  = 1
      IDEND(2)  = MAXCOL
      IDINCR(2) = 1
C
      NMDIM = 1
      IDIMM(1)  = MAXCOL
      IMBEG(1)  = 1
      IMEND(1)  = MAXCOL
      IMINCR(1) = 1
C
      CALL ADFWRIT(CID,IDBEG,IDEND,IDINCR,
     X             NMDIM,IDIMM,IMBEG,IMEND,IMINCR,
     X             R4VEC,IERR)
      CALL ERRCHK(IERR)
C
C *** NOW PULL THE REVISED ARRAY OFF DISK AND PRINT IT
C
      CALL ADFRALL(CID,R4ARRO,IERR)
      CALL ERRCHK(IERR)
```

```
C
      PRINT *,' '
      PRINT *,' AFTER SCATTER:'
      WRITE(*,300)((R4ARRO(I,J),J=1,MAXCOL),I=1,MAXROW)
C
STOP
END

C
C ************* SUBROUTINES ****************
C
      SUBROUTINE ERRCHK(IERR)
C
C *** CHECK ERROR CONDITION
C
      CHARACTER*80 MESS
      IF (IERR .GT. 0) THEN
         CALL ADFERR(IERR,MESS)
         PRINT *,MESS
         CALL ABORT('ADF ERROR')
      ENDIF
      RETURN
      END
```

The resulting output is:

```
ORIGINAL ARRAY:
          2.00           4.00           6.00
          4.00           8.00          12.00
          6.00          12.00          18.00
          8.00          16.00          24.00
         10.00          20.00          30.00
         12.00          24.00          36.00
         14.00          28.00          42.00
         16.00          32.00          48.00
         18.00          36.00          54.00
         20.00          40.00          60.00

NODE NAME            = NODE 1
LABEL                = LABEL FOR NODE 1
DATA TYPE            = R4
NUMBER OF DIMENSIONS =            2
DIMENSIONS           =           10            3
ORIGINAL DATA ON DISK:
          2.00           4.00           6.00
          4.00           8.00          12.00
          6.00          12.00          18.00
          8.00          16.00          24.00
         10.00          20.00          30.00
         12.00          24.00          36.00
         14.00          28.00          42.00
```

```
          16.00          32.00          48.00
          18.00          36.00          54.00
          20.00          40.00          60.00
 AFTER SCATTER:
           2.00           4.00           6.00
           4.00           8.00          12.00
           6.00          12.00          18.00
           8.00          16.00          24.00
           2.20           4.40           6.60
          12.00          24.00          36.00
          14.00          28.00          42.00
          16.00          32.00          48.00
          18.00          36.00          54.00
          20.00          40.00          60.00
```

`ADF_Write_All_Data` — *Write All the Data to a Node*

`ADF_Write_All_Data (ID,data,error_return)`		
Language	**C**	**Fortran**
Routine Name	`ADF_Write_All_Data`	`ADFWALL`

Input	`const double ID`
	`const char *data`
		`real*8 ID`
		`character*(*) data`

Output	`int *error_return`	`integer error_return`

ID	The ID of the node to use.
data	The starting address of the data in memory.
error_return	Error return code. (See Appendix G.)

This routine, `ADF_Write_All_Data`, writes all data to a node. It copies all the node's data from a contiguous memory space into a contiguous disk space.

The disk performance of `ADF_Write_All_Data` is very good. The routine issues a single write command to the system for the entire data set; therefore, it is as fast as the system can put the data on disk.

Note: See the discussion of Fortran character array portability in Appendix E.7.

Example

See the example for `ADF_Get_Data_Type`.

ADF_Write_Block_Data — *Write a Contiguous Block of Data To a Node*

ADF_Write_Block_Data (ID,b_start,b_end,data,error_return)		
Language	**C**	**Fortran**
Routine Name	`ADF_Write_Block_Data`	`ADFWBLK`

Input	`const double ID`
	`const long b_start`
	`const long b_end`
	`char *data`
		`real*8 ID`
		`integer b_start`
		`integer b_end`
		`character*(*) data`

Output	`int *error_return`	`integer error_return`

ID	The ID of the node to use.
b_start	The starting point of the block in token space.
b_end	The ending point of the block in token space.
data	The starting address of the data in memory.
error_return	Error return code. (See Appendix G.)

This routine, `ADF_Write_Block_Data`, writes a contiguous block of data from memory to a node.

Note: See the discussion of Fortran character array portability in Appendix E.7.

I.7 Miscellaneous Routines

`ADF_Flush_to_Disk` — *Flush the Data to the Disk*

`ADF_Flush_to_Disk (ID,error_return)`		
Language	**C**	**Fortran**
Routine Name	`ADF_Flush_to_Disk`	`ADFFTD`
Input	`const double ID`	`real*8 ID`
Output	`int *error_return`	`integer error_return`

ID	The ID of the node to use.
error_return	Error return code. (See Appendix G.)

This routine, `ADF_Flush_to_Disk`, flushes data to disk; it is used to force any modified information to be flushed to the physical disk. This ensures that data will not be lost if a program aborts. The control of when to flush all data to disk is provided to the user rather than flushing the data every time it is modified, which would result in reduced performance.

`ADF_Database_Garbage_Collection` — *Flush the Data to the Disk*

`ADF_Database_Garbage_Collection (ID,error_return)`

Language	**C**	**Fortran**
Routine Name	`ADF_Database_Garbage_Collection`	`ADFDGC`
Input	`const double ID`	`real*8 ID`
Output	`int *error_return`	`integer error_return`

ID	The ID of the node to use.
error_return	Error return code. (See Appendix G.)

This routine, `ADF_Database_Garbage_Collection`, redistributes the data within a file to use free space within the file. This free space is not located at the end of the file, and it may have been created during node deletions or other file operations. Neighboring free spaces will be merged.

Note: For better file compaction, a utility could be written to copy an ADF file into a newly created ADF file without wasted space.

Note: This routine is currently not implemented.

`ADF_Error_Message` — *Get a Description of the Error*

`ADF_Error_Message (error_code,error_string)`		
Language	**C**	**Fortran**
Routine Name	`ADF_Error_Message`	`ADFERR`
Input	`const int error_code`	`integer error_code`
Output	`char *error_string`	`character*(*) error_string`

error_code	The error return code from any ADF routine. (See Appendix G.)
error_string	An 80-byte description for the specified error. If `ADF_Error_Message` cannot find a message corresponding to the input `error_code`, the error string "`Unknown error#`*nnn*" will be returned.

This routine, `ADF_Error_Message`, returns a textual error message when given the error return code from any ADF routine.

Example

See the example for `ADF_Set_Error_State`.

ADF_Set_Error_State — *Set the Error State Flag*

ADF_Set_Error_State (error_state,error_return)		
Language	**C**	**Fortran**
Routine Name	ADF_Set_Error_State	ADFSES
Input	const int error_state	integer error_state
Output	int *error_return	integer error_return

error_state	Flag specifying the action to take when an error occurs; 0 to return the error code and continue, 1 to abort. The default is 0.
error_return	Error return code. (See Appendix G.)

This routine, `ADF_Set_Error_State`, sets the error state flag. This flag controls the error handling convention for all ADF routines: either return the error codes or print an error message and abort the program on error. The flag covers all open ADF files associated with the current program, and it is not done on a file-by-file basis. The default state for the ADF interface is to return error codes and not abort.

Example

```
PROGRAM TEST
C
      PARAMETER (MAXCHR=32)
C
      CHARACTER*(MAXCHR) NODNAM
      CHARACTER*(80) MESS
C
C *** NODE IDS
C
      REAL*8 RID,CID1,CID2
C
C *** OPEN DATABASE
C
      CALL ADFDOPN('db.adf','NEW',' ',RID,IERR)
C
C *** CREATE 2 NODES
C
      CALL ADFCRE(RID,'NODE 1',CID1,IERR)
      CALL ADFCRE(RID,'NODE 2',CID2,IERR)
C
      CALL ADFGES(IDEFS,IERR)
      PRINT *,' DEFAULT ERROR STATE = ',IDEFS
C
C *** REQUEST NODE NAME FOR A NODE THAT DOES NOT EXIST
C
      PRINT *,' *** ON ERROR CONTINUE'
      CALL ADFGNAM(CID3,NODNAM,IERR)
      CALL ADFERR(IERR,MESS)
      PRINT *,'     ADF ERROR OCURRED, MESSAGE: ',MESS
      PRINT *,' '
C
C *** SET ABORT ON ERROR FLAG
C
      INEWS = 1
      CALL ADFSES(INEWS,IERR)
      PRINT *,' *** ABORT ON ERROR SET'
C
C *** REQUEST NODE NAME FOR A NODE THAT DOES NOT EXIST
C
      CALL ADFGNAM(CID3,NODNAM,IERR)
      PRINT *,' HELLO WORLD'
C
      STOP
      END
```

The resulting output is:

```
DEFAULT ERROR STATE =           0
  *** ON ERROR CONTINUE
      ADF ERROR OCURRED, MESSAGE:
ADF 10: ADF file index out of legal range.

  *** ABORT ON ERROR SET
ADF 10: ADF file index out of legal range.
ADF Aborted: Exiting
```

`ADF_Get_Error_State` — *Get the Error State*

ADF_Get_Error_State (error_state,error_return)		
Language	**C**	**Fortran**
Routine Name	`ADF_Get_Error_State`	`ADFGES`
Input		

Output	`int *error_state`
	`int *error_return`
		`integer error_state`
		`integer error_return`

error_state	Flag specifying the action to take when an error occurs; 0 to return the error code and continue, 1 to abort. The default is 0.
error_return	Error return code. (See Appendix G.)

This routine, `ADF_Get_Error_State`, returns the currently set error state.

Example

```
PROGRAM TEST
C
C *** NODE IDS
C
      REAL*8 RID
C
C *** OPEN DATABASE
C
      CALL ADFDOPN('db.adf','NEW',' ',RID,IERR)
C
C *** CHECK THE DEFAULT ERROR STATE
C
      CALL ADFGES(IDEFS,IERR)
C
C *** SET THE ERROR STATE TO ABORT ON ERROR
C
      CALL ADFSES(1,IERR)
C
C *** MAKE SURE STATE WAS SET AS DESIRED
C
      CALL ADFGES(NDEFS,IERR)
C
C *** PRINT OUT RESULTS
C
      PRINT *,' DEFAULT ERROR STATE = ',IDEFS
      PRINT *,' RESET ERROR STATE   = ',NDEFS
C
      STOP
      END
```

The resulting output is:

```
   DEFAULT ERROR STATE =            0
   RESET ERROR STATE   =            1
```

`ADF_Database_Version` — *Get the Version Number of the ADF Library That Created the ADF Database*

```
ADF_Database_Version (root_ID,version,creation_date,modification_date,
                      error_return)
```

Language	C	Fortran
Routine Name	`ADF_Database_Version`	`ADFDVER`
Input	`constant double root_ID`	`real*8 root_ID`

Output	`char *version`
	`char *creation_date`
	`char *modification_date`
	`int *error_return`
		`character*(*) version`
		`character*(*) creation_date`
		`character*(*) modifica-tion_date`
		`integer error_return`

root_ID	The ID of the root node in the ADF file.
version	A 32-byte character string containing the version ID.
creation_date	A 32-byte character string containing the creation date of the file.
modification_date	A 32-byte character string containing the last modification date of the file.
error_return	Error return code. (See Appendix G.)

This routine, `ADF_Database_Version`, returns the version number of the ADF library that created an ADF database, the file creation date and time, and the last modification date and time. A modified ADF database will take on the version ID of the current ADF library version if it is higher than the version indicated in the file. On return, "version" contains a six-character string. The meaning of the characters are described in detail in Appendix D.

Example

```
PROGRAM TEST
C
      PARAMETER (MAXCHR=32)
C
      CHARACTER*(MAXCHR) NODNAM
      CHARACTER*(MAXCHR) CVER,LVER,CDATE,MDATE
      CHARACTER*(80) MESS
C
C *** NODE IDS
C
      REAL*8 RID,CID1,CID2
C
C *** OPEN DATABASE
C
      CALL ADFDOPN('db.adf','NEW',' ',RID,IERR)
```

```
C
C *** CREATE 2 NODES
C
      CALL ADFCRE(RID,'NODE 1',CID1,IERR)
      CALL ADFCRE(RID,'NODE 2',CID2,IERR)
C
      CALL ADFDVER(RID,CVER,CDATE,MDATE,IERR)
      CALL ADFLVER(LVER,IERR)
      PRINT *,' VERSION INFORMATION:'
      PRINT *,'    ADF LIBRARY USED FOR CREATION: ',CVER
      PRINT *,'    CREATION DATE                : ',CDATE
      PRINT *,'    MODIFICATION DATE            : ',MDATE
      PRINT *,'    ADF LIBRARY BEING USED       : ',LVER
C
      STOP
      END
```

The resulting output is:

```
VERSION INFORMATION:
   ADF LIBRARY USED FOR CREATION: ADF Database Version A01007
   CREATION DATE                : Thu Apr 24 15:41:55 1997
   MODIFICATION DATE            : Thu Apr 24 15:41:55 1997
   ADF LIBRARY BEING USED       : ADF Library  Version C01
```

`ADF_Library_Version` — *Get the Version Number of the ADF Library That the Application Program is Currently Using*

`ADF_Library_Version (version,error_return)`		
Language	**C**	**Fortran**
Routine Name	`ADF_Library_Version`	`ADFLVER`
Input		

Output	`char *version`
	`int *error_return`
		`character*(*) version`
		`integer error_return`

version	A 32-byte character string containing the ADF library version ID information.
error_return	Error return code. (See Appendix G.)

This routine, `ADF_Library_Version`, gets the ADF library version number. This is the version number of the ADF library that the application program is currently using. For this routine, the format of the version ID is "`ADF Library Version A`*XXXxxx*".

Example

See the example for `ADF_Database_Version`.

J Sample Fortran Program

The following Fortran program builds the ADF file shown in Figure 1 on p. 4.

```
      PROGRAM TEST
C
C     SAMPLE ADF TEST PROGRAM TO BUILD ADF FILES ILLUSTRATED
C     IN THE EXAMPLE DATABASE FIGURE
C
      PARAMETER (MAXCHR=32)
C
      CHARACTER*(MAXCHR) TSTLBL,DTYPE
      CHARACTER*(MAXCHR) FNAM,PATH
C
      REAL*8 RID,PID,CID,TMPID,RIDF2
      REAL A(4,3),B(4,3)
      INTEGER IC(6),ID(6)
      INTEGER IERR
      INTEGER IDIM(2),IDIMA(2),IDIMC,IDIMD
C
      DATA A /1.1,2.1,3.1,4.1,
     X        1.2,2.2,3.2,4.2,
     X        1.3,2.3,3.3,4.3/
      DATA IDIMA /4,3/
C
      DATA IC /1,2,3,4,5,6/
      DATA IDIMC /6/
C
C     SET ERROR FLAG TO ABORT ON ERROR
C
      CALL ADFSES(1,IERR)
C
C *** 1.) OPEN 1ST DATABASE (ADF_FILE_TWO.ADF)
C     2.) CREATE THREE NODES AT FIRST LEVEL
C     3.) PUT LABEL ON NODE F3
C     4.) PUT DATA IN F3
C     5.) CREATE TWO NODES BELOW F3
C     6.) CLOSE DATABASE
C
      CALL ADFDOPN('adf_file_two.adf','NEW',' ',RID,IERR)
      RIDF2 = RID
      CALL ADFCRE(RID,'F1',TMPID,IERR)
      CALL ADFCRE(RID,'F2',TMPID,IERR)
      CALL ADFCRE(RID,'F3',PID,IERR)
      CALL ADFSLB(PID,'LABEL ON NODE F3',IERR)
      CALL ADFPDIM(PID,'R4',2,IDIMA,IERR)
      CALL ADFWALL(PID,A,IERR)
C
      CALL ADFCRE(PID,'F4',CID,IERR)
C
```

```
      CALL ADFCRE(PID,'F5',CID,IERR)
C
      CALL ADFDCLO(RID,IERR)
C
C *** 1.) OPEN 2ND DATABASE
C     2.) CREATE NODES
C     3.) PUT DATA IN N13
C
      CALL ADFDOPN('adf_file_one.adf','NEW',' ',RID,IERR)
C
C     THREE NODES UNDER ROOT
C
      CALL ADFCRE(RID,'N1',TMPID,IERR)
      CALL ADFCRE(RID,'N2',TMPID,IERR)
      CALL ADFCRE(RID,'N3',TMPID,IERR)
C
C     THREE NODES UNDER N1 (TWO REGULAR AND ONE LINK)
C
      CALL ADFGNID(RID,'N1',PID,IERR)
      CALL ADFCRE(PID,'N4',TMPID,IERR)
      CALL ADFLINK(PID,'L3','adf_file_two.adf','/F3',TMPID,IERR)
      CALL ADFCRE(PID,'N5',TMPID,IERR)
C
C     TWO NODES UNDER N4
C
      CALL ADFGNID(PID,'N4',CID,IERR)
      CALL ADFCRE(CID,'N6',TMPID,IERR)
      CALL ADFCRE(CID,'N7',TMPID,IERR)
C
C     ONE NODE UNDER N6
C
      CALL ADFGNID(RID,'/N1/N4/N6',PID,IERR)
      CALL ADFCRE(PID,'N8',TMPID,IERR)
C
C     THREE NODES UNDER N3
C
      CALL ADFGNID(RID,'N3',PID,IERR)
      CALL ADFCRE(PID,'N9',TMPID,IERR)
      CALL ADFCRE(PID,'N10',TMPID,IERR)
      CALL ADFCRE(PID,'N11',TMPID,IERR)
C
C     TWO NODES UNDER N9
C
      CALL ADFGNID(PID,'N9',CID,IERR)
      CALL ADFCRE(CID,'N12',TMPID,IERR)
      CALL ADFCRE(CID,'N13',TMPID,IERR)
C
C     PUT LABEL AND DATA IN N13
C
      CALL ADFSLB(TMPID,'LABEL ON NODE N13',IERR)
      CALL ADFPDIM(TMPID,'I4',1,IDIMC,IERR)
      CALL ADFWALL(TMPID,IC,IERR)
C
C     TWO NODES UNDER N10
C
      CALL ADFGNID(RID,'/N3/N10',PID,IERR)
      CALL ADFLINK(PID,'L1',' ','/N3/N9/N13',TMPID,IERR)
      CALL ADFCRE(PID,'N14',TMPID,IERR)
C
C     TWO NODES UNDER N11
C
      CALL ADFGNID(RID,'/N3/N11',PID,IERR)
      CALL ADFLINK(PID,'L2',' ','/N3/N9/N13',TMPID,IERR)
      CALL ADFCRE(PID,'N15',TMPID,IERR)
C
C *** READ AND PRINT DATA FROM NODES
C     1.) NODE F5 THROUGH LINK L3
C
      CALL ADFGNID(RID,'/N1/L3',PID,IERR)
      CALL ADFGLB(PID,TSTLBL,IERR)
      CALL ADFGDT(PID,DTYPE,IERR)
      CALL ADFGND(PID,NUMDIM,IERR)
      CALL ADFGDV(PID,IDIM,IERR)
      CALL ADFRALL(PID,B,IERR)
      PRINT *,' NODE F3 THROUGH LINK L3:'
      PRINT *,'   LABEL       = ',TSTLBL
      PRINT *,'   DATA TYPE   = ',DTYPE
      PRINT *,'   NUM OF DIMS = ',NUMDIM
      PRINT *,'   DIM VALS    = ',IDIM
      PRINT *,'   DATA:'
      WRITE(*,100)((B(J,I),I=1,3),J=1,4)
  100 FORMAT(5X,3F10.2)
C
C     2.) N13
C
      CALL ADFGNID(RID,'N3/N9/N13',PID,IERR)
      CALL ADFGLB(PID,TSTLBL,IERR)
      CALL ADFGDT(PID,DTYPE,IERR)
      CALL ADFGND(PID,NUMDIM,IERR)
      CALL ADFGDV(PID,IDIMD,IERR)
      CALL ADFRALL(PID,ID,IERR)
      PRINT *,' '
      PRINT *,' NODE N13:'
      PRINT *,'   LABEL       = ',TSTLBL
      PRINT *,'   DATA TYPE   = ',DTYPE
      PRINT *,'   NUM OF DIMS = ',NUMDIM
      PRINT *,'   DIM VALS    = ',IDIMD
      PRINT *,'   DATA:'
      WRITE(*,200)(ID(I),I=1,6)
  200 FORMAT(5X,6I6)
C
C     3.) N13 THROUGH L1
```

```
C
      CALL ADFGNID(RID,'N3/N10/L1',TMPID,IERR)
      CALL ADFGLB(TMPID,TSTLBL,IERR)
      CALL ADFRALL(TMPID,ID,IERR)
      PRINT *,' '
      PRINT *,' NODE N13 THROUGH LINK L1:'
      PRINT *,'   LABEL       = ',TSTLBL
      PRINT *,'   DATA:'
      WRITE(*,200)(ID(I),I=1,6)
C
C     4.) N13 THROUTH L2
C
      CALL ADFGNID(RID,'N3/N11/L2',CID,IERR)
      CALL ADFGLB(CID,TSTLBL,IERR)
      CALL ADFRALL(CID,ID,IERR)
      PRINT *,' '
      PRINT *,' NODE N13 THROUGH LINK L2:'
      PRINT *,'   LABEL       = ',TSTLBL
      PRINT *,'   DATA:'
      WRITE(*,200)(ID(I),I=1,6)
C
C     PRINT LIST OF CHILDREN UNDER ROOT NODE
C
      CALL PRTCLD(RID)
C
C     PRINT LIST OF CHILDREN UNDER N3
C
      CALL ADFGNID(RID,'N3',PID,IERR)
      CALL PRTCLD(PID)
C
C     REOPEN ADF_FILE_TWO AND GET NEW ROOT ID
C
      CALL ADFDOPN('adf_file_two.adf','OLD',' ',RID,IERR)
      PRINT *,' '
      PRINT *,' COMPARISON OF ROOT ID: '
      PRINT *,' ADF_FILE_TWO.ADF ORIGINAL ROOT ID = ',RIDF2
      PRINT *,' ADF_FILE_TWO.ADF NEW ROOT ID      = ',RID
C
      STOP
      END
C
C ************* SUBROUTINES ****************
C
      SUBROUTINE PRTCLD(PID)
C
C *** PRINT TABLE OF CHILDREN GIVEN A PARENT NODE-ID
C
      PARAMETER (MAXCLD=10)
      PARAMETER (MAXCHR=32)
      REAL*8 PID
      CHARACTER*(MAXCHR) NODNAM,NDNMS(MAXCLD)
      CALL ADFGNAM(PID,NODNAM,IERR)
      CALL ADFNCLD(PID,NUMC,IERR)
      WRITE(*,120)NODNAM,NUMC
  120 FORMAT(/,' PARENT NODE NAME = ',A,/,
     X       '     NUMBER OF CHILDREN = ',I2,/,
     X       '     CHILDREN NAMES:')
      NLEFT = NUMC
      ISTART = 1
C     --- TOP OF DO-WHILE LOOP
  130 CONTINUE
         CALL ADFCNAM(PID,ISTART,MAXCLD,LEN(NDNMS),
     X                NUMRET,NDNMS,IERR)
         WRITE(*,140)(NDNMS(K),K=1,NUMRET)
  140    FORMAT(8X,A)
         NLEFT = NLEFT - MAXCLD
         ISTART = ISTART + MAXCLD
      IF (NLEFT .GT. 0) GO TO 130
      RETURN
      END
```

The resulting output is:

```
NODE F3 THROUGH LINK L3:
  LABEL       = LABEL ON NODE F3
  DATA TYPE   = R4
  NUM OF DIMS = 2
  DIM VALS    = 4 3
  DATA:
          1.10      1.20      1.30
          2.10      2.20      2.30
          3.10      3.20      3.30
          4.10      4.20      4.30

NODE N13:
  LABEL       = LABEL ON NODE N13
  DATA TYPE   = I4
  NUM OF DIMS = 1
  DIM VALS    = 6
  DATA:
         1      2      3      4      5      6

NODE N13 THROUGH LINK L1:
  LABEL       = LABEL ON NODE N13
  DATA:
         1      2      3      4      5      6

NODE N13 THROUGH LINK L2:
  LABEL       = LABEL ON NODE N13
  DATA:
         1      2      3      4      5      6
```

```
PARENT NODE NAME = ADF MotherNode
    NUMBER OF CHILDREN =  3
    CHILDREN NAMES:
       N1
       N2
       N3

PARENT NODE NAME = N3
    NUMBER OF CHILDREN =  3
    CHILDREN NAMES:
       N9
       N10
       N11

COMPARISON OF ROOT ID:
  ADF_FILE_TWO.ADF ORIGINAL ROOT ID = 1.2653021189994324-320
  ADF_FILE_TWO.ADF NEW ROOT ID      = 4.7783097267391979-299
```

K Sample C Program

The following C program builds the ADF file shown in Figure 1 on p. 4.

```
/*
   Sample ADF test program to build adf files illustrated
   in example database figure.
*/

#include <stdio.h>
#include <ctype.h>
#include <string.h>

#include "../include/ADF.h"

void print_child_list(double node_id);

main ()
{
  /* --- Node header character strings */
   char label[ ADF_LABEL_LENGTH+1];
   char data_type[ADF_DATA_TYPE_LENGTH+1];
   char file_name[ADF_FILENAME_LENGTH+1];
   char path[ADF_MAX_LINK_DATA_SIZE+1];

  /* --- Node id variables */
   double root_id,parent_id,child_id,tmp_id,root_id_file2;

  /* --- Data to be stored in database */
   float a[3][4] = {
                    1.1,2.1,3.1,4.1,
                    1.2,2.2,3.2,4.2,
                    1.3,2.3,3.3,4.3
                   };
   int a_dimensions[2] = {4,3};

   int c[6] = {1,2,3,4,5,6};
   int c_dimension = 6;

  /* --- miscellaneous variables */
   int error_flag, i, j;
   int error_state = 1;
   int num_dims, dim_d, d[6], dims_b[2];
   float b[3][4];

/* ------ begin source code ----- */

  /* --- set database error flag to abort on error */
   ADF_Set_Error_State(error_state,&error_flag);
  /* -------- build file: adf_file_two.adf ---------- */
  /* --- 1.) open database
         2.) create three nodes at first level
         3.) put label on node f3
         4.) put some data in node f3
         5.) create two nodes below f3
         6.) close database */

   ADF_Database_Open("adf_file_two.adf","new"," ",&root_id,&error_flag);
   root_id_file2 = root_id;
   ADF_Create(root_id,"f1",&tmp_id,&error_flag);
   ADF_Create(root_id,"f2",&tmp_id,&error_flag);
   ADF_Create(root_id,"f3",&parent_id,&error_flag);
   ADF_Set_Label(parent_id,"label on node f3",&error_flag);

   ADF_Put_Dimension_Information(parent_id,"R4",2,a_dimensions,&error_flag);
   ADF_Write_All_Data(parent_id,(char *)(a),&error_flag);

   ADF_Create(parent_id,"f4",&child_id,&error_flag);
   ADF_Create(parent_id,"f5",&child_id,&error_flag);
   ADF_Database_Close(root_id,&error_flag);

  /* -------- build file: adf_file_one.adf ---------- */
  /* open database and create three nodes at first level */
   ADF_Database_Open("adf_file_one.adf","new"," ",&root_id,&error_flag);
   ADF_Create(root_id,"n1",&tmp_id,&error_flag);
   ADF_Create(root_id,"n2",&tmp_id,&error_flag);
   ADF_Create(root_id,"n3",&tmp_id,&error_flag);

  /* put three nodes under n1 (two regular and one link) */
   ADF_Get_Node_ID(root_id,"n1",&parent_id,&error_flag);
   ADF_Create(parent_id,"n4",&tmp_id,&error_flag);
   ADF_Link(parent_id,"l3","adf_file_two.adf","/f3",&tmp_id,&error_flag);
   ADF_Create(parent_id,"n5",&tmp_id,&error_flag);

  /* put two nodes under n4 */
   ADF_Get_Node_ID(parent_id,"n4",&child_id,&error_flag);
   ADF_Create(child_id,"n6",&tmp_id,&error_flag);
   ADF_Create(child_id,"n7",&tmp_id,&error_flag);

  /* put one nodes under n6 */
   ADF_Get_Node_ID(root_id,"/n1/n4/n6",&parent_id,&error_flag);
   ADF_Create(parent_id,"n8",&tmp_id,&error_flag);

  /* put three nodes under n3 */
   ADF_Get_Node_ID(root_id,"n3",&parent_id,&error_flag);
   ADF_Create(parent_id,"n9",&tmp_id,&error_flag);
   ADF_Create(parent_id,"n10",&tmp_id,&error_flag);
   ADF_Create(parent_id,"n11",&tmp_id,&error_flag);

  /* put two nodes under n9 */
```

```
 ADF_Get_Node_ID(parent_id,"n9",&child_id,&error_flag);
 ADF_Create(child_id,"n12",&tmp_id,&error_flag);
 ADF_Create(child_id,"n13",&tmp_id,&error_flag);

/* put label and data in n13 */
 ADF_Set_Label(tmp_id,"Label on Node n13",&error_flag);
 ADF_Put_Dimension_Information(tmp_id,"i4",1,&c_dimension,&error_flag);
 ADF_Write_All_Data(tmp_id,(char *)(c),&error_flag);

/* put two nodes under n10 (one normal, one link) */
 ADF_Get_Node_ID(root_id,"/n3/n10",&parent_id,&error_flag);
 ADF_Link(parent_id,"l1"," ","/n3/n9/n13",&tmp_id,&error_flag);
 ADF_Create(parent_id,"n14",&tmp_id,&error_flag);

/* put two nodes under n11 (one normal, one link) */
 ADF_Get_Node_ID(root_id,"/n3/n11",&parent_id,&error_flag);
 ADF_Link(parent_id,"l2"," ","/n3/n9/n13",&tmp_id,&error_flag);
 ADF_Create(parent_id,"n15",&tmp_id,&error_flag);

/* ----------------- finished building adf_file_one.adf ------------- */

/* ------------- access and print data --------------- */

/* access data in node f3 (adf_file_two.adf) through link l3 */
 ADF_Get_Node_ID(root_id,"/n1/l3",&tmp_id,&error_flag);
 ADF_Get_Label(tmp_id,label,&error_flag);
 ADF_Get_Data_Type(tmp_id,data_type,&error_flag);
 ADF_Get_Number_of_Dimensions(tmp_id,&num_dims,&error_flag);
 ADF_Get_Dimension_Values(tmp_id,dims_b,&error_flag);
 ADF_Read_All_Data(tmp_id,(char *)(b),&error_flag);
 printf (" node f3 through link l3:\n");
 printf ("   label       = %s\n",label);
 printf ("   data_type   = %s\n",data_type);
 printf ("   num of dims = %5d\n",num_dims);
 printf ("   dim vals    = %5d %5d\n",dims_b[0],dims_b[1]);
 printf ("   data:\n");
 for (i=0; i<=3; i++)
   {
     for (j=0; j<=2; j++)
       {
         printf("     %10.2f",b[j][i]);
       };
     printf("\n");
   };

/* access data in node n13 */
 ADF_Get_Node_ID(root_id,"/n3/n9/n13",&tmp_id,&error_flag);
 ADF_Get_Label(tmp_id,label,&error_flag);
 ADF_Get_Data_Type(tmp_id,data_type,&error_flag);
 ADF_Get_Number_of_Dimensions(tmp_id,&num_dims,&error_flag);
 ADF_Get_Dimension_Values(tmp_id,&dim_d,&error_flag);
```

```
     ADF_Read_All_Data(tmp_id,(char *)(d),&error_flag);
     printf (" node n13:\n");
     printf ("   label       = %s\n",label);
     printf ("   data_type   = %s\n",data_type);
     printf ("   num of dims = %5d\n",num_dims);
     printf ("   dim val     = %5d\n",dim_d);
     printf ("   data:\n");
     for (i=0; i<=5; i++)
       {
         printf("     %-4d",d[i]);
       };
     printf("\n\n");

    /* access data in node n13 through l1 */
     ADF_Get_Node_ID(root_id,"/n3/n10/l1",&tmp_id,&error_flag);
     ADF_Get_Label(tmp_id,label,&error_flag);
     ADF_Read_All_Data(tmp_id,(char *)(d),&error_flag);
     printf (" node n13 through l1:\n");
     printf ("   label       = %s\n",label);
     printf ("   data:\n");
     for (i=0; i<=5; i++)
       {
         printf("     %-4d",d[i]);
       };
     printf("\n\n");

    /* access data in node n13 through l2 */
     ADF_Get_Node_ID(root_id,"/n3/n11/l2",&tmp_id,&error_flag);
     ADF_Get_Label(tmp_id,label,&error_flag);
     ADF_Read_All_Data(tmp_id,(char *)(d),&error_flag);
     printf (" node n13 through l2:\n");
     printf ("   label       = %s\n",label);
     printf ("   data:\n");
     for (i=0; i<=5; i++)
       {
         printf("     %-4d",d[i]);
       };
     printf("\n\n");

    /* print list of children under root node */
     print_child_list(root_id);

    /* print list of children under n3 */
     ADF_Get_Node_ID(root_id,"/n3",&tmp_id,&error_flag);
     print_child_list(tmp_id);

    /* re-open adf_file_two and get new root id */
     ADF_Database_Open("adf_file_two.adf","old"," ",&root_id,&error_flag);
     printf (" Comparison of root id:\n");
     printf ("   adf_file_two.adf original root id = %x\n",root_id_file2);
     printf ("   adf_file_two.adf new      root id = %x\n",root_id);
```

```
}

void print_child_list(double node_id)
{

/*
   print table of children given a parent node-id
*/
   char node_name[ADF_NAME_LENGTH+1];
   int i, num_children, num_ret, error_return;

   ADF_Get_Name(node_id,node_name,&error_return);
   ADF_Number_of_Children(node_id,&num_children,&error_return);
   printf ("Parent Node Name = %s\n",node_name);
   printf ("  Number of Children = %2d\n",num_children);
   printf ("  Children Names:\n");
   for (i=1; i<=num_children; i++)
     {
       ADF_Children_Names(node_id,i,1,ADF_NAME_LENGTH+1,
           &num_ret,node_name,&error_return);
       printf ("     %s\n",node_name);                               };
    printf ("\n");
}
```

The resulting output is:

```
node f3 through link l3:
  label       = label on node f3
  data_type   = R4
  num of dims =    2
  dim vals    =    4     3
  data:
          1.10          1.20          1.30
          2.10          2.20          2.30
          3.10          3.20          3.30
          4.10          4.20          4.30
node n13:
  label       = Label on Node n13
  data_type   = i4
  num of dims = 1
  dim val     = 6
  data:
    1        2        3        4        5        6

node n13 through l1:
  label       = Label on Node n13
  data:
    1        2        3        4        5        6

node n13 through l2:
  label       = Label on Node n13
```

```
    data:
      1        2        3        4        5        6

 Parent Node Name = ADF MotherNode
   Number of Children =  3
   Children Names:
      n1
      n2
      n3

 Parent Node Name = n3
   Number of Children =  3
   Children Names:
      n9
      n10
      n11

 Comparison of root id:
   adf_file_two.adf original root id = 18
   adf_file_two.adf new      root id = 2a
```